A Forward-secure Efficient Two-factor Authentication Protocol

Abstract. Two-factor authentication (2FA) schemes that rely on a combination of knowledge factors (e.g., PIN) and device possession have gained popularity. Some of these schemes remain secure even against strong adversaries that (a) observe the traffic between a client and server, and (b) have physical access to the client's device, or its PIN, or breach the server. However, these solutions have several shortcomings; namely, they (i) require a client to remember *multiple* secret values to prove its identity, (ii) involve *several modular exponentiations*, and (iii) are in the *non-standard* random oracle model. In this work, we present a 2FA protocol that resists such a strong adversary while addressing the above shortcomings. Our protocol requires a client to remember only a single secret value/PIN, does not involve any modular exponentiations, and is in a standard model. It is the first one that offers these features without using trusted chipsets. This protocol also imposes up to 40% lower communication overhead than the state-of-the-art solutions do.

1 Introduction

The adoption of online services, such as online banking and e-commerce, has been swiftly increasing, and so has the effort of adversaries to gain unauthorised access to such services. For clients to prove their identity to a remote service provider, they provide a piece of evidence, called an "authentication factor". Authentication factors can be based on (i) knowledge factors, e.g., PIN or password, (ii) possession factors, e.g., access card or physical hardware token, or (iii) inherent factors, e.g., fingerprint. Knowledge factors are still the most predominant factors used for authentication [6,15]. The knowledge factors themselves are not strong enough to adequately prevent impersonation [29,15]. Multi-factor authentication methods that depend on more than one factor are more difficult to compromise than single-factor methods. Recently (on January 26, 2022), the "Executive Office of the US President" released a memorandum requiring the Federal Government's agencies to meet specific cybersecurity standards, including the use of multi-factor authentication, to reinforce the Government's defences against increasingly sophisticated threat campaigns [23]. Among multi-factor authentication schemes, two-factor authentication (2FA) methods, including those that rely on a combination of PIN and device possession, have attracted special attention, from banks and e-commerce, due to their low cost and good usability.

Researchers and companies have proposed various 2FA solutions based on a combination of PIN and device possession. Some of these solutions offer a strong security guarantee against an adversary which may (a) observe the communication between a client and server, and (b) have physical access to the client's

device, or its PIN, or breaches the server. These solutions do not rely on trusted chipsets and still ensure that even such a strong adversary cannot succeed during the authentication. Nevertheless, these solutions (i) require a client to remember multiple secret values (instead of a single PIN) to prove its identity which ultimately harms these solutions' usability, (ii) involve several modular exponentiations that make the device battery power run out fast, and (iii) are in the non-standard random oracle model.

Our Contributions. In this work, we present a 2FA protocol that resists the strong adversary above while addressing the aforementioned shortcomings and imposing a lower communication cost. Specifically, our protocol:

- requires a client to remember only a single PIN.
- allows the device to generate a short authentication message.
- does not involve any modular exponentiations.
- is in a standard model.
- imposes up to 40% lower communication costs than the state-of-the-art does.

To attain its goals, our protocol does not use any trusted chipsets; instead, it relies on a novel combination of the following two approaches. First, it requires only the server (not the device) to verify a client's PIN. This allows separating the location where the PIN's secret key is stored from the location where the authenticator itself is stored. This approach ensures that an adversary cannot retrieve the PIN, even if it penetrates either location. Second, it (a) requires that the server and device use key-evolving symmetric-key encryption (i.e., a combination of forward-secure pseudorandom bit generator and authenticated encryption) to encrypt sensitive messages they exchange, and (b) requires that used keys be discarded right after their use. This approach ensures the secrecy of the communication between the parties and guarantees that the adversary cannot learn the PIN, even if it eavesdrops on the parties' communication and breaks into the device or server. We formally prove the security of this protocol.

2 Related Work

In this section, first, we briefly discuss the common approaches for generating a One-Time Password (OTP) which yields from a combination of a PIN and a hardware token's parameters. Then, we provide an overview of hardware token variants. We refer readers to Appendix A for a more detailed discussion.

2.1 Common Approaches for Generating OTP

In the authentication that relies on a combination of knowledge and possession factors, once the client enters the secret into the hardware token, the device combines this secret with the output of one of the following methods to generate a unique OTP: (i) a random challenge, sent by the server to the device; (ii) an internal counter maintained by the server and device, or (iii) the current accurate time, kept by the server and device. There exist 2FA solutions (including ours) that employ a combination of the above approaches.

2.2 Variants of OTP Hardware Tokens

Connected Tokens. This type of token requires a client to physically connect the token to their computer via which the client is authenticating. After that, the device transmits the authentication information to the computer. USB tokens and smart cards are two popular token technologies in this category. Various companies including Google and "Fast IDentity Online" (FIDO) Alliance have developed USB hardware tokens. However, researchers have discovered many vulnerabilities within this standard, e.g., in [26,9,22,12].

Since the introduction of smart card technology, there have been numerous smart card-based 2FA protocols (e.g., in [14,33,27,19]). But, they use either public-key cryptography which imposes a high computation cost or tamper-proof secure chipsets embedded in the card which would increase the device's cost.

Disconnected Tokens. This type of token does not have a physical connection to a client's computer, making them more convenient than connected tokens. Two main categories of disconnected tokens are (1) dedicated hardware-based tokens (e.g., in [2,24,30]) and (2) mobile phone-based tokens (e.g., in [1,20,21]). The first category includes RSA SecureID [2], OneSpan Digipass 770 [24], and Thales Gemalto SWYS QR Token Eco [30]. In RSA SecureID, an adversary who has access to the device can generate the OTP by extracting the secret key stored on the device. The advantage of Digipass 770 and Thales Gemalto to RSA SecureID is that they let clients see the transaction details through the token which gives them more information about the transaction they approve, so phishing becomes harder. Our investigation suggests that Digipass 770 and Thales Gemalto also *locally store and verify* clients' PINs. Jarecki *et al.* [16] proposed a protocol to ensure that even if the server or device is corrupted a client's PIN cannot be extracted. But, it imposes high costs due to the use of public-key cryptography and numerous rounds of communication. This protocol requires the client (in addition to remembering its PIN) to locally store a cryptographic secret key.

Solutions in the second category use a mobile phone as a hardware token to generate OTP. They often rely on the added features that mobile phones offer, such as possessing a Trusted Execution Environment (TEE) or being able to communicate directly with the server. The mobile phone-based scheme in [20] uses a combination of time-based OTP and a hash chain. It ensures that even if the adversary corrupts the server, it cannot extract the client's secret. Nevertheless, it requires: (a) the client to store a long secret key (on the mobile phone), (b) the laptop/PC that the client uses to be equipped with a camera, and (c) the mobile phone to invoke a hash function over a million times that can cause the phone's battery to run out fast. The protocol proposed in [21] relies on a phone's TEE and messages that the server can directly send to the phone. Later, Imran *et al.* [1] proposes a protocol that also relies on a phone's TEE, but it improves the protocol presented in [21], in that it is compatible with more android devices and supports biometric authentication. A primary limitation of mobile phone-based OTP tokens is that they cannot be used when there is no (phone) network coverage. Moreover, in certain cases sharing phone numbers with the server may not suit all clients, e.g., transactions' details along with the phone number might be sold for targeted advertisements.

3 Notations and Preliminaries

3.1 Notations

To disambiguate the different uses of keys and other items of data, variables are annotated with a superscript to indicate their origin. $\cdot^C$ indicates data stored at the client, $\cdot^S$ means data stored at the server, and $\cdot^M$ indicates data item has been extracted from a message. Table 1 presents a summary of variables.

Table 1: Notation table.

Symbol	Purpose	Source and lifetime
$\mathsf{PRF}_k(.)$	Pseudorandom function.	Used to derive a verifier and session key.
FS-PRG	Forward-secure Pseudorandom Bit Generator.	Used to derive temporary keys.
k^C, k^S	Authenticated Encryption (AE) key at the client and server sides respectively.	Key k randomly generated by the system operator and stored by the client as k^C and server as k^S at device creation. Constant for the lifetime of the device.
st^C, st^S	The state of FS-PRG at the client and server sides respectively.	Initialised to randomly generated state st_0 at device creation. Updated using FS-PRG.
kt_1^C, kt_1^S	Temporary key for the enrolment phase.	Output by FS-PRG and used for a single message exchange before being discarded.
kt_2^C, kt_2^S, kt_3^C, kt_3^S	Temporary keys of PRF, used in the authentication phase.	Output by FS-PRG and used for a single message exchange before being discarded.
ct^C, ct^S	Counter for synchronising FS-PRG state and detecting replayed messages.	Initialised to zero at device creation. ct^C and ct^S are updated atomically along with st^C and st^S respectively.
N^S, N^M	Random challenge for detecting replayed messages.	Generated randomly by the server for each message.
sa^C	Random PIN-obfuscation secret key.	Initialised to randomly generated value at device creation. Not known by the server or system operator.
PIN^C	Client's PIN.	Entered by the client. It is never stored in the device and used to generate a verifier.
v^C, v^S, v^M	Verifier, generated from PIN-obfuscation key and the client's PIN.	Stored by the server after the enrolment phase. It is not stored by the client.
t^S, t^M	Description of a transaction to be authenticated.	Generated and sent by the server.
$response^C$	Authentication response.	Computed by the client.
$expected^S$	Expected authentication response.	Computed by the server.

3.2 Pseudorandom Function

Informally, a pseudorandom function (PRF) is a deterministic function that takes as input a key and some argument. It outputs a value indistinguishable from that of a truly random function with the same domain and range. A PRF is formally defined as follows [18].

Definition 1. *Let* $\mathsf{PRF} : \{0,1\}^\psi \times \{0,1\}^\eta \to \{0,1\}^\lambda$ *be an efficient keyed function. It is said* PRF *is a pseudorandom function if for all probabilistic polynomial-time distinguishers* B*, there is a negligible function,* $\mu(.)$*, such that:*

$$\left| \Pr[B^{\mathsf{PRF}_{\hat{k}}(.)}(1^\psi) = 1] - \Pr[B^{\omega(.)}(1^\psi) = 1] \right| \leq \mu(\psi)$$

where the key, $\hat{k} \xleftarrow{\$} \{0,1\}^\psi$*, is chosen uniformly at random and* ω *is chosen uniformly at random from the set of functions mapping* η*-bit strings to* ι*-bit strings. We define* $Adv^{\mathsf{PRF}}(\mathcal{A})$ *as the advantage of the adversary which interacts with pseudorandom and random functions.*

Since a pseudorandom function is deterministic and outputs the same value if queried twice on the same inputs, when proving a protocol that uses a PRF, it is assumed that the distinguisher never queries oracles PRF and ω twice on the same inputs [18].

3.3 Forward-Secure Pseudorandom Bit Generator

A Forward-Secure Pseudorandom Bit Generator (FS-RPG), is a *stateful* object which consists a pair of algorithms and a pair of positive integers, i.e., $\mathsf{FS\text{-}RPG} = \Big((\mathsf{FS\text{-}RPG.KGen},\ \mathsf{FS\text{-}RPG.next}), (b, n)\Big)$, as defined in [4]. The probabilistic key generation algorithm FS-RPG.KGen takes a security parameter as input and outputs an initial state st_0 of length s bits. FS-RPG.next is a key-updating algorithm which, given the current state st_{i-1}, outputs a pair of a b-bit block out_i and the next state st_i. We can produce a sequence $out_1, ..., out_n$ of b-bit output blocks, by first generating a key $st_0 \xleftarrow{\$} \mathsf{FS\text{-}RPG.KGen}(1^\lambda)$ and then running $(out_i, st_i) \leftarrow \mathsf{FS\text{-}RPG.next}(st_{i-1})$ for all $i, 1 \leq i \leq n$. As with a standard pseudorandom bit generator, output blocks of this generator should be computationally indistinguishable from a random bit string of the same length. The additional property required from a FS-RPG is that even when the adversary learns the state, output blocks generated before the point of compromise remain computationally indistinguishable from random bits. This requirement implies that it is computationally infeasible to recover a previous state from the current state. We restate a formal definition and construction of a forward-secure pseudorandom bit generator in Appendix B.

Recall, FS-RPG.next updates the state of the forward-secure generation by one step; however, our protocol sometimes needs to invoke FS-RPG.next multiple times sequentially. Thus, for the sake of simplicity, we define a wrapper algorithm $\mathsf{Update}(st_a, d)$ which wraps FS-RPG.next. Algorithm Update as input takes a current state (similar to FS-RPG.next) and new parameter d that determines how many times FS-RPG.next must be invoked internally. It invokes FS-RPG.next d times and outputs the pair (out_b, st_b) which are the output of FS-RPG.next when it is invoked for d-th time, where $b > a$.

3.4 Authenticated Encryption (AE)

Informally, authenticated encryption $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is an encryption scheme that simultaneously ensures the secrecy and integrity of a massage. It can be built via symmetric or asymmetric key encryptions. In this work, we use authenticated symmetric-key encryption, due to its efficiency. Gen is a probabilistic key generating algorithm that takes a security parameter and returns an encryption key k. Enc is a deterministic encryption algorithm that takes the secret key k and a message m, it returns a ciphertext M along with the corresponding tag t. Dec is a deterministic algorithm that takes the ciphertext M, the tag t, and the secret key k. It first checks the tag's validity, if it accepts the tag, then it decrypts the message and returns $(m, 1)$. Otherwise, it returns $(., 0)$.

The security of such encryption consists of the notion of secrecy and integrity. The secrecy notion requires that the encryption be secure against chosen-ciphertext attacks, i.e., CCA-secure. The notion of integrity considers existential

unforgeability under an adaptive chosen message attack. We refer readers to [18] for a formal definition of authenticated symmetric-key encryption.

4 Threat Model

In this section, we present the threat model that this work considers. A two-factor authentication scheme involves two players, Client (C): an honest party which tries to prove its identity to a server by using a combination of a PIN and a device and Server (S): a semi-honest adversary which follows the protocol's instructions and tries to learn C's PIN. It also tries to authenticate itself to C.

We let the server communicate with the device through the client. Specifically, similar to previous works (e.g., those in [16,24,30]), we assume the device has a camera that lets the device scan the (QR code) messages the server sends to it via the client. Each of the above parties may have several instances running concurrently. In this work, we denote instances of client and server by C^i and S^j respectively. Each instance is called an oracle. To formally capture the capabilities of an adversary, $\mathcal{A}$, in a hardware token-based 2FA, we mainly use the (adjusted) model proposed by Bellare *et al.* [3]. In this model, the adversary's capabilities are cast via various queries that it sends to different oracles, i.e., instances of the honest parties; the client and server interact with each other for some fixed number of flows, until both instances have terminated. By that time, each instance should have accepted holding a particular session key (sk), session id (SID), and partner id (PID). At any point in time, an oracle may *"accept"*. When an oracle accepts, it holds sk, SID, and PID. A client instance and a server instance can accept at most once. The above model was initially proposed for password-based key exchange schemes in which the adversary does not corrupt either player. Later, Wang *et al.* [33] added a few more queries to the model of Bellare *et al.* to make it suitable for two-factor authentication schemes. The added queries would allow an adversary to learn either of the client's factors (i.e., either PIN or secret parameters stored in the hardware token) or the server's secret parameters. Below, we restate the related queries.

- `Execute`(C^i, S^j): this query captures **passive** attacks in which the adversary, $\mathcal{A}$, has access to the messages exchanged between C^i and S^j during the correct executions of a 2FA protocol, π.
- `Reveal`(I): this query models the misuse of the session key sk by instance I. Adversary $\mathcal{A}$ can use this query if I holds a session key; in this case, upon receiving this query, sk is given to $\mathcal{A}$.
- `Test`(I): this query models the semantic security of the session key. It is sent at most once by $\mathcal{A}$ if the attacked instance I is "fresh" (i.e., in the current protocol execution I has accepted and neither it nor the other instance with the same SID was asked for a Reveal query). This query is answered as follows. Upon receiving the query, a coin b is flipped. If $b = 1$, then session key sk is given to $\mathcal{A}$; otherwise (if $b = 0$), a random value is given to $\mathcal{A}$.
- `Send`(I, m): this query models **active** attacks where $\mathcal{A}$ sends a message, m, to instance I which follows π's instruction, generates a response, and sends the response back to $\mathcal{A}$. Query `Send`(C^i, start) initialises π; when it is sent, $\mathcal{A}$ would receive the message that the client would send to the server.

- `Corrupt`(I, a): this query models the adversary's capability to corrupt the involved parties.
  - if $I = C$: it can learn (only) one of the factors of C. Specifically,
    - if $a = 1$, it outputs C's PIN.
    - if $a = 2$, it outputs all parameters stored in the hardware token.
  - if $I = S$, it outputs all parameters stored in S.

Authenticated Key Exchange (AKE) Security. Security notions (i.e., session key's semantic security and authentication) are defined with regard to the executing of protocol π, in the presence of $\mathcal{A}$. To this end, a game $Game^{ake}(\mathcal{A}, \pi)$ is initialized by drawing a PIN from the PIN's universe, providing coin tosses to $\mathcal{A}$ as well as to the oracles, and then running the adversary by letting it ask a polynomial number of queries defined above. At the end of the game, $\mathcal{A}$ outputs its guess b' for bit b involved in the Test-query.
Semantic security. It requires that the privacy of a session key be preserved in the presence of $\mathcal{A}$, which has access to the above queries. We say that $\mathcal{A}$ wins if it manages to correctly guess bit b in the Test-query, i.e., manages to output $b' = b$. We denote its advantage as the probability that $\mathcal{A}$ can correctly guess the value of b; specifically, such an advantage is defined as $Adv_{\pi}^{ss}(\mathcal{A}) = 2Pr[b = b'] - 1$, where the probability space is over all the random coins of the adversary and all the oracles. Protocol π is said to be semantically secure if $\mathcal{A}$'s advantage is negligible in the security parameter, i.e., $Adv_{\pi}^{ss}(\mathcal{A}) \leq \mu(\lambda)$.
Authentication. It requires that $\mathcal{A}$ must not be able: (a) to impersonate C, even if it has access to the traffic between the two parties as well as having access to either C's PIN, or its authentication device, or (b) to impersonate S, although it has access to the traffic between the two parties. We say that $\mathcal{A}$ violates mutual authentication if some oracle accepts a session key and terminates, but has no partner oracle, which shares the same key. Protocol π is said to achieve mutual authentication if for any adversary $\mathcal{A}$ interacting with the parties, there exists a negligible function $\mu(.)$ such that for any security parameter λ the advantage of $\mathcal{A}$ (i.e., the probability of successfully impersonating a party) is negligible in the security parameter, i.e., $Adv_{\pi}^{aut}(\mathcal{A}) \leq \mu(\lambda)$.

In certain schemes (including ours), during the key agreement and authentication phase, the client needs to also verify a message, e.g., a bank transaction. To allow such verification to be carried out deterministically, which will be particularly useful in the schemes proof, we define a predicate $y \leftarrow \phi(m, \pi)$, where $y \in \{0, 1\}$. This predicate takes as input a message m (e.g., bank's transactions) and a policy π (e.g., a client's policy specifying a payment amount and destination account number). It checks if the message matches the policy. If they match, it outputs 1; otherwise, it outputs 0.

5 The Protocol

In this section, we present an efficient 2FA protocol that remains secure even if an adversary (a) observes the traffic between a client and server, and (b) has access to the client's device, or its PIN, or breaches the server. To design a protocol that can offer the above features, we rely on a novel combination of the following

two approaches. First, we require the client's PIN verification to take place only on the server. This allows us to separate the location where the PIN's secret key (used to generate the PIN's authenticator) is stored from the location where the authenticator is stored. This approach ensures that even if either location is breached, then the adversary would not have sufficient information to retrieve the PIN even through brute-forcing all possible PINs. Our observation is that even in this setting, the device can perform a basic check to detect the client's mistake without having to permanently store the (representation of the) PIN; for instance, this can be done by asking the client to type in the PIN twice and checking if the two entries match with each other.

Second, we (a) require every sensitive message exchanged between the server and client to be encrypted using key-evolving symmetric-key encryption (i.e., a combination of forward-secure pseudorandom bit generator and authenticated encryption) and (b) require the used keys, of key-evolving symmetric-key encryption, to be discarded right after their use. This approach ensures the secrecy of the communication between the parties and ensures that if the device or server is broken in, the adversary cannot learn the past communication to learn the PIN, with the assistance of the information it extracts from the breached location.

Our protocol consists of three main phases; namely, (i) a setup phase, performed once when the authentication device is manufactured, (ii) an enrolment phase for setting or changing a client's PIN, and (iii) an authentication phase in which the actual authentication is performed. As we already stated, each party has a unique (public) ID. In the protocol, we assume the parties include their IDs in their outgoing messages. Similar to other (two-factor) authentication schemes, we assume the server maintains a local threshold, and if the number of incorrect responses from a client within a fixed time exceeds the threshold, then the client and its device will be locked out. Such a check is implicit in the protocol's description. In the remaining of this section, we describe each phase.

5.1 Setup Phase

To bootstrap the protocol, in the setup phase, we require that the client and server share *initial* randomly generated key k for AE and key st_0 for FS-PRG. The counter for the FS-PRG state is set to 0 on both sides. These values could be securely loaded into the device at the time of manufacture or can be sent (via a secure channel) to the client who can use the device camera to scan and store them in the device. In this phase, the device generates and locally stores a random secret key sa^C for PRF. Figure 1 presents the setup in detail.

5.2 Enrolment Phase

The goal of the enrolment phase is to set the client's PIN, without providing the server with sufficient information to discover this PIN. At the end of this phase, the server will have stored the verifier v corresponding to the client's selected PIN. The steps involved in this phase are shown in Figure 2 in detail.

We briefly explain how this phase works. The server first updates the FS-PRG's state, which results in a new state and random value kt_1^S; it also increments its counter by one. Then, the server generates a random challenge N^S.

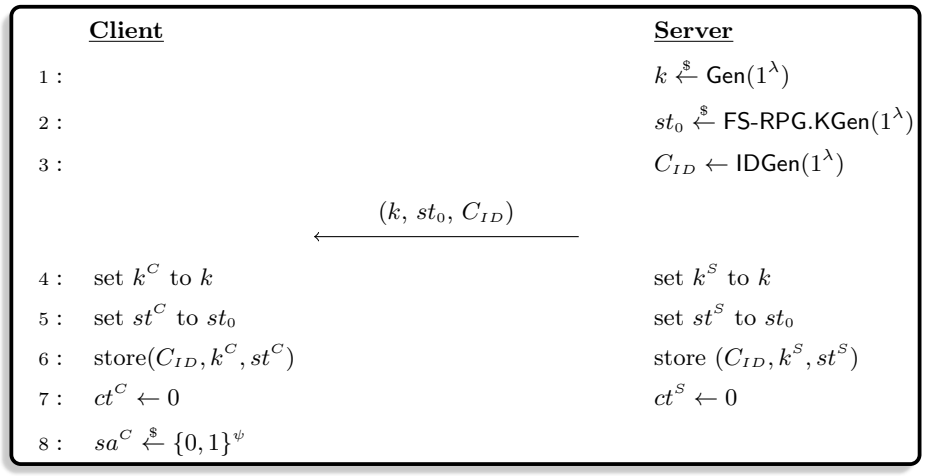

Fig. 1: Setup phase.

The server sends the enrolment challenge message which is a combination of the current counter and the challenge encrypted via the AE under the shared key k. On receiving this message, the client uses its device to scan the (QR) message it receives. The device decrypts the message using k that was shared with the server during the setup phase. If decryption succeeds, it extracts the server's challenge and counter from the message. If the device's local counter is greater than the counter it received from the server, it would be impossible for the device to recover the kt_1^S that the server will use, so the protocol must abort here. As we will discuss in Appendix C, this case would not occur, with a high probability. Next, the device requests the PIN from the client and ensures it is what the client intends, e.g., by requesting it twice and checking they match.

The device then generates a verifier v^C, by deriving a pseudorandom value from the PIN using PRF and the random key sa^C it generated in the setup phase. After that, the device locally synchronises the FS-PRG's state with the server by updating the state until it matches the counter received from the server; this yields kt_1^C. This synchronisation is possible because the check at line 9 has already assured that the device's state is behind the server's state by at least one step. After the update, kt_1^C will equal kt_1^S because the initial FS-PRG's state is the same (from the setup phase) and the two generators have been updated the same number of times. The client then encrypts the verifier and challenge under kt_1^C and sends this to the server. On receiving and validating this message, the server decrypts the message using $kt_1{}^S$, then extracts the challenge and verifier. If the challenge does not match the one corresponding to the current protocol exchange, the protocol halts. If the challenge does match, the server stores the verifier, v^S, associated with the client's account.

Finally, the device discards the challenge, kt_1^C, PIN, and v^C so that the PIN can no longer be recovered from the device. Note that the device can re-generate

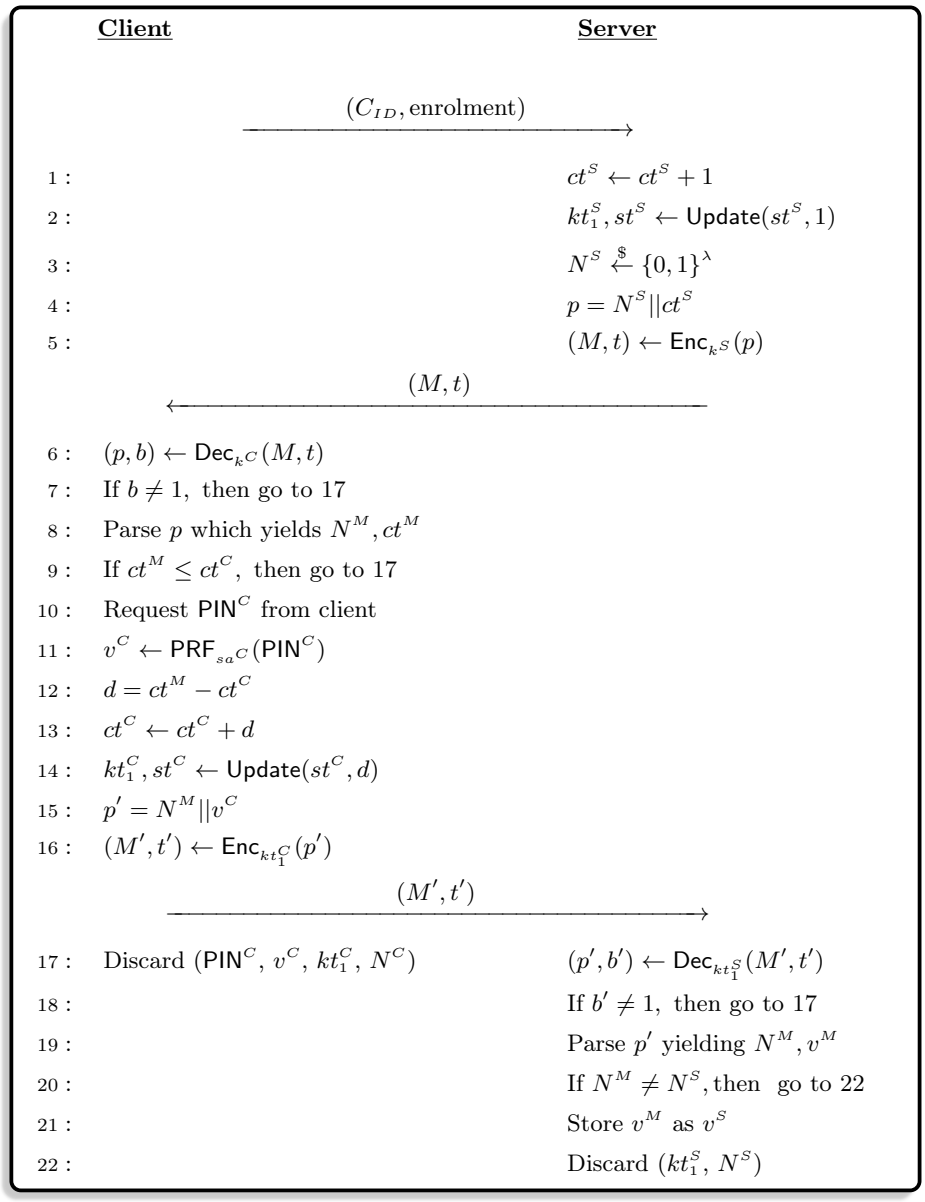

Fig. 2: Enrolment phase.

v^C using sa^C when the client types in its PIN again. The server also discards the challenge and $kt_1{}^C$ as they are no longer needed. Following the successful completion of this protocol, the server will store the verifier corresponding to

the client's selected PIN and both server and device will have synchronised their FS-PRG's state.

5.3 Authentication Phase

The goal of the authentication process is to give the server assurance that the device is currently present, the correct PIN has been entered, and the client has been shown the transaction that the server wishes to execute.

This phase works as follows. The server first updates the FS-PRG's state and corresponding counter, which results in a new state st^S, a new random value kt_2^S, and a new temporary counter tmp_{ct^S}. The server updates the state and the counter one more time which yields a new state st^S, a new random value kt_3^S, and a new counter ct^S. The server generates a random challenge and two ciphertexts, $\ddot{M}$ and $\hat{M}$. The former ciphertext consists of the random challenge and the description of the transaction, encrypted under key kt_2^S. The latter ciphertext contains the counter tmp_{ct^S}, encrypted under key k^C. The reason tmp_{ct^S} is encrypted under key k^C is to allow the device to decrypt the ciphertext easily in case of previous message loss; for instance, when the server sends $(\ddot{M}, \hat{M})$ to the server, but they are lost in transit, multiple times, and a fresh pair finally arrives the client after the server sends them upon the client's request. Encrypting tmp_{ct^S} under key k^C (instead of one of the evolving keys) lets the device deal with such a situation.

Upon receiving the ciphertexts, the device validates and decrypts the messages. It extracts the challenge N^M, counter tmp_{ct^S}, and transaction t^M. It ensures that its own counter is behind the received counter. As will be discussed in Section C, this check should always succeed. The device synchronises its state and counter using the server's messages. Next, the device displays the transaction for the client to check. If the client does not accept the transaction (e.g., due to an attempted man-in-the-browser attack), then the protocol aborts immediately. Assuming the client is willing to proceed, then the device prompts for the PIN, and computes the verifier v^C using the key sa^C. If the client enters the correct PIN, the verifier will be the same as the one sent to the server during the enrolment phase.

For the device to generate the response message, first it updates its state one more time, which results in a pseudorandom value kt_3^C. Then, it derives a pseudorandom value, $response^C$, from a combination of the random challenge N^M, transaction t^M, verifier v^C, and $x = 1$ using PRF and kt_3^C. The device generates a session key, using the above combination and key with a difference that now $x = 2$. The response message is sent to the server. The device discards the PIN, the verifier, all FS-PRG keys, the challenge, and the transaction's description, so as to protect the PIN from discovery. The server computes the expected response message based on its own values of the challenge, transaction, and verifier. Note that the verifier is retrieved from the value set during the enrolment phase. The server then compares the expected response with the response sent by the client. Only if these match, the authentication is considered to have succeeded. If the response does not match the one the server expects this could indicate that the message was tampered with, or that the client entered an incorrect PIN. Next,

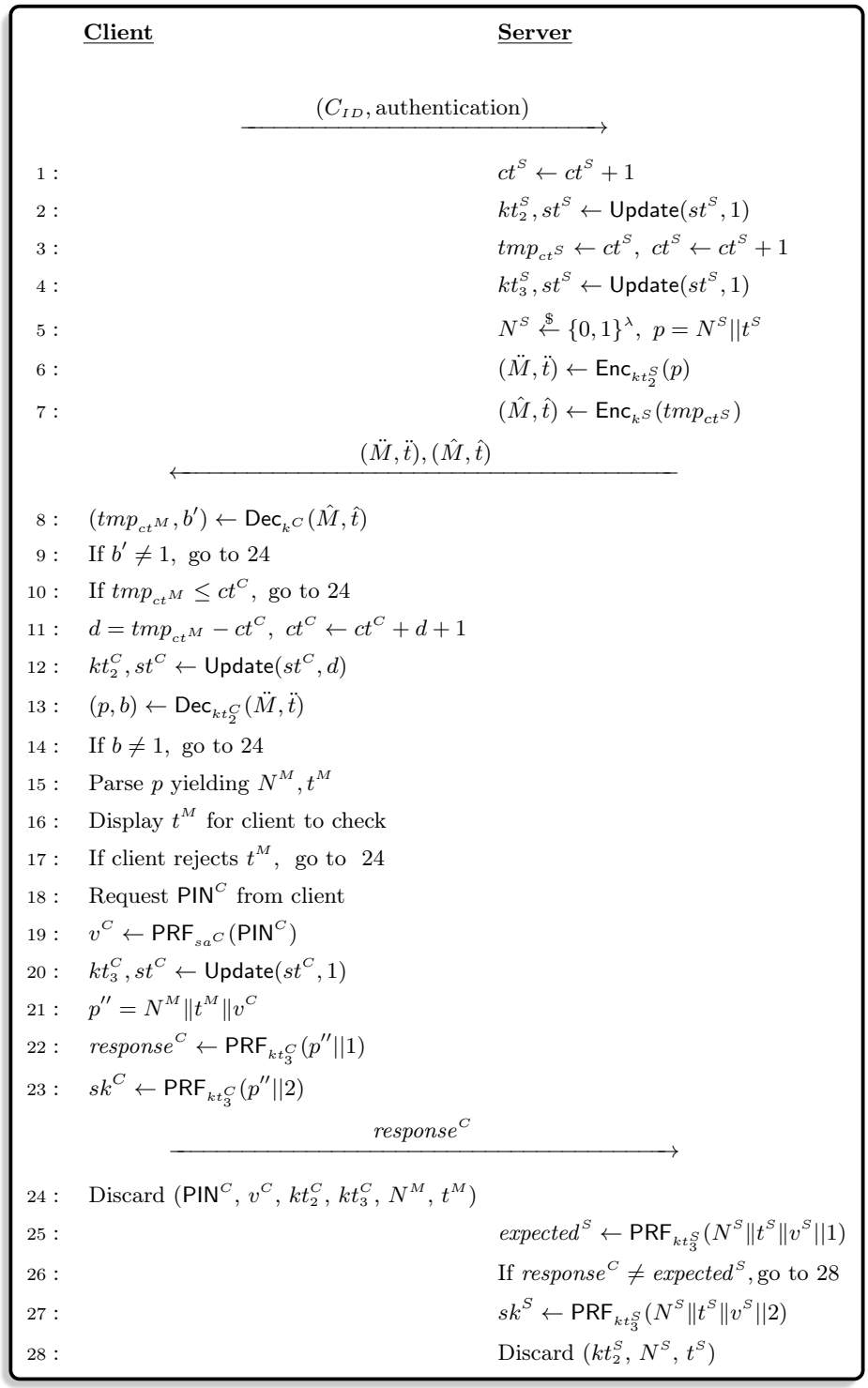

Fig. 3: Authentication phase.

the server generates the session key the same way as the device does. The server also discards the FS-PRG key, the challenge, and the transaction's description.

In Appendix D, we present a couple of straw-man solutions (that one might be tempted to use to achieve the same goals that our protocol attains) and explain why they would not work. Below, we formally state the security of our protocol. First, we present a theorem stating that the advantage of an adversary in breaking the semantic security of the above protocol is negligible.

Theorem 1 (Semantic Security). *Let $\mathcal{A}$ be a probabilistic polynomial time (PPT) adversary with less than q_s interactions with the parties and q_p passive eavesdropping, i.e., number of local executions. Let λ be a security parameter and $Adv_{\pi}^{ss}(\mathcal{A})$ be $\mathcal{A}$'s advantage (in breaking the semantic security of an AKE scheme π) as defined in Section 4. Then, such an advantage for the protocol ψ has the following upper bound:*

$$Adv_{\psi}^{ss}(\mathcal{A}) \leq 2(q_s + q_p)\Big(Adv^{\mathsf{PRF}}(\mathcal{A}) + Adv^{Enc}(\mathcal{A})\Big) + \frac{8(2q_s + q_p)}{2^{\lambda}}$$

Next, we present a theorem stating that the advantage of an adversary in breaking the authentication of the above protocol is negligible.

Theorem 2 (Authentication). *Let PIN be an element distributed uniformly at random over a finite dictionary of size N. Also, let $\mathcal{A}$ be a PPT adversary with less than q_s interactions with the parties and q_p passive eavesdroppings. Let λ be a security parameter and $Adv_{\pi}^{aut}(\mathcal{A})$ be $\mathcal{A}$'s advantage (in breaking the authentication of an AKE scheme π) as defined in Section 4. Then, in the protocol ψ, $Adv_{\psi}^{aut}(\mathcal{A})$ has the following upper bound:*

$$Adv_{\psi}^{aut}(\mathcal{A}) \leq (q_s + q_p)\Big(Adv^{\mathsf{PRF}}(\mathcal{A}) + Adv^{Enc}(\mathcal{A})\Big) + \frac{9q_s + 4q_p}{2^{\lambda}} + \frac{q_s}{N}$$

We refer to Appendix E for informal security analysis and Appendix F for formal security proof where the above two theorems will be proven.

6 System Usability

Usability is of critical importance for an effective authentication system as otherwise, clients will refuse to use it or implement insecure workarounds [11]. As we highlighted in Section 4, in our protocol, the server interacts with the device via the client. To accommodate usability and let the device easily receive the server's message, we require the device to be equipped with a scanner/camera, such as the one shown in Figure 4.

In our protocol, the encrypted messages the server sends are in a form of a QR code, so the client can easily scan them from its computer's screen and transfer them to the device which verifies and decrypts the input messages, as explained in the protocol. However, to avoid limiting the protocol's application, we do not require the client's computer to be equipped with a scanner/camera. Therefore, the client needs to manually insert the message output of the device into its computer. Once again, to improve usability, it is desirable to reduce the size of each message that the device outputs. Our protocol allows truncating the

Fig. 4: OneSpan Digipass 770 authentication device.

verifier, as it is resistant to offline brute-force attacks, with a caveat. Truncating the verifier will create collisions such that for some values of the verifier there will be multiple PINs which are valid. Consequently, an adversary who has stolen a device has a better chance of guessing the PIN than the ideal case where the original message is used and its chance is negligible in the security parameter or the PIN's universe size. This security-usability trade-off is not specific to our protocol and exists in all hardware token-based multi-factor authentication protocols that do not assume clients' computers are equipped with a scanner and clients have to insert the device's messages into their computer.

Another consideration is handling mistyped or forgotten PINs. As we highlighted in Section 5, the device can ask the client to confirm its PIN by entering the PIN twice; then the device compares the two entries and alters the client if they do not match. However, it does not check and alert the client whether the PIN is correct. Such a check has to be carried out on the server-side which comes at a cost of decreased usability. The protocol presented in this work is useful in its own right; however, it could be considered a basis for a more user-friendly system, tailored to a particular scenario in which it is to be used.

7 Evaluation

In this section, we briefly analyse and compare our 2FA protocol with the smart-card-based protocol proposed in [33] and the hardware token-based protocol in [16] as the latter two protocols are relatively efficient, do not use secure chipsets, and they consider the same security threats as we do. We summarise the analysis result in Table 2. We refer readers to Appendix G for a more detailed evaluation.

7.1 Computation Cost

In our protocol, each party (client or server) invokes the authenticated encryption scheme 4 times and the pseudorandom function 5 times. In the protocol proposed in [33], the client invokes a hash function 11 times and performs 3 modular exponentiations while the server invokes the hash function 8 times and performs 2 modular exponentiations. Moreover, in the protocol presented in [16], the client invokes a hash function 3 times and runs symmetric key encryption once. It also

Table 2: Comparison of efficient two-factor authentication protocols.

Features	Operation	Our Protocol	[33]	[16]
Computation cost	Sym-key	18	19	7
	Modular expo.	0	5	12
Communication cost	—	2804-bit	3136-bit	3900-bit
Not requiring multiple pass/PIN	—	✓	×	×
Not requiring modular expo.	—	✓	×	×
Security assumption	—	Standard	Random oracle	Random oracle

performs 10 modular exponentiations. While the server invokes a pseudorandom function once and performs at least 2 modular exponentiations and 2 symmetric-key encryptions. Thus, our protocol and the ones in [33,16] involve a constant number of symmetric key primitive invocations; however, our protocol does not involve any modular exponentiations, whereas those in [33,16] involve a constant number of modular exponentiations which leads to a higher cost.

7.2 Communication Cost

In our protocol, the communication cost of the client is 1268 bits and the server is 1536 bits. However, in the protocol proposed in [33], the communication cost of the client is 1952 bits and the server is 1184 bits; while in the protocol developed in [16] the client's and server's communication costs are at least 2856 and 1044 bits respectively. Hence, our protocol imposes 10% and 40% lower communication costs than the protocols in [33] and [16] do respectively.

7.3 Other Features

In our protocol, a client needs to know only a single secret, i.e., a PIN. Nevertheless, in the protocol in [33], a client has to know an additional secret, i.e., a random ID. As shown in [28], the scheme in [33] will not remain secure, even if only the client's ID is revealed. The protocol in [16] requires the client (in addition to remembering its PIN) to locally store a cryptographic secret key of sufficient length, e.g., 128 bits; this secret key must not be kept on the device. Furthermore, our protocol is secure in the standard model while the protocols in [33,16] are in the non-standard random oracle model.

8 Conclusion and Future Work

In this paper, we have presented an efficient 2FA protocol that resists a strong adversary who may (a) observe the traffic between a client and server, and (b) have physical access to the client's device, or its PIN, or breach the server. Our protocol offers a unique combination of key features that state-of-the-art schemes do not. Specifically, our protocol (i) requires a client to remember only one secret/PIN, (ii) is based on only symmetric key primitives, (iii) is in a standard model, and (iv) imposes low communication costs. This is the first protocol that offers the aforementioned features without using any trusted chipsets. Future research could investigate the usability of a hardware token that embeds our 2FA protocol.

References

1. SARA: Secure android remote authorization. In: USENIX Security (2022)
2. RSA Security LLC: Rsa securid hardware authenticators, technical specifications (2021), https://www.securid.com/wp-content/uploads/2021/11/rsa-securid-hardware-tokens-technical-specifications-012621.pdf
3. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: EUROCRYPT (2000)
4. Bellare, M., Yee, B.S.: Forward-security in private-key cryptography. In: CT-RSA (2003)
5. Biryukov, A., Lano, J., Preneel, B.: Cryptanalysis of the alleged securid hash function. In: International Workshop on Selected Areas in Cryptography (2003)
6. Bonneau, J., Preibusch, S.: The password thicket: Technical and market failures in human authentication on the web. In: WEIS. Citeseer (2010)
7. Bresson, E., Chevassut, O., Pointcheval, D.: Security proofs for an efficient password-based key exchange. In: CCS (2003)
8. Chang, C.C., Wu, T.C.: Remote password authentication with smart cards. IEE Computers and Digital Techniques (1991)
9. Chang, D., Mishra, S., Sanadhya, S.K., Singh, A.P.: On making U2F protocol leakage-resilient via re-keying. IACR Cryptol. ePrint Arch. p. 721 (2017)
10. Chaturvedi, A., Das, A.K., Mishra, D., Mukhopadhyay, S.: Design of a secure smart card-based multi-server authentication scheme. J. Inf. Secur. Appl. (2016)
11. De Cristofaro, E., Du, H., Freudiger, J., Norcie, G.: A comparative usability study of two-factor authentication. arXiv preprint arXiv:1309.5344 (2013)
12. Feng, H., Li, H., Pan, X., Zhao, Z.: A formal analysis of the FIDO UAF protocol. In: NDSS. The Internet Society (2021)
13. Gentry, C., MacKenzie, P.D., Ramzan, Z.: A method for making password-based key exchange resilient to server compromise. In: CRYPTO (2006)
14. Gupta, B., Prajapati, V., Nedjah, N., Vijayakumar, P., El-Latif, A.A.A., Chang, X.: Machine learning and smart card based two-factor authentication scheme for preserving anonymity in telecare medical information system (tmis). Neural Computing and Applications (2021)
15. Jacomme, C., Kremer, S.: An extensive formal analysis of multi-factor authentication protocols. ACM Trans. Priv. Secur. (2021)
16. Jarecki, S., Jubur, M., Krawczyk, H., Saxena, N., Shirvanian, M.: Two-factor password-authenticated key exchange with end-to-end security. ACM Trans. Priv. Secur. (2021)
17. Juels, A., Triandopoulos, N., Van Dijk, M., Brainard, J., Rivest, R., Bowers, K.: Configurable one-time authentication tokens with improved resilience to attacks (Feb 23 2016), uS Patent 9,270,655
18. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. CRC Press (2014)
19. Kim, S.K., Chung, M.G.: More secure remote user authentication scheme. Computer Communications (2009)
20. Kogan, D., Manohar, N., Boneh, D.: T/key: Second-factor authentication from secure hash chains. In: CCS. ACM (2017)
21. Konoth, R.K., Fischer, B., Fokkink, W.J., Athanasopoulos, E., Razavi, K., Bos, H.: Securepay: Strengthening two-factor authentication for arbitrary transactions. In: EuroS&P (2020)
22. Loutfi, I., Jøsang, A.: FIDO trust requirements. In: NordSec 2015 (2015)

23. of the President-Office of Management, E.O., Budget: Moving the u.s. government toward zero trust cybersecurity principles (2022), https://www.whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf
24. OneSpan: Digipass 770 datasheet, https://www.onespan.com/resources/digipass-770/datasheet#:~:text=Digipass%20770%20is%20a%20non,capturing%20the%20Cronto%20activation%20code.
25. OneSpan: Digipass 770 (2018), https://www.onespan.com/sites/default/files/2019-08/Digipass-770_datasheet.pdf
26. Panos, C., Malliaros, S., Ntantogian, C., Panou, A., Xenakis, C.: A security evaluation of fido's UAF protocol in mobile and embedded devices. In: TIWDC (2017)
27. Radhakrishnan, N., Muniyandi, A.P.: Dependable and provable secure two-factor mutual authentication scheme using ecc for iot-based telecare medical information system. Journal of Healthcare Engineering (2022)
28. Scott, M.: Cryptanalysis of a recent two factor authentication scheme. IACR Cryptol. ePrint Arch. p. 527 (2012)
29. Sinigaglia, F., Carbone, R., Costa, G., Zannone, N.: A survey on multi-factor authentication for online banking in the wild. Comput. Secur. (2020)
30. Thales: Gemalto dynamic signing token a user-friendly token that enables strong authentication and transaction data signing (2020), https://www.thalesgroup.com/sites/default/files/database/document/2020-12/fs-dynamic-signing-token.pdf, visited on 2022.05.28
31. Tian, Y., Li, Q., Hu, J., Lin, H.: Secure limitation analysis of public-key cryptography for smart card settings. World Wide Web (2020)
32. Wang, D., Gu, Q., Cheng, H., Wang, P.: The request for better measurement: A comparative evaluation of two-factor authentication schemes. In: AsiaCCS (2016)
33. Wang, D., Wang, P.: Two birds with one stone: Two-factor authentication with security beyond conventional bound. IEEE Trans. Dependable Secur. Comput. (2018)
34. Zhang, Z., Wang, Y., Yang, K.: Strong authentication without temper-resistant hardware and application to federated identities. In: NDSS (2020)

A Survey of Related Work

In this section, first, we briefly discuss the common approaches for generating a One-Time Password (OTP) which yields from a combination of a PIN and a hardware token. After that, we provide an overview of hardware token variants.

A.1 Common Approaches for Generating OTP

In the authentication that relies on a combination of knowledge and possession factors, once the client enters the secret into the hardware token, the device (in some cases after validating the secret) combines this secret with the output of one of the following methods to generate a unique OTP: (i) a random challenge: this approach requires the server to send a random challenge to the device (through the client); those protocols that use this approach needs to ensure the random challenges themselves remain confidential in the presence of an eavesdropping adversary, (ii) an internal counter: the solutions that use this approach needs to take into account the situation where the token-side counter becomes out of

synchronisation, or (iii) the current accurate time: this approach requires the authentication server and token use a synchronised clock and the two endpoints may get out of synchronisation after a certain time. There exist 2FA solutions (including the one we propose in this paper) that employ a combination of the above approaches.

A.2 Variants of OTP Hardware Tokens

Connected Tokens. This type of token requires a client to physically connect the token to their computer (e.g., a laptop or card reader) via which the client is authenticating. Once it is connected, the device transmits the authentication information to the computer (either automatically or after pressing a button on the token). USB tokens and smart cards are two popular token technologies in this category. Various companies including Google, Dropbox, and "Fast IDentity Online" (FIDO) Alliance have developed USB hardware tokens. YubiKey[1] is one of the well-known ones developed by FIDO. The FIDO Alliance has proposed a standard that aims at allowing clients to log in to remote services with a local and trusted authenticator. It supports a wide range of authentication technologies including USB (security) tokens. However, researchers have discovered various vulnerabilities within this standard via manual, e.g., in [26,9,22] and formal analysis, e.g., in [12].

Smart card technology is another authentication means which has been widely used. Often it comprises two separate components; namely, a smart card and a card reader, where the former includes an integrated secure chipset while the former includes a keypad and a screen. Since its introduction in [8], there have been numerous protocols for smart card-based 2FA (e.g., in [14,33,27]) along with a few works that identify vulnerabilities of existing solutions, e.g., in [31,32,10]. However, the existing smart card-based solutions (e.g., in [14,33,27]) are often based on public-key cryptography which imposes a high computation cost and makes the card reader's battery run out relatively fast; also some solutions (e.g., in [19]) rely on tamper-proof secure chipsets embedded in the card which would ultimately increase the device's cost.

Disconnected Tokens. This type of token does not have a physical connection to a client's computer making them more convenient than connected tokens. A disconnected token is often equipped with a built-in screen and a keypad letting a client type in the knowledge factor and view the OTP on the screen (see below for an exception). Below, we provide an overview of two main categories of disconnected tokens.

1. Dedicated hardware-based Tokens, such as RSA SecureID [2], OneSpan Digipass 770 [24], and Thales Gemalto SWYS QR Token Eco [30]. RSA SecureID (unlike the other two tokens) does not have a keypad. Briefly, in RSA SecureID, the OTP is generated using the current time and a secret key (allocated to the client and) stored in the token [5]. Thus, not only the token

[1] https://www.yubico.com

has to have a synchronised clock with the server, but also the token's OTP can be generated by an adversary who has physical access to the device, as it can extract the device's secret key. The main advantage of Digipass 770 and Thales Gemalto SWYS QR Token Eco to RSA SecureID is that they allow clients to see and verify the transaction details through the token. Therefore, the client is given more understandable information about the transaction it is approving, so phishing (by Man-in-the-Browser attacks or social engineering attacks) becomes harder.

Our investigation suggests that Digipass 770 also *locally stores and verifies* clients' PINs. Specifically, once a client receives the token, it also receives an activation code from the verifier, e.g., the client's bank. Then, the client (i) registers the activation code in the device and (ii) registers the activation code to the verifier, so the verifier knows that this specific client has a device with the provided activation code. Then, the client registers its PIN in the device which stores it locally. Every time a client uses the verifier's online system (e.g., online banking) and makes a transaction, the system generates and displays an encrypted visual image. The client uses its token (camera) to scan the image, and then enters its PIN into the device. Next, the device checks the PIN; if the PIN matches the previously registered PIN, then it decrypts the image and displays the transaction's content on the token's screen which allows the client to check whether the transaction is the one it has made. If the client accepts the transaction and presses a certain button, then the token generates and displays an OTP that the client can insert into the verifier's online system [24,25]. Thales Gemalto SWYS QR Token Eco also uses a mechanism similar to the one we described above.

Jules et al. [17] discussed that the adversary who can intercept the client and server's communication and also has physical access to the client's token or the server's storage can extract the client's PIN and impersonate the client. To address the issue they also suggested a solution that can address the above issue by using (i) a forward-secure pseudorandom number generation, (ii) multiple servers, etc. However, the proposed scheme lacks formal proof and does not consider the case where transactions' details must be verified by clients on the token.

Moreover, Jarecki *et al.* [16] proposed a (single server) protocol to ensure that even if the server or the device is corrupted a client's PIN cannot be extracted and the adversary cannot impersonate an honest client. It is mainly based on a hash function, both symmetric and asymmetric key encryptions, and (Diffie–Hellman) key exchange. This scheme has a high computation and communication cost due to its complexity, the use of public-key cryptography, and numerous rounds of communication, even between the client and token. Also, it requires the token to perform asymmetric-key operations and invoke symmetric key primitives many times, which would make the token's battery run out quickly. This protocol requires the client (in addition to remembering its PIN) to remember/store a cryptographic secret key locally (but not on the token), as a result of invoking a subroutine called asymmetric "password-authenticated key exchange" (PAKE). Furthermore, there

is another authentication protocol, that does not rely on a trusted chipset, presented in [34]. Nevertheless, it has been designed for "federated identity systems" and is not suitable for two/multi-factor authentication settings.

2. Mobile phone-based Tokens, such as the solutions presented in [1,20,21]. There have been protocols that generate an OTP with the use of a mobile phone as a hardware token. Such solutions often rely on the added features that mobile phones offer, such as possessing a Trusted Execution Environment (TEE), being able to communicate directly with the server, or having a rechargeable battery. The scheme in [20] relies on a combination of time-based OTP and a hash chain. This scheme ensures that even if the adversary corrupts the server at some point, then it cannot extract the client's secret. Nevertheless, it (a) requires the client to store a sufficiently long secret key (on the mobile phone), (b) requires the laptop/PC that the client uses to be equipped with a camera, and (c) needs the mobile phone to invoke a hash function over a million times that can cause the phone's battery to run out fast. The protocol proposed in [21] mainly relies on a phone's TEE (i.e., ARM TrustZone technology) and messages that the server can directly send to the phone. Later, Imran *et al.* [1] proposes a new protocol that also relies on a phone's TEE, but it improves the protocol presented in [21], in the sense that it is compatible with more android devices and supports biometric authentication too.
   A primary limitation of mobile phone-based OTP tokens is that they cannot be used when there is no (mobile phone) network coverage. Another limitation is that in certain cases (beyond internet banking) sharing phone numbers with the authentication server may not suit all clients, e.g., transactions' details along with the phone number might be sold for targeted advertisements.

B Definition and Construction of Forward-Secure Pseudorandom Bit Generator

In this section, we restate the formal definition of the forward-secure pseudorandom bit generator (taken from [4]), and then briefly explain how it can be constructed. A standard pseudorandom generator is said to be secure if its output is computationally indistinguishable from a random string of the same length. However, the forward security of a stateful generator requires more security guarantees. Specifically, in this setting, an adversary $\mathcal{A}$ may at some point penetrate the machine in which the state is stored and obtain the current state. In this case, the adversary is able to compute the future output of the generator. But, it is required that the bit strings generated in the past still be secure, i.e., the strings are computationally indistinguishable from random bit strings. This implies that it is computationally infeasible for the adversary to recover the previous state from the current one.

In this setting, the adversary is allowed to choose when it wants to penetrate the machine, as a function of the output blocks it has seen so far. Thus, first,

the adversary runs in a "find" stage where it is fed output blocks, one at a time, until it says it wants to break in, and at that time the current state is returned. Next, in the "guess" stage, it must decide if the output blocks that were given to it were the outputs of the generator, or were independent random bits. This is captured formally by two experiments; namely, real and random. In the real experiment, the forward secure generator is used to generate output blocks. Nevertheless, in the ideal experiment, the output blocks are truly random strings (of the same length as that of the blocks in the real experiment). Note that below "$\mathcal{A}(\text{find}, out, h)$" denotes $\mathcal{A}$ in the find stage, and is given an output block out and current history h and returns a pair (I, h) where h is an updated history and $I \in \{\text{find}, \text{guess}\}$. Below, we restate the two experiments.

$\underline{\mathsf{Exp}^{\text{fs-prg}}_{\text{real}}(\mathcal{A}, \text{aux})}$

$st_0 \xleftarrow{\$} \mathsf{FS\text{-}RPG.KGen}(1^\lambda)$
$i \leftarrow 0$
$h \leftarrow \text{aux}$
Repeat
$i \leftarrow i + 1$
$(out_i, st_i) \leftarrow \mathsf{FS\text{-}RPG.next}(st_{i-1})$
$(I, h) \leftarrow \mathcal{A}(\text{find}, out_i, h)$
Until $(I = guess)$ or $(i = n)$
$g \leftarrow \mathcal{A}(\text{guess}, st_i, h)$
Return$g$

$\underline{\mathsf{Exp}^{\text{fs-prg}}_{\text{ideal}}(\mathcal{A}, \text{aux})}$

$st_0 \xleftarrow{\$} \mathsf{FS\text{-}RPG.KGen}(1^\lambda)$
$i \leftarrow 0; h \leftarrow \text{aux}$
Repeat
$i \leftarrow i + 1$
$(out_i, st_i) \leftarrow \mathsf{FS\text{-}RPG.next}(st_{i-1})$
$out_i \xleftarrow{\$} \{0, 1\}$
$(I, h) \leftarrow \mathcal{A}(\text{find}, out_i, h)$
Until $(I = guess)$ or $(i = n)$
$g \leftarrow \mathcal{A}(\text{guess}, st_i, h)$
Return$g$

Given the experiments, the adversary's advantages are defined in the following two equations.

$$\mathtt{Adv}^{\text{fs-prg}}(\mathcal{A}) = Pr[\mathsf{Exp}^{\text{fs-prg}}_{\text{real}}(\mathcal{A}, \text{aux}) = 1] - Pr[\mathsf{Exp}^{\text{fs-prg}}_{\text{ideal}}(\mathcal{A}, \text{aux}) = 1] \quad (1)$$

$$\mathtt{Adv}^{\text{fs-prg}}(t) = Max\{\mathtt{Adv}^{\text{fs-prg}}(\mathcal{A})\} \quad (2)$$

Equation 1 refers to the (fs-prg) advantage of $\mathcal{A}$ in attacking the forward-secure pseudorandom bit generator, FS-PRG. Moreover, Equation 2 refers to the maximum advantage of $\mathcal{A}$ in attacking FS-PRG, where the adversary has a time-complexity at most t. It is required that the adversary's advantage is negligible for practical values of t.

Bellare *et al.* [4] proposed various instantiations of FS-PRG, including the one based on AES. In the latter case, one can set a block size b and a state size s to 128 bits. We refer readers to [4] for further discussion.

C Synchronisation

A user's device needs to be synchronised with the server in order for the server to check the correctness of the response generated by the device. This is particularly the case in our proposed protocol because if one side advances too far, it is by design impossible for it to move backwards. Specifically, we must provide assurance that the server state remains at the same state as the client's state, or that the server is ahead of the client, i.e., $ct^S \geq ct^C$. Then, as challenge messages always contain the current value of the server's counter, the client is always able to catch up with the server. We achieve this via three approaches. Firstly, by requiring the FS-PRG's state to advance with the counter, such that the counter is consistent with the state. Secondly, by requiring that the client never advances its state directly, but only advances to the point that the server currently is at. Thirdly, by requiring the client only to advance its state in response to an authenticated challenge from the server.

The protocol takes into account the case where messages are dropped. Response messages are not involved in advancing the forward-secure state; therefore, if these messages are dropped, then it would not have any effect on synchronisation. However, challenge messages are important, if any of them is dropped, then the device would not advance the state and would be behind the server. Nevertheless, this would not cause any issue, because the server's next challenge message will include the new value of the counter and the device will advance the state until it matches the server's state. Note that the FS-PRG advance process is fast; thus, multiple invocations of this will not create a noticeable delay. Note that in the case where the enrolment's response message is dropped, the PIN will remain unchanged; as a result, the client may be surprised that the new PIN does not work. But, the old PIN will keep working and enrolment can be repeated to update the PIN.

D Straw-man Solutions

In this section, we provide an overview of a couple of solutions that seem to work and discuss their shortcomings.

D.1 Straw-man Proposal I

A simple authentication protocol would be for the server to generate a secret key k, then enrol a client's device by sharing this key over a secure channel, e.g., loaded onto the device at the time of manufacture. Then, when the secure channel is not available (e.g., during Internet banking) the server sends a randomly generated challenge to the client which replies with a Message Authentication Code (MAC) computed of this challenge, under k. This protocol provides the server assurance that the response originated from the correct client and bounds the time at which the response was generated to be between the time that the challenge was sent and when the response was received. The resulting protocol is

shown in Figure 5. An alternative design would be to omit the challenge message containing the nonce. For example, we could compute the MAC of a counter. However, this protocol is vulnerable to a pre-play attack, where a response is collected and replayed at a later time. Alternatively, the MAC could be computed of a timestamp. This approach gives the server assurance of when the response was generated. But, it requires the device to have a real-time clock. Doing so would increase power requirements and limit the device's lifespan because the battery could not be replaced by the client without desynchronising the clock. Alternatively, a backup battery could be included, but this would significantly increase the device cost.

All of these protocols/approaches have a major weakness; namely, if the device is stolen, then the adversary can generate valid authentication responses.

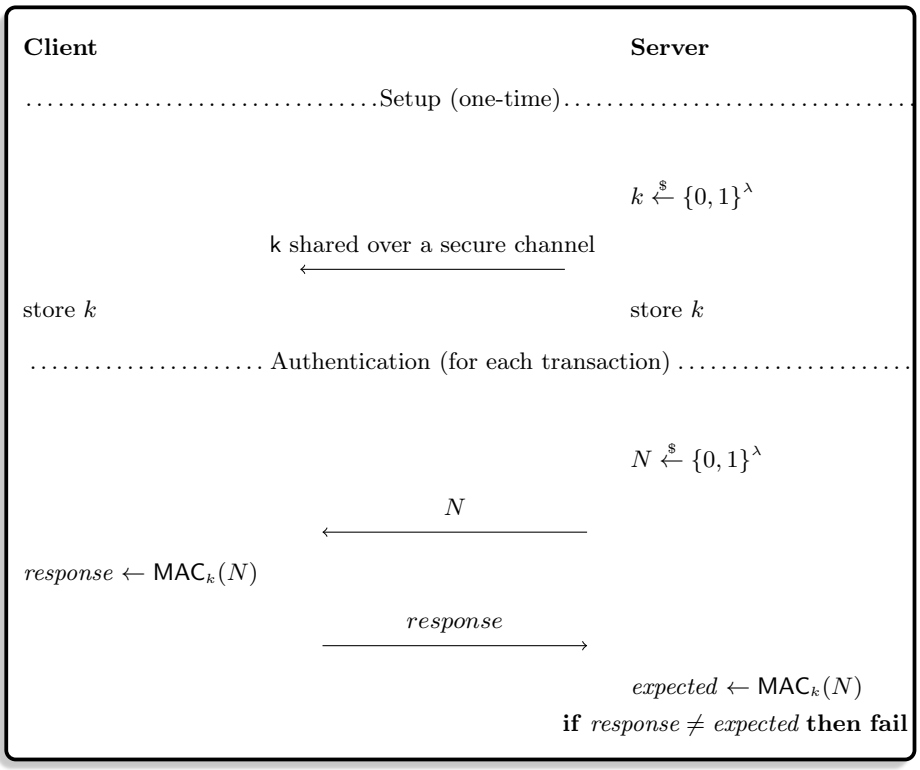

Fig. 5: Straw-man protocol I.

D.2 Straw-man Proposal II

We extend the previous challenge-response protocol to compute the response over both the challenge and a client's PIN, to prevent an adversary who has stolen the device from completing the authentication phase. This creates a 2FA

scheme, depending on something the client has (i.e., the authentication device) and something the client knows (i.e., the PIN). The server can compute the expected response and validate the response produced by the device. If this validation succeeds, then the server has the same assurances of Straw-man Proposal 1 and additionally knows that the correct PIN was entered into the device. To this end, we could store the PIN on the server.

Nevertheless, it is undesirable for the server to know the PIN, as the client may use the same PIN for other unrelated purposes. Having the server store the hash of the PIN would not help because the low entropy of a convenient PIN (around 13 bit for a 4-digits) is trivially vulnerable to a brute-force preimage attack. We can avoid this problem by replacing the PIN in the protocol with a verifier, which is the output of a PRF computed over the PIN under a secret key held only by the device. This verifier must be sent to the server when the device is enrolled. Given the correct PIN, the device could compute the verifier. In this case, a corrupt server would not be able to recover the PIN from the verifier, without knowledge of the secret key. We can incorporate a description of the transaction into the challenge message and computation of the authentication response. This transaction is generated by the server to indicate to the client what action will be performed if the authentication succeeds. The resulting protocol is shown in Figure 6.

An alternative protocol design would be to store the PIN on the device, and for the device to only permit the authentication key k to be used if the PIN is entered correctly. For this design to be resistant to an adversary who has stolen the device, it must not be feasible to extract the PIN and must not be feasible to bypass the PIN verification. This functionality requires security features not commonly available on low-cost microcontrollers. Security assured co-processors are available, but would substantially increase the cost of the device. The protocol above meets many desirable criteria for an authentication protocol. Specifically, a verified authentication response gives the server assurance that (a) the device is present, due to the random nonce, (b) the correct device was used, due to the use of the key, (c) the device has not been stolen, due to the PIN, and (d) the client saw the transaction that the server is about to perform, due to the inclusion of the transaction's description within the MAC computation. The protocol does not require the device to have a real-time clock, so a single client-replaceable battery may be used. The PIN is also not stored by the device and so no special tamper-resistant hardware is necessary. There is no need to protect the authentication key against physical tampering because anyone with access to the device could simply use the device to perform authentication.

There is still a remaining serious risk. Let us suppose that the adversary has recorded a valid authentication response and the corresponding challenge. The adversary who has access to the device can extract the authentication key. Now, the adversary has all the information needed to locally brute-force the short PIN, and then generate a valid response to any future authentication challenge from the server.

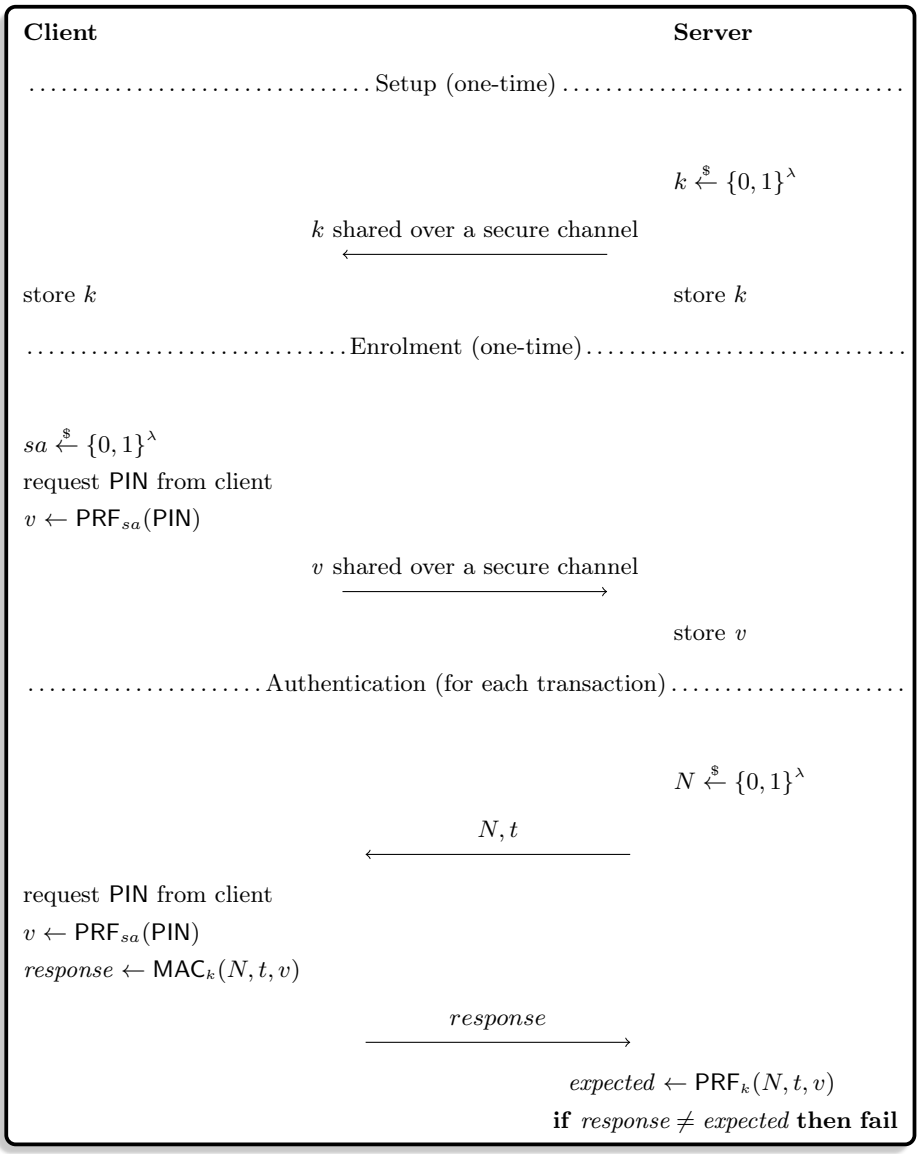

Fig. 6: Straw-man protocol II.

E Informal Security Analysis

In this section, we informally analyse the security of the proposed protocol. We analyse its security through five scenarios defined in terms of adversary capabilities and protection goals. The scenarios are designed to assume a strong adversary so that the results are generalisable to other situations, but are constrained so as to make sense, e.g., we assume that at least one factor is secure.

E.1 Threats and Protection objectives

In this section, we first list a set of threats that our protocol must resist.

- *T.DEV: Device access.* An adversary may steal the authentication device. The adversary will then know k^C, sa^C and the current values of ct^C and st^C, because we assume the device does not take advantage of a trusted chipset. The adversary does not learn v^C, t^M, PIN^C, kt_1^C, kt_2^C, kt_3^C or previous values of st^C; as these are all discarded at the end of a protocol exchange. We assume that the client will not use the device after it has been stolen, and will be issued with a replacement.
- *T.MITM: Man-in-the-middle.* An adversary may have access to the traffic exchanged between the client and server.
- *T.PIN: Knowledge of PIN.* An adversary may know the PIN entered by a client, for example from observing them type it in.
- *T.SRV: Server compromise.* The server is the party relying on the authentication, so it does not make sense for the server to be wholly malicious, i.e., an active adversary. However, it is reasonable to believe that the server database could be compromised, disclosing k^S, v^S, and the current values of ct^C and st^C.

Next, we present the high-level security objective that our protocol must achieve.

- *O.AUTH: Authentication.* If the server considers the authentication to have succeeded then the correct device was used and the correct PIN was entered.
- *O.TRAN: Transaction authentication.* If the server considers the authentication to have succeeded then the correct device was used, the correct PIN was entered, and the device showed the correct transaction.
- *O.PIN: PIN protection.* The adversary should not be able to discover the client's PIN.

E.2 Scenarios

In this section, we briefly explain why the protocol meets its objective in different threat scenarios.

1. *O.AUTH against T.PIN and T.MITM.* The first scenario we consider is the case where the adversary does not have access to the authentication device but does know the client's PIN and communication between the client and server. For the adversary to perform a successful authentication, it must compute $\mathsf{PRF}_{kt_3^C}(N^M \| t^M \| v^C || 1)$. Nevertheless, it does not know kt_3^C or the state from which kt_3^C has been generated; since kt_3^C is an output of PRF and is sufficiently large, it is computationally indistinguishable from a truly random value. The probability of finding it is negligible in the security parameter. Thus, the only party which will generate a valid response is the device itself (when the PIN is provided) at line 22 of Figure 3. We have already assumed that the adversary does not have access to the device; therefore, it cannot generate a valid response.

2. *O.AUTH against T.DEV and T.MITM.* In this scenario, the adversary has compromised the client's device (but not its PIN), has records of previous messages, and wishes to impersonate the client. Since the random challenge in the expected response is unique, and the PRF provides an unpredictable output, previous responses will not be valid; so, a reply attack would not work. The adversary can use the device to discover (kt_2^C, kt_3^C) and all the parameters of the response message, except the PIN. In this case, it has to perform an online dictionary attack by guessing a PIN, using the extracted parameters to generate a response, and sending the response to the server. But, the server will lock out the device if the number of incorrect guesses exceeds the predefined threshold. Other places where the PIN is used are in (i) the enrolment response, where the verifier derived from the PIN is encrypted under an evolving fresh secret key, and (ii) the authentication response, where the response is a pseudorandom value derived from the PIN's verifier using an evolving fresh secret key. In both cases, the evolving keys cannot be obtained from the current state, due to the security of FS-PRG.
3. *O.PIN against T.DEV and T.MITM.* The adversary has compromised the device and wishes to obtain the client's PIN. As with Scenario 2, the PIN cannot be obtained from the device, the responses in the authentication or enrolment phases.
4. *O.PIN against T.SRV and T.MITM.* The adversary has compromised the server and wishes to obtain the client's PIN. In this case, the adversary has learned the verifier but does not know the value of the secret key, used to generate the verifier. If the server retains values of the verifier for previous PINs (in the case where the server does not delete them), then the adversary would also learn further verifiers for the same device. The PIN is only used for computing the verifier, so the only way to obtain the PIN would be to find the key of the PRF which is not possible except for a negligible probability in the security parameter. The only information this discloses is that if two values for the verifier are equal, then that implies that two PINs for the same device were equal. Even this minimal information leakage can be removed if the server rejects the PINs that were used before.
5. *O.TRAN against T.PIN and T.MITM or T.DEV and T.MITM.* As we discussed above, an adversary cannot successfully authenticate, even if it sees the traffic between the client and server and has access to either the PIN or the device. Furthermore, due to the security of the authenticated encryption, the device can detect (except for a negligible probability) if the transaction's description, that the server sends to it, has been tampered with.

E.3 Excluded Scenarios

We exclude some scenarios that do not make sense or are not possible to secure against.

- *Compromised PIN and device.* If the adversary has compromised both factors of a 2FA solution, then the server cannot distinguish between the adversary and the legitimate client.

- *Authentication on server compromise.* If the adversary has compromised the server, then it can either directly perform actions of the server or change keys to ones known by the adversary. Therefore, it does not make sense to aim for O.AUTH in this situation.
- *Compromised server and device.* If the adversary has compromised the server and the device, then the PIN can be trivially brute-forced with knowledge of v^S and sa^C.

F Formal Security Analysis

In this section, we present the security proof of the protocol, presented in Section 5. First, we prove the semantic security of the scheme and then prove its authentication.

F.1 Semantic Security

In this section, we assert that under standard assumptions protocol ψ, presented in Section 5, securely distributes session keys. To do so, we incrementally define a sequence of games starting at the real game G_0 and ending up at G_7. We first define various events in every game and then explain each game.

- S_i: it takes place if $b = b'$, where b is the bit involved in the test query and b' is the output of $\mathcal{A}$ which wants to guess b.
- $Auth_i$: it occurs if $\mathcal{A}$ generates and sends to the server an authenticator message that is accepted by the server.
- Enc_i: occurs if $\mathcal{A}$ submits data it has encrypted by itself using the correct key that an honest party would use to encrypt.

- ***Game*** G_0: This is the real attack game. Several oracles are available to the adversary; namely, the pseudorandom function (PRF), the encryption/decryption oracles ($\mathtt{Enc}$ and $\mathtt{Dec}$) and all instances C^i and S^j. According to the definition we presented in Section 4, the advantage of the adversary in this protocol is:

$$Adv_{\psi}^{ss}(\mathcal{A}) = 2Pr[S_0] - 1 \tag{3}$$

  Similar to the security proof in [7], we assume that if any of the games halts and $\mathcal{A}$ does not output b', then b' is chosen at random. Also, if $\mathcal{A}$ has not finished playing the game after sending q_s $\mathtt{Send}(.)$ queries or if it plays the game more than a predefined time t, the game is stopped and a random value is assigned to b'.
- ***Game*** G_1: This game is similar to G_0, except that the output of the PRF is replaced by an output of a uniformly random function f, i.e., when the simulator in Figure 7 is used. Since the output of f (in the simulator) and PRF are indistinguishable, except with a negligible probability, we will have:

$$|Pr[S_1] - Pr[S_0]| \leq (q_s + q_p)Adv^{\mathsf{PRF}}(\mathcal{A}) \tag{4}$$

  that captures both send and execute queries. We highlight that as we use a standard PRF, the probability of finding a collision is 0.

- ***Game*** G_2: This game is the same as G_1, with the difference that we simulate the authenticated encryption scheme (i.e., *Enc* and *Dec* algorithms). We replace the output of *Enc* with a uniformly random value picked from the encryption scheme's range. The adversary has access to the encryption and decryption oracles. Since we treat the encryption scheme as a black box, the two games are distinguishable except with a negligible probability; this we will have:
$$|Pr[S_2] - Pr[S_1]| \leq (q_s + q_p)Adv^{Enc}(\mathcal{A}) \tag{5}$$
The above also captures both the send and execute queries. Since we have used a standard encryption scheme, the probability of finding a collision (e.g., two ciphertexts result in the same plaintext or two plaintexts result in the same ciphertext) is 0, as the scheme is bijective.
- ***Game*** G_3: This game is the same as G_2, with the difference that we simulate the verification of a transaction, i.e., via predicate ϕ defined in Section 4. Moreover, we simulate all parties' instances via defining simulators for `Send`, `Execute`, `Reveal`, and `Test` queries. We present the simulators for client's and server's `Send` queries in Figures 8 and 9 respectively. Also, we present the simulators for the rest of the queries in Figure 10. By definition, ϕ is a deterministic function, given the transaction t^C and policy π, it always returns the same output as the client does when verifying t^C in the previous game. Therefore, both ϕ and the client would output identical values, given pair (t^C, π), meaning that their outputs are indistinguishable in both games. Given the above argument, we conclude that:
$$Pr[S_3] - Pr[S_2] = 0 \tag{6}$$
- ***Game*** G_4: This game is the same as G_3, with the difference that when the adversary manages to use the correct encryption key and encrypts (or decrypts) a message itself, then the simulation aborts. Therefore, we have:
$$|Pr[S_4] - Pr[S_3]| \leq Pr[Enc_4]$$
We know that the key has been picked uniformly at random and is of length λ bits (recall that the outputs of PRF have been replaced with truly random values in G_1). Therefore:
$$Pr[Enc_4] = \frac{4(q_s + q_p)}{2^\lambda}$$
and
$$|Pr[S_4] - Pr[S_3]| \leq \frac{4(q_s + q_p)}{2^\lambda} \tag{7}$$
- ***Game*** G_5: In this game, we modify the simulator such that it would abort if the adversary correctly guesses the authenticator. Therefore, we modify the way the server responds to query $\texttt{Send}(S^j, \bar{response}^C)$ as follows:
  1. computes $expected^S \leftarrow \mathsf{PRF}_{k\bar{t}_3^S}(\ddot{N}^S \| t^S \| v^S || 1)$.

2. checks if $\bar{response}^C = expected^S$. It proceeds to the next step if the equation holds.
3. checks if $\Big((C_{ID}, \text{enrolment}), (C_{ID}, \text{authentication}), (\bar{M}, \bar{t}),\ (\bar{M}',\ \bar{t}'),\ (\ddot{M}',\ \ddot{t}'),\ (\hat{M}', \hat{t}'), \bar{response}^C\Big) \in \vec{L}$.
4. checks if $\bar{response}^C \in L_{\mathcal{A}}$.
5. if both checks in steps 3 and 4 fail, then it rejects authenticator $\bar{response}^C$ and terminates without accepting the key. Otherwise, it accepts the key.

This game ensures that if the message (i.e., the authenticator) does not come from the simulator or the adversary (which decrypted $\ddot{M}', \ddot{t}', \hat{M}'$, and $\hat{t}'$, then correctly computed a valid authenticator by querying Update and $\mathsf{PRF}_{k'}$) then it aborts. So, games G_4 and G_5 are indistinguishable unless the server rejects a valid authenticator. However, this means the adversary has correctly guessed the output of PRF. Thus,

$$|Pr[S_5] - Pr[S_4]| \leq \frac{q_s}{2^\lambda} \tag{8}$$

- ***Game*** G_6: In this game, we modify the simulator in a way that it would abort if $\mathcal{A}$ decrypts $(\ddot{M}', \ddot{t}', \hat{M}', \hat{t}')$ and uses the result to generate and send a valid authenticator to the server. To do so, we modify the way the server responds to query $\mathtt{Send}(S^j, \bar{response}^C)$, as follows:
  1. computes $expected^S \leftarrow \mathsf{PRF}_{k\bar{t}_3^S}(\ddot{N}^S \| t^S \| v^S || 1)$.
  2. checks if $\bar{response}^C = expected^S$. It proceeds to the next step if the equation holds.
  3. checks if $\Big((C_{ID}, \text{enrolment}), (C_{ID}, \text{authentication}), (\bar{M}, \bar{t}),\ (\bar{M}',\ \bar{t}'),\ (\ddot{M}',\ \ddot{t}'),\ (\hat{M}', \hat{t}'), \bar{response}^C\Big) \in \vec{L}$. If this check fails, then it rejects authenticator $\bar{response}^C$ and terminates, without accepting any key.
  4. checks if $(\ddot{N}^S \| t^S \| *, \bar{response}^C) \in L_{\mathcal{A}}$.
  5. aborts, if the above check (in step 4) passes.

  The above modification ensures that all valid authenticators are sent by the simulator. Let $\hat{Auth}_6$ be the event that the check in step 4 passes. Games G_5 and G_6 are indistinguishable unless $\hat{Auth}_6$ occurs. Hence,

$$|Pr[S_6] - Pr[S_5]| \leq Pr[\hat{Auth}_6]$$

  We know that $\hat{Auth}_6$ occurs with probability $\frac{q_s}{2^\lambda}$ when the query $q = \ddot{N}^S \| t^S \| *$ to PRF results in $\bar{response}^C$. Thus,

$$|Pr[S_6] - Pr[S_5]| \leq \frac{q_s}{2^\lambda} \tag{9}$$

- ***Game*** G_7: In this game, we modify the simulator such that it would abort if the adversary comes up with the authenticator and session key without decrypting $(\ddot{M}', \ddot{t}', \hat{M}', \hat{t}')$. Therefore, we modify the way the client processes query $\mathtt{Send}(C^i, (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'))$ as follows.
  1. compute $\bar{response}^C \leftarrow \mathsf{PRF}_{i'}(\ddot{M}' || 1)$.

2. compute $\bar{sk}^C \leftarrow \mathsf{PRF}_{i'}(\ddot{M}'||2)$.

We also amend the way the server compiles query $\texttt{Send}(S^j, \bar{response}^C)$ as follows.

a checks if $(\ddot{M}'||1, \bar{response}^C) \in L_{\mathcal{A}}$ or $(\ddot{M}'||2, \bar{sk}^C) \in L_{\mathcal{A}}$.

b aborts, if either of the above checks (in step a) passes.

Let $\hat{Auth}_7$ be the event that the check in step a passes. Games G_6 and G_7 are indistinguishable unless $\hat{Auth}_7$ occurs. Therefore,

$$|Pr[S_7] - Pr[S_6]| \leq Pr[\hat{Auth}_7]$$

Event $\hat{Auth}_7$ occurs with probability $\frac{2q_s}{2^\lambda}$ when the query $q = \ddot{M}'||1$ to PRF results in $\bar{response}^C$ or $q = \ddot{M}'||2$ to PRF results in $\bar{sk}^C$. Thus,

$$|Pr[S_7] - Pr[S_6]| \leq \frac{2q_s}{2^\lambda} \tag{10}$$

Moreover, the session key and authenticator are random values, as they are the outputs of PRF whose secrete key is not known. Therefore, $Pr[S_7] = \frac{1}{2}$. By summing up all the above relations 4-10, we would have

$$|Pr[S_7] - Pr[S_0]| \leq (q_s + q_p)\Big(Adv^{\mathsf{PRF}}(\mathcal{A}) + Adv^{Enc}(\mathcal{A})\Big) + \frac{4(q_s + q_p)}{2^\lambda} + \frac{4q_s}{2^\lambda} \tag{11}$$

By combining Equations 3 and 11, we would have:

$$Adv^{ss}_{\psi}(\mathcal{A}) \leq 2(q_s + q_p)\Big(Adv^{\mathsf{PRF}}(\mathcal{A}) + Adv^{Enc}(\mathcal{A})\Big) + \frac{8(2q_s + q_p)}{2^\lambda}$$

This completes the proof.

Pseudorandom function

The simulator upon receiving query (PRF, q) acts as follows.

- picks a function f, i.e., $f \xleftarrow{\$} Func$, where $Func$ is the set of all functions mapping $|q|$-bit strings to $|q|$-bit strings.
- adds record $(q, f(r))$ to list $L_{\mathcal{A}}$ and then outputs $f(r)$.

Fig. 7: Pseudorandom function's simulator.

$\underline{\texttt{Send}(C^i, .)}$

This query is dealt with as below:

– if the client's instance is not in the "expecting" state and it receives query $\texttt{Send}(C^i, \text{start}, \text{phase})$, where phase $\in$ {enrolment, authentication} then it:
  1. generates pair (C_{ID}, phase).
  2. responds to the query with (C_{ID}, phase).
  3. sets the client's instance state to expecting.

– if the client's instance state is in expecting, then:

• upon receiving $\underline{\texttt{Send}(C^i, (\bar{M}, \bar{t}))}$, it:
  1. authenticates and decrypts the ciphertext $\bar{M}$ as $(p, b) \leftarrow \mathsf{Dec}_{k^C}(\bar{M}, \bar{t})$. If the authentication fails (i.e., $b \neq 1$), it halts.
  2. extracts (N^M, ct^M) from plaintext p and checks if $ct^M > ct^C$. If the check fails, it halts.
  3. generates v^C using sa^C and PIN^C as follows $v^C \leftarrow \mathsf{PRF}_{sa^C}(\mathsf{PIN}^C)$. Then, it updates its state as follows: $\forall i, 1 \leq i \leq ct^M - ct^C$: (a) $ct^C \leftarrow ct^C + 1$ and (b) $(k, st^C) \leftarrow \mathsf{Update}(st^C, ct^C)$.
  4. encrypts $p' = N^M || v^C$ using key k as follows: $(\bar{M}', \bar{t}') \leftarrow \mathsf{Enc}_k(p')$, which results in a ciphertext $\bar{M}'$ and tag $\bar{t}'$.
  5. responds to the query with $(\bar{M}', \bar{t}')$. It sets the client's instance state to "not expecting".

• upon receiving $\underline{\texttt{Send}(C^i, (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'))}$, it:
  1. authenticates and decrypts the ciphertext $\hat{M}'$ as follows: $(p', b') \leftarrow \mathsf{Dec}_{k^C}(\hat{M}', \hat{t}')$. If the authentication fails (i.e., $b' \neq 1$), it halts.
  2. extracts (tmp_{ct^M}, ct^M) from p' and checks if $tmp_{ct^M} > ct^C$. If the check fails, it halts.
  3. updates its state as follows: $\forall i, 1 \leq i \leq tmp_{ct^M} - ct^C$: (a) $ct^C \leftarrow ct^C + 1$ and (b) $(k, st^C) \leftarrow \mathsf{Update}(st^C, ct^C)$.
  4. authenticates and decrypts $\ddot{M}'$ as follows: $(p, b) \leftarrow \mathsf{Dec}_k(\ddot{M}', \ddot{t}')$. If the authentication fails (i.e., $b \neq 1$), it halts.
  5. extracts (N^M, t^M) from plaintext p.
  6. runs the predicate, $y \leftarrow \phi(t^M, \pi)$. If $y = 0$, it halts.
  7. generates v^C using sa^C and PIN^C as follows, $v^C \leftarrow \mathsf{PRF}_{sa^C}(\mathsf{PIN}^C)$.
  8. updates its state one more time as follows, $(k', st^C) \leftarrow \mathsf{Update}(st^C, ct^M)$.
  9. computes the authenticator: $\bar{response}^C \leftarrow \mathsf{PRF}_{k'}(N^M || t^M || v^C || 1)$ and session key: $\bar{sk}^C \leftarrow \mathsf{PRF}_{k'}(N^M || t^M || v^C || 2)$.
  10. responds the send query with $\bar{response}^C$. It makes the client's instance accept the key and then terminates the instance.

To keep track of all the exchanged messages, it stores the above incoming and going messages in vector $\vec{L}$. So, we have $\Big((C_{ID}, \text{enrolment}), (C_{ID}, \text{authentication}), (\bar{M}, \bar{t}), (\bar{M}', \bar{t}'), (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'), \bar{response}^C\Big) \in \vec{L}$.

Fig. 8: Simulators for Send query to a client's instance.

$\underline{\underline{\texttt{Send}(S^j, .)}}$

This query is dealt with as below:

- upon receiving $\underline{\texttt{Send}(S^j, (C_{ID}, \text{enrolment}))}$, it:
  1. increments its counter as $ct^S \leftarrow ct^S + 1$, updates its state as $k\bar{t}_1^S, st^S \leftarrow \mathsf{Update}(st^S, ct^S)$, picks a random value $\bar{N}^S \xleftarrow{\$} \{0,1\}^\lambda$, and generates ciphertext and tag $(\bar{M}, \bar{t}) \leftarrow \mathsf{Enc}_{k^S}(\bar{N}^S || ct^S)$.
  2. responds to the query with $(\bar{M}, \bar{t})$. The state of the server instance is set to "expecting".
- upon receiving $\underline{\texttt{Send}(S^j, (\bar{M}', \bar{t}'))}$, it:
  1. authenticates and decrypts the ciphertext $\bar{M}'$ as $(p', b') \leftarrow \mathsf{Dec}_{k\bar{t}_1^S}(\bar{M}', \bar{t}')$. If the authentication fails (i.e., $b' \neq 1$), it halts.
  2. extracts (N^M, v^M) from plaintext p'. It sets $v^S \leftarrow v^M$ and also checks if $N^M = \bar{N}^S$. If the equation does not hold, it halts. The state of the server instance is set to expecting.
- upon receiving $\underline{\texttt{Send}(S^j, (C_{ID}, \text{authentication}))}$, it:
  1. increments its counter $ct^S \leftarrow ct^S + 1$, updates its state $k\bar{t}_2^S, st^S \leftarrow \mathsf{Update}(st^S, ct^S)$, temporarily stores this counter $tmp_{ct^S} \leftarrow ct^S$, increments the counter again $ct^S \leftarrow ct^S + 1$, updates its state again $k\bar{t}_3^S, st^S \leftarrow \mathsf{Update}(st^S, ct^S)$, and picks a random value $\ddot{N}^S \xleftarrow{\$} \{0,1\}^\lambda$.
  2. generates two pairs of ciphertext and tag as follows, $(\ddot{M}', \ddot{t}') \leftarrow \mathsf{Enc}_{k\bar{t}_2^S}(\ddot{N}^S || t^S)$ and $(\hat{M}', \hat{t}') \leftarrow \mathsf{Enc}_{k^S}(tmp_{ct^S} || ct^S)$.
  3. responds to the query with $(\ddot{M}', \ddot{t}')$, $(\hat{M}', \hat{t}')$. The state of the server instance is set to expecting.
- upon receiving $\underline{\texttt{Send}(S^j, \bar{response}^C)}$, it:
  1. computes $expected^S \leftarrow \mathsf{PRF}_{k\bar{t}_3^S}(\ddot{N}^S \| t^S \| v^S || 1)$. It checks whether $\bar{response}^C = expected^S$. If the equality does not hold, the server instance terminates without accepting any session key.
  2. generates the session key $s\bar{k}^S \leftarrow \mathsf{PRF}_{k\bar{t}_3^S}(\ddot{N}^S \| t^S \| v^S || 2)$. It accepts the key and terminates.

Fig. 9: Simulators for Send query to a server's instance.

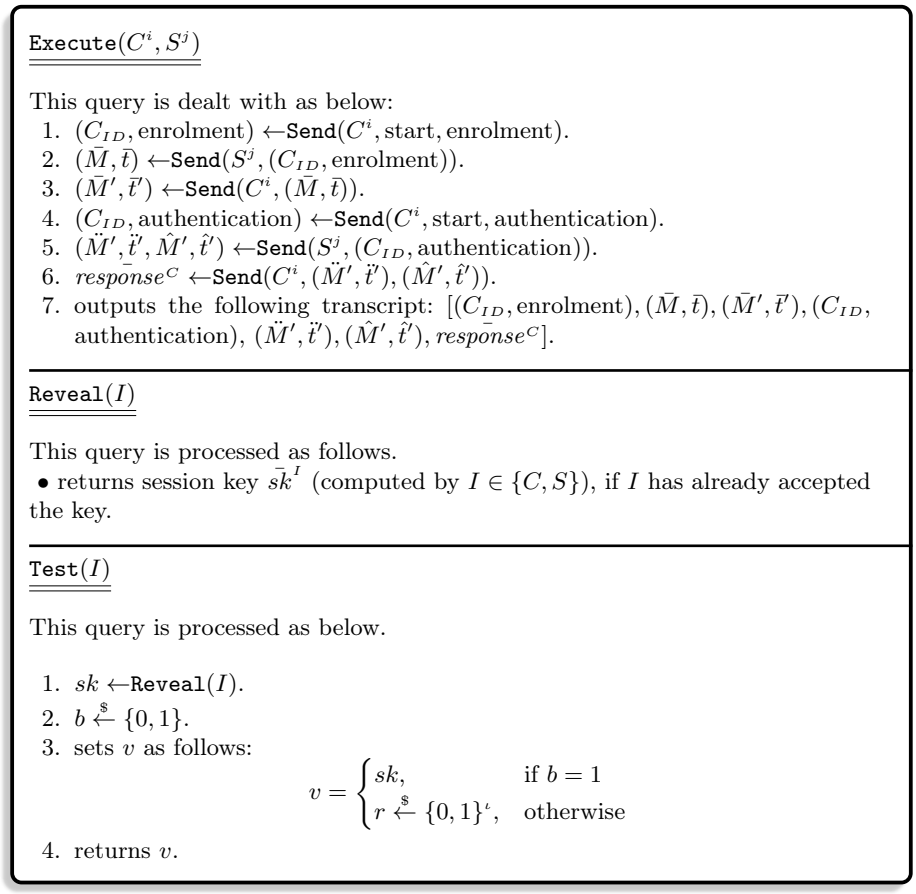

Fig. 10: Simulators for Execute, Reveal, and Test queries.

F.2 Authentication

In this section, we prove the protocol's authentication. We begin with the case where the adversary $\mathcal{A}$ has access to the traffic between the two parties and wants to impersonate the client, C; we denote such a case with $a\bar{ut}$. The Authentication proof relies on the semantic security proof (and games) we presented in Section F.1. Now, we outline the proof. By definition, it holds that:

$$Adv_{\psi}^{a\bar{ut}}(\mathcal{A}) = Pr[Auth_0] \tag{12}$$

Also, we can extend Equation 4 to:

$$|Pr[Auth_1] - Pr[Auth_0]| \leq (q_s + q_p) Adv^{\mathsf{PRF}}(\mathcal{A}),$$

because the only difference between the two games (i.e., G_0 and G_1) is that the output of the PRF is replaced with an output of a uniformly random function f.

Furthermore, we can extend Equations 4-10 as follows:

$$
\begin{aligned}
&|Pr[Auth_1] - Pr[Auth_0]| \leq (q_s + q_p) Adv^{\mathsf{PRF}}(\mathcal{A}) \\
&|Pr[Auth_2] - Pr[Auth_1]| \leq (q_s + q_p) Adv^{Enc}(\mathcal{A}) \\
&Pr[Auth_3] - Pr[Auth_2] = 0 \\
&|Pr[Auth_4] - Pr[Auth_3]| \leq \frac{4(q_s + q_p)}{2^\lambda} \\
&|Pr[Auth_5] - Pr[Auth_4]| \leq \frac{q_s}{2^\lambda} \\
&|Pr[Auth_6] - Pr[Auth_5]| \leq \frac{q_s}{2^\lambda} \\
&|Pr[Auth_7] - Pr[Auth_6]| \leq \frac{2q_s}{2^\lambda}
\end{aligned}
$$

Moreover, since the authenticator is a random value in G_7, it holds that $Pr[Auth_7] = \frac{q_s}{2^\lambda}$. We conclude the proof, by summing up the above relations and combining with Equation 12:

$$
Adv_{\psi}^{a\bar{u}t}(\mathcal{A}) = Pr[Auth_0] \leq (q_s + q_p)\Big(Adv^{\mathsf{PRF}}(\mathcal{A}) + Adv^{Enc}(\mathcal{A})\Big) + \frac{9q_s + 4q_p}{2^\lambda} \quad (13)
$$

Next, we proceed to the case where the adversary is given further access to the PIN, i.e., $\mathcal{A}$ can also send query $\texttt{Corrupt}(C, 1)$. We argue that given such an extra capability does not affect the adversary's advantage and the above analysis (as the protocol and its analysis have relied on the security of the CCA-secure symmetric encryption and PRF). Now move on to the case where $\mathcal{A}$ (a) is given all the parameters stored in the hardware token, and (b) has access to all the traffic between the two parties, i.e., $\mathcal{A}$ can also send query $\bar{\text{Cpt}}_2 = \texttt{Corrupt}(C, 2)$. We argue that in this case, the upper bound of $\mathcal{A}$'s advantage will be changed as follows: $Adv_{\psi, \bar{\text{Cpt}}_2}^{aut}(\mathcal{A}) \leq \frac{q_s}{N}$. The reason for such a big change is that in this case, $\mathcal{A}$ has all secret parameters, except the PIN and verifier v^C. [2] Thus, when we take the forward security into account, the advantage of the adversary (due to the union bound) is as follows:

$$
Adv_{\psi}^{aut}(\mathcal{A}) \leq (q_s + q_p)\Big(Adv^{\mathsf{PRF}}(\mathcal{A}) + Adv^{Enc}(\mathcal{A})\Big) + \frac{9q_s + 4q_p}{2^\lambda} + \frac{q_s}{N}
$$

F.3 PIN's Privacy Against A Corrupt Server

In the case where the adversary (i) has access to the parties' traffic and (ii) can make query $\texttt{Corrupt}(S, 1)$, to extract all parameters of the server, then the

[2] The case where $\mathcal{A}$ has the additional capability to send query $\texttt{Corrupt}(C, 2)$ was never discussed and analysed in [7]. However, we noticed that $\mathcal{A}$ in that scheme would have the same upper bound advantage as $\mathcal{A}$ in our scheme does.

probability that the adversary can find the valid PIN depends on the probability of finding the correct PIN and finding C's correct key of PRF; therefore, the probability is at most $\frac{q_p}{2^\lambda N}$.

G Full Evaluation

In this section, we analyse and compare the 2FA protocol, we presented in Section 5, with the smart-card-based protocol proposed in [33] and the hardware token-based protocol in [16] because the latter two protocols are relatively efficient, do not rely on secure chipsets, and they consider the same security threats as we do, e.g., resistance against card/token loss, against an offline attack, against a corrupt server.

G.1 Computation Cost

We start by analysing our protocol's computation cost. First, we focus on the protocol's enrolment phase. The client's computation cost, in this phase, is as follows. It invokes the authenticated encryption scheme 2 times. It also invokes once the pseudorandom function, PRF. Moreover, the server invokes the authenticated encryption scheme twice, and calls PRF only once, in this phase. Now, we move on to the authentication phase. The client invokes the authenticated encryption scheme 2 times and invokes PRF 4 times. In this phase, the server invokes the authenticated encryption scheme and PRF 2 and 4 times respectively.

Next, we analyse the computation cost of the protocol in [33]. We consider all operations performed on the smart card or card reader as client-side operations. The enrolment phase involves 3 and 2 invocations of a hash function at the client and server sides respectively. This protocol has an additional phase called login which costs the client 5 invocations of the hash function and 2 modular exponentiations for each authentication. The verification requires the server 6 invocations of the hash function and 2 modular exponentiations. This phase requires the client to perform 1 modular exponentiation and invoke the hash function 3 times.

Now, we analyse the computation cost of the protocol presented in [16]. In our analysis, due to the high complexity of this protocol, we estimate the protocol's *minimum* costs. The actual cost of this protocol is likely to be higher than our estimation. The protocol's phases have been divided into enrolment and login, i.e., verification. The enrolment phase requires a client to perform single modular exponentiation and invoke a hash function 2 times. It also involves, as a subroutine, the initialisation of asymmetric "password-authenticated key exchange" (PAKE) proposed in [13], which involves at least 2 modular exponentiations, 1 invocation of hash function and symmetric key encryption. In the login phase, the client performs at least 7 modular exponentiations. In the login phase, the server invokes a pseudorandom function once and performs at least 2 modular exponentiations and 2 symmetric-key encryptions (due to the execution of PAKE).

Thus, our protocol and the ones in [33,16] involve a constant number of symmetric key primitive invocations; however, our protocol does not involve any modular exponentiations, whereas the protocol in [33,16] involves a constant number of modular exponentiations which leads to a higher cost.

G.2 Communication Cost

We first analyse our protocol's communication cost. In the enrolment phase, the client only sends two pairs of messages: (C_{ID}, enrolment) and (M', t'), where the total size of messages in the first pair is about 250 bits (assuming the ID is of length 128 bits), while the total size of messages in the second pair is about 512 bits as they are the outputs of symmetric-key primitives, i.e., symmetric key encryption and message authentication code schemes whose output size is 256 bits. The server sends out only a single pair (M', t') whose total size is about 512 bits. The parties' communication cost in the authentication phase is as follows. The client only sends three messages: (C_{ID}, authentication, $response^C$), where the combined size of the first two messages is about 250 bits while the third message's size is about 256 bits. The server sends only two pairs of messages $(\ddot{M}, \ddot{t})$ and $(\hat{M}, \hat{t})$ with a total size of 1024 bits. Therefore, the total communication cost that our protocol imposes is about 2804 bits.

Next, we evaluate the cost of the protocol in [33]. The client's total communication cost in the enrolment and login phases is 1792 bits. Note that we set the client's ID's size to 128 bits and we set the hash function output size to 160 bits, as done in [33]. In the verification phase, the client sends to the server a single value of size 160 bits. In the verification, the server sends to the client two values that in total costs the server 1184 bits. So, this protocol's total communication concrete cost is about 3136 bits.

Now, we analyse the communication cost of the protocol in [16]. As before, in our cost evaluation, we estimate the protocol's minimum cost. In the enrolment phase, a client sends a random key, of a pseudorandom function, to the server and the device, where the size of the key is about 128 bits. It also, due to the initialisation of PAKE, sends a 128-bit value to the server. In the login phase, the client sends out three parameters of size 128 bits and a single parameter of size 20 bits. It also invokes PAKE with the server that requires the client to send out at least one signature of size 1024 bits. The device sends to the client a ciphertext of asymmetric key encryption which is of size 1024 bits along with a 20-bit message. Thus, the client-side total communication cost is at least 2856 bits. The server in the login phase sends out a message *zid* of size 20 bits and invokes PAKE that requires the server to send out at least a ciphertext of symmetric key encryption which is of size 1024 bits. So, the server-side communication cost is at least 1044 bits. So, the total communication cost of this protocol is at least 3900 bits.

Hence, our protocol imposes a 10% and 40% lower communication cost than the protocols in [33] and [16] do respectively.

G.3 Other Features

In our protocol, a client needs to know only a single secret (i.e., a PIN). Nevertheless, in the protocol in [33] a client requires to know (and insert into the verification algorithm) an additional secret; namely, a secret random ID. Thus, the client needs to remember two secrets in total. As shown in [28], this scheme will not remain secure, even if only the client's ID is revealed. Furthermore, the protocol in [16] requires the client to remember or locally store at least one cryptographic secret key of sufficient length, e.g., 128 bits; this secret key is generated via invocation of a subroutine protocol (called PAKE) and must not be kept on the device. Furthermore, our protocol is secure in the standard model while the protocols in [33,16] are in the non-standard random oracle model.