A Forward-secure Efficient Two-factor Authentication Protocol

Abstract. Two-factor authentication (2FA) is commonly used for controlling access to high-value online accounts, particularly where financial transactions are involved. In such schemes, knowledge factors (*e.g.*, PIN) and hardware token possession are required to complete authentication. However, existing 2FA protocols leave the PIN vulnerable should the server, user's computer, or hardware token be compromised. In this paper, we propose a 2FA protocol proved to be secure against adversaries who can (a) observe the traffic between a user and server and (b) have physical access to the user's hardware token, or its PIN, or breach the server. Compared to previous work, our protocol reduces the cost of authentication tokens by not requiring tamper-resistant hardware or that the hardware token be connected to the computer, and by only using efficient symmetric-key primitives. Furthermore, our protocol is highly usable, requiring the user only to remember a short PIN and type short authentication codes; it imposes up to 40% lower communication overhead compared to the state-of-the-art. The protocol achieves these goals through a novel combination of splitting secrets between the server and hardware token and key-evolving symmetric-key encryption.

1 Introduction

Two-factor authentication (2FA) is increasingly required for access to online services, as a way to mitigate the risk of a single factor (typically a password) being compromised. This trend has been accelerated by regulations, such as the Payment Services Directive 2 (PSD 2) [1] in the EU and federal requirements in the US [10]. 2FA can bring significant security benefits, but only if the solution is secure, usable, and cost-effective. However, existing systems fall short of these requirements. In this paper, we propose a new 2FA protocol that significantly improves existing systems and so facilitates the wider adoption of 2FA for authenticating access to online services, in particular authorising financial transactions.

2FA requires the combination of two factors out of knowledge (*e.g.*, PIN or a password), possession (*e.g.*, of a hardware device/token), and biometrics (*e.g.*, a fingerprint). In this paper, we will focus on combining knowledge and possession, since biometrics require that devices have a special sensor and so imposes a significant cost penalty. Biometrics also have shortcomings in that they cannot be effectively revoked, will fail to work for certain people, and create privacy concerns. Our requirements for the 2FA protocol are therefore that it be:

- **secure:** provably capable of preventing unauthorised access in as a wide range of circumstances as possible, including compromise of the server, com-

promise of the hardware token, compromise of the user's computer, and compromise of the communication network.
- **usable:** create a minimal imposition on the user and environment by not requiring special software to be installed on the user's computer and not requiring that the user remember long passwords or enter long strings into the computer or hardware token.
- **cost-effective:** the hardware token must be cheap and so must not be required to have tamper-resistant trusted hardware or the ability to perform complex computations.

Furthermore, the 2FA protocol must support **transaction authentication** *i.e.*, be able to bind its execution to a particular transaction that is shown to the user on a trusted display, as required by the PSD2. This means that for the authentication protocol to succeed the user must have had the ability to check the transaction details and so be able to detect if malware on their computer has tampered with the transaction details shown on screen.

Researchers and companies have proposed various 2FA solutions based on a combination of PIN and token possession. Some of these solutions offer a strong security guarantee against an adversary which may (a) observe the communication between a user and server and (b) have physical access to the user's token, or its PIN, or even breaches the server. Nevertheless, these solutions suffer from a subset of the following shortcomings: (i) require a user to remember multiple secret values (instead of a single PIN) to prove its identity which ultimately harms these solutions' usability, (ii) require users to fully trust their personal computers in which they insert their PINs, (iii) rely on a trusted chipset, (iv) involve several modular exponentiations that make the token's battery power run out quickly, or (v) have been proven in the non-standard random oracle model.

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

- requires a user to remember and type into the token only a single PIN.
- allows the token to generate a short authentication message.
- does not involve any modular exponentiations.
- is proved secure in a standard model.
- does not put any security assumptions on users' personal computers.
- imposes up to 40% lower communication costs than the state-of-the-art protocols designed to remain secure against the aforementioned adversary.

To attain its goals, our protocol does not use any trusted chipset; instead, it relies on a novel combination of the following two approaches. Firstly, neither the server nor the hardware token has the ability to verify the PIN – secret information stored by both is needed to do so. This approach ensures that an adversary cannot retrieve the PIN, even if it penetrates either location. Secondly, it (a) requires that the server and token 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 immediately 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 subsequently breaks into the token or server. We formally prove the security of this protocol.

2 Survey of Related Work

In this section, first, we 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.

2.1 Common Approaches for Generating OTP

In authentication that relies on a combination of knowledge and possession factors, once the user enters the secret into the hardware token, the token (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 token (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.

2.2 Variants of OTP Hardware Tokens

Connected Tokens. This type of token requires a user to physically connect the token to their computer (*e.g.*, a laptop or card reader) via which the user is authenticating. Once it is connected, the token 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 specifications for USB hardware tokens. YubiKey[1] is one of the well-known ones implementing FIDO specifications. The FIDO Alliance has proposed a standard that aims at allowing users 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 [25,7,21] and formal analysis, *e.g.*, in [11]. Also, for such devices to work there must be corresponding software installed on the computer, and this might not be possible on shared devices or computers implementing a corporate IT policy. If any special software is needed it would have to be implemented for

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

every supported operating system and processor architecture, and periodically updated, so imposing higher development costs.

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 [6], there have been numerous protocols for smart card-based 2FA (*e.g.*, in [13,33,26]) along with a few works that identify vulnerabilities of existing solutions, *e.g.*, in [31,32,8]. However, the existing smart card-based solutions (*e.g.*, in [13,33,26]) are often based on public-key cryptography which imposes a high computation and energy cost; also some solutions (e.g., in [18]) rely on tamper-proof secure chipsets embedded in the card which would ultimately increase its cost.

Disconnected Tokens. This type of token does not have a physical connection to a user's computer making them more convenient than connected tokens. A disconnected token is often equipped with a built-in screen and a keypad allowing a user to 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 [28], 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 user and) stored in the token [4]. Thus, not only the token 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 token, as it can extract the token's secret key. The main advantage of Digipass 770 and Thales Gemalto SWYS QR Token Eco to RSA SecureID is that they allow users to see and verify the transaction details through the token. Therefore, the user 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* users' PINs. Specifically, once a user receives the token, it also receives an activation code from the verifier, *e.g.*, the user's bank. Then, the user (i) registers the activation code in the token and (ii) registers the activation code to the verifier, so the verifier knows that this specific user has a token with the provided activation code. Then, the user registers its PIN in the token which stores it locally. Every time a user uses the verifier's online system (e.g., online banking) and makes a transaction, the system generates and displays an encrypted visual image. The user uses the token (camera) to scan the image and then enters its PIN into the token. Next, the token 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 user to check whether the transaction is the one it has made. If the user accepts the transaction and presses a certain button,

then the token generates and displays an OTP that the user can type into the verifier's online system [24]. Thales Gemalto SWYS QR Token Eco also uses a mechanism similar to the one we described above.

Jules et al. [16] discussed that the adversary who can intercept the user and server's communication and also has physical access to the user's token or the server's storage can extract the user's PIN and impersonate the user. 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 transaction details must be verified by users on the token.

Matsuo *et al.* [22] proposed a scheme that relies on symmetric-key cryptography and is highly efficient. Nevertheless, the scheme uses a *secure chipset* (called TPM in the paper) that keeps a secret key, which most schemes (including ours) try to avoid using secure chipsets. Moreover, the scheme assumes the server is never corrupted. Thus, the protocol does not deal with offline dictionary attacks.

Moreover, Jarecki *et al.* [15] proposed a (single server) protocol to ensure that even if the server or the token is corrupted a user's PIN cannot be extracted and the adversary cannot impersonate an honest user. It is mainly based on a hash function, both symmetric and asymmetric-key encryptions, and (Diffie–Hellman) key exchange. This scheme suffers from several shortcomings; namely, (1) it imposes a high computation and communication cost due to its complexity, the use of public-key cryptography, and numerous rounds of communication, even between the user's computer and token, (2) it requires the token to perform asymmetric-key operations and invoke symmetric-key primitives many times, (3) it requires users to insert their passwords/PINs into their (personal) computers (called client C in the paper) rather than inserting them into the dedicated hardware token, which is problematic because the PINs will be at a higher risk of being exposed to attackers, as users computers are almost always connected to the Internet, are used for various purposes, and are more likely to be broken into, (4) for the protocol's security to hold, it is required that users personal computers to be fully trusted, in the case where the token or the server is corrupted; this is an additional assumption that may not be always desirable. 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 [14,19,20]. 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 [19] 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 user's secret.

Nevertheless, it (a) requires the user to store a sufficiently long secret key (on the mobile phone), (b) requires the laptop/PC that the user 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 [20] 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.* [14] proposes a new protocol that also relies on a phone's TEE, but it improves the protocol presented in [20], 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 users, *e.g.*, transaction details along with the phone number might be sold for targeted advertisements. Moreover, not everyone has a smartphone or is willing to install special software on it, and smartphones have a large attack surface that can be exploited to extract authentication secrets.

3 Notations and Preliminaries

3.1 Notations and Assumptions

To disambiguate the different uses of keys and other items of data, variables are annotated with a superscript to indicate their origin. $\cdot^U$ indicates data stored at the user, $\cdot^S$ means data stored at the server, and $\cdot^M$ indicates data item has been extracted from a message. We define a function `Discard`(.) that takes an array of inputs and securely deletes them (e.g., from storage, memory, catch). We assume the token is not penetrated by an adversary in the (very short) period when the inputs of `Discard`(.) are set and when `Discard`(.) is executed. We refer readers to [27] for a survey of secure data deletion approaches. We also assume that a user will not use its hardware token after it has been stolen (and it will be issued with a replacement with fresh parameters). We present a summary of variables in Table 1.

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 formal definition of a PRF is given by Katz and Lindell [17] and is included in Appendix A.

3.3 Forward-Secure Pseudorandom Bit Generator

A Forward-Secure Pseudorandom Bit Generator (FS-PRG), is a *stateful* object which consists of a pair of algorithms and a pair of positive integers, *i.e.*, $\mathsf{FS\text{-}PRG} = \Big((\mathsf{FS\text{-}PRG.KGen}, \mathsf{FS\text{-}PRG.next}), (b, n)\Big)$, as defined in [3]. The probabilistic key generation algorithm $\mathsf{FS\text{-}PRG.KGen}$ takes a security parameter as

Table 1: Notation used in the protocol.

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

input and outputs an initial state st_0 of length s bits. FS-PRG.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 $st_0 \xleftarrow{\$} \mathsf{FS\text{-}PRG.KGen}(1^\lambda)$ and then running $(out_i, st_i) \leftarrow \mathsf{FS\text{-}PRG.next}(st_{i-1})$ for all $i, 1 \leq i \leq n$. As with a standard pseudorandom bit generator, the 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-PRG 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 A.

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

times and outputs the pair (out_b, st_b) which are the output of FS-PRG.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 message. 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 [17] for a formal definition of authenticated symmetric-key encryption.

4 Threat Mode and System Design

A two-factor authentication scheme involves two players; namely, (1) User (U): an honest party which tries to prove its identity to a server by using a combination of a PIN and a token and (2) Server (S): a semi-honest adversary which follows the protocol's instructions and tries to learn U's PIN. It also tries to authenticate itself to U.

We let the server communicate with the hardware token through the user's computer, *i.e.*, the client. Specifically, similar to previous works (*e.g.*, those in [15,24,30]), we assume the token has a camera that lets the token scan a 2-D barcode containing 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 user and server by U^i and S^j respectively. Each instance is called an oracle.

Figure 1 outlines the message flow of our 2FA scheme during the authentication phase. At a high level, the authentication phase works as follows. Any time the user wants to authenticate itself to the server, the user sends a message to the server via the client. The server replies to the client with a challenge and transaction details (Step 1). The user scans with the token the message that the client received (Step 2). The token shows the transaction details on the screen (Step 3). The user types the PIN into the token (Step 4). The token generates a response (Step 5). The user manually types the response into the client (Step 6). The client sends the response to the server for authentication (Step 7).

To formally capture the capabilities of an adversary, $\mathcal{A}$, in hardware token-based 2FA, we mainly use the (adjusted) model proposed by Bellare *et al.* [2]. In this model, the adversary's capabilities are cast via queries that it sends to different oracles, *i.e.*, instances of the honest parties; the user and server

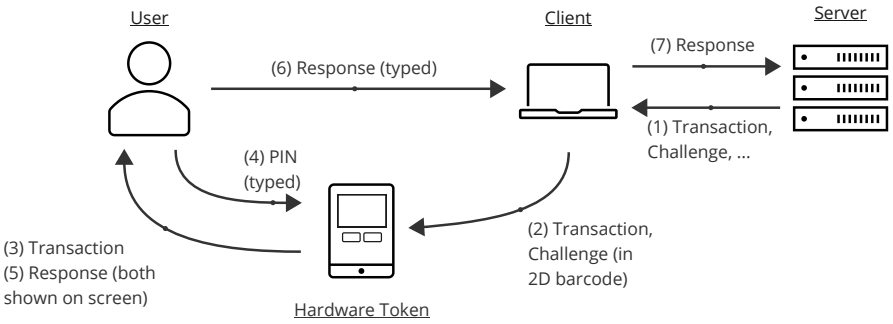

Fig. 1: Protocol participants and message flows

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 user 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 more queries to the model of Bellare *et al.* to make it suitable for two-factor authentication schemes. The added queries allow an adversary to learn either of the user'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.

- $\texttt{Execute}(U^i, S^j)$: this query captures **passive** attacks in which the adversary, $\mathcal{A}$, has access to the messages exchanged between U^i and S^j during the correct executions of a 2FA protocol, π.
- $\texttt{Reveal}(I)$: 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 $\mathcal{A}$.
- $\texttt{Test}(I)$: 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 $\mathcal{A}$.
- $\texttt{Send}(I, m)$: 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 $\texttt{Send}(U^i, \text{start})$ initialises π; when it is sent, $\mathcal{A}$ would receive the message that the user would send to the server.
- $\texttt{Corrupt}(I, a)$: this query models the adversary's capability to corrupt the involved parties.
  - if $I = U$: it can learn (only) one of the factors of U. Specifically,
    - if $a = 1$, it outputs U'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 U, even if it has access to the traffic between the two parties as well as having access to either U's PIN, or its hardware token, or (b) to impersonate S, even if 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 user 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 scheme's proof, we define a predicate $y \leftarrow \phi(m, \gamma)$, where $y \in \{0, 1\}$. This predicate takes as input a message m (*e.g.*, bank's transactions) and a policy γ (*e.g.*, a user'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

Recall that we wish to build an authentication protocol for which the server can verify that the PIN has been entered correctly but that an adversary cannot discover the correct PIN given access to challenge/response pairs and all data stored on the token, or access to all data stored on the server. These properties must be assured even when the PIN is small enough to be brute-forced. We achieve this goal through (a) performing the PIN verification only on the server, which imposes a rate limit on verification, (b) encrypting every sensitive message exchanged between the server and user using key-evolving symmetric-key

encryption (*i.e.*, a combination of forward-secure pseudorandom bit generator and authenticated encryption), and (c) protecting against server compromise, by never directly sending the PIN to the server. The confidentiality of PINs (e.g., in the case of the server breach) is crucial, as often people use their PINs for multiple purposes and in different places [23]. Thus, the leakage of PINs can have serious repercussions for people.

Our protocol consists of three phases: (i) a setup phase, performed once when the hardware token is manufactured, (ii) an enrolment phase for setting or changing a user'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. 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 user within a fixed time exceeds the threshold, then the user and its token will be locked out. Such a check is implicit in the protocol's description.

5.1 Setup Phase

To bootstrap the protocol, in the setup phase, we require that the user and server share an *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 token at the time of manufacture or can be sent (via a secure channel) to the user who can use the hardware token's camera to scan and store them in the token. In this phase, the token generates and locally stores a random secret key sa^U for PRF. Figure 2 presents the setup in detail.

	User's token		**Server**
1 :			$k \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
2 :			$st_0 \xleftarrow{\$} \mathsf{FS\text{-}PRG.KGen}(1^\lambda)$
3 :			$U_{\text{ID}} \leftarrow \mathsf{IDGen}(1^\lambda)$
		$\xleftarrow{(k,\, st_0,\, U_{\text{ID}})}$	
4 :	set k^U to k		set k^S to k
5 :	set st^U to st_0		set st^S to st_0
6 :	$ct^U \leftarrow 0$		$ct^S \leftarrow 0$
7 :	$sa^U \xleftarrow{\$} \{0,1\}^\psi$		
8 :	store$(U_{\text{ID}}, k^U, st^U, ct^U, sa^U)$		store $(U_{\text{ID}}, k^S, st^S, ct^S)$

Fig. 2: Setup phase.

5.2 Enrolment Phase

The goal of the enrolment phase is to set the user'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 user's selected PIN. The steps involved in this phase are detailed in Figure 3.

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. 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 passes it to the token. The token 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 counters from the message. To recover kt_1^S from the message the token's counter must be less than or equal to the counter it received from the server, which the protocol ensures is the case with a high probability (see Section 7 for more detail). Next, the token requests the PIN from the user and ensures it is what the user intends, *e.g.*, by requesting it twice and checking whether they match.

The token then generates a verifier v^U, by deriving a pseudorandom value from the PIN using PRF and the random key sa^U it generated in the setup phase. After that, the token 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^U. This synchronisation is possible because the check at line 10 has already assured that the token's state is behind the server's state by at least one step. After the update, kt_1^U 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 user's token then encrypts the verifier and challenge under kt_1^U and sends this to the server. On receiving and validating this message, the server decrypts the message using $kt_1{}^S$ and then extracts the challenge and verifier. If the received challenge does not match the challenge that the server sent, then the protocol halts. If the challenge does match, the server stores the verifier, v^S, associated with the user's account.

Finally, the token discards the challenge, kt_1^U, PIN, and v^U so that the PIN can no longer be recovered from the token. Note that the token can re-generate v^U using sa^U when the user types in their PIN again. The server also discards the challenge and $kt_1{}^U$ as they are no longer needed. Following the successful completion of this phase, the server stores the verifier corresponding to the user's PIN and the server and token 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 token is currently present, the correct PIN has been entered, and the user 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

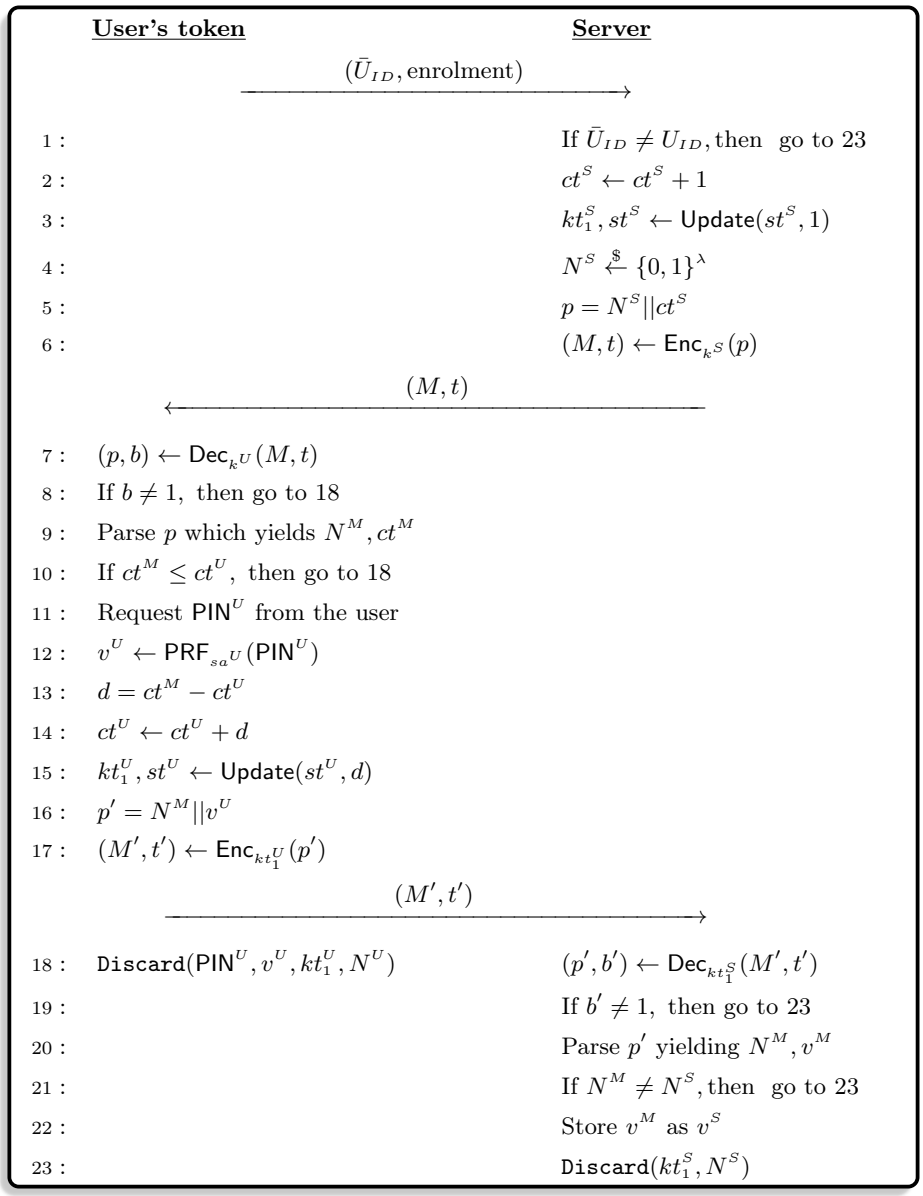

Fig. 3: Enrolment phase. For the sake of brevity, we have left out from the protocol's description the steps where the user inserts M' and t' into its computer. The $\bar{U}_{ID}$ and enrolment are two plaintext messages sent to the server.

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

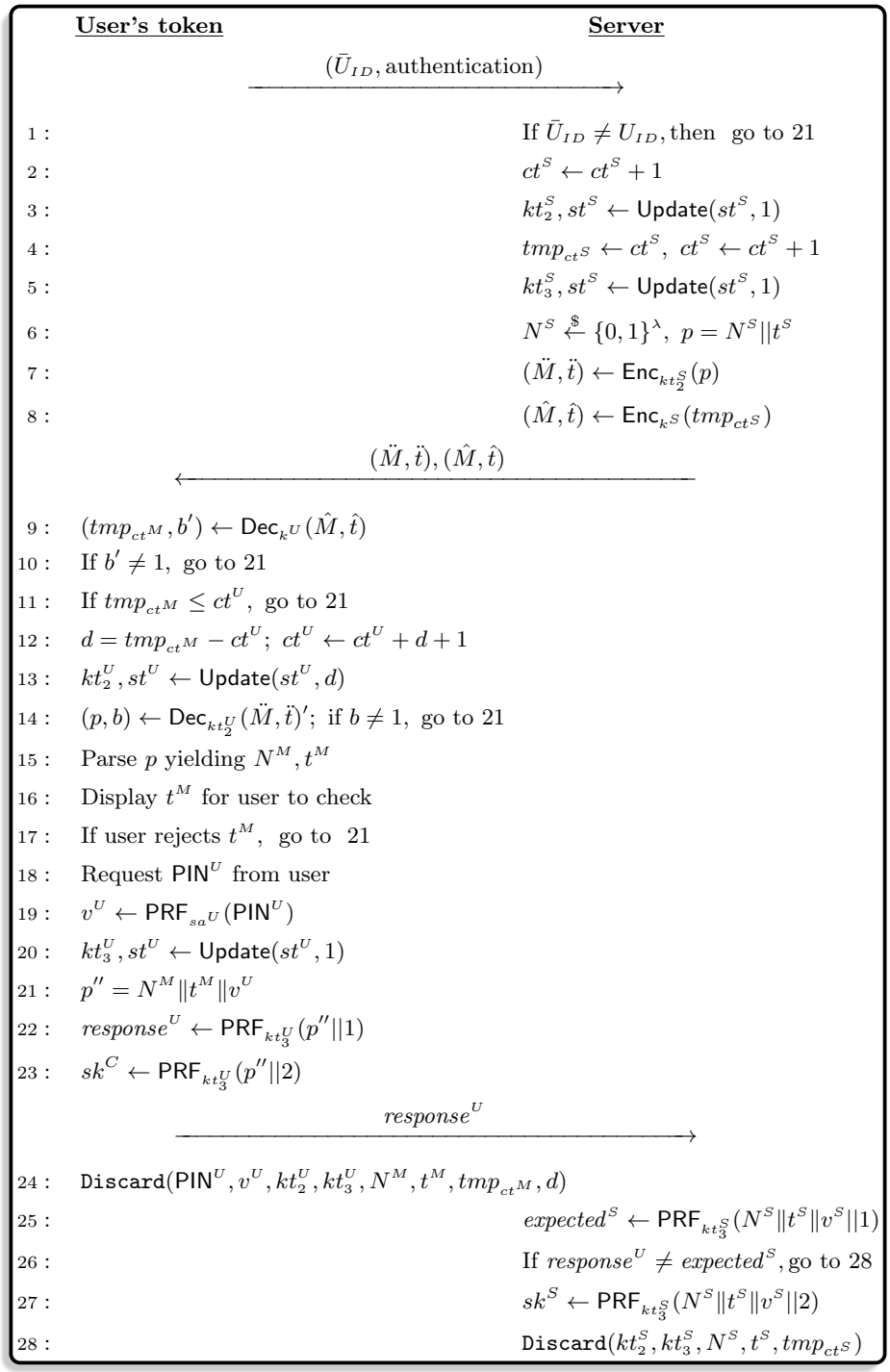

Fig. 4: Authentication phase. For the sake of brevity, we have left out from the protocol's description the step where the user inserts $response^U$ into its computer.

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^U. The reason tmp_{ct^S} is encrypted under key k^U is to allow the token to decrypt the ciphertext easily in case of previous messages were lost in transit; for instance, when the server sends $(\ddot{M}, \hat{M})$ to the server, but they were lost in transit, multiple times, and a fresh pair finally arrives at the client after the server sends them upon the user's request. Encrypting tmp_{ct^S} under key k^U (instead of one of the evolving keys) lets the token deal with such a situation.

Upon receiving the ciphertexts, the token 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 7, this check will succeed with high probability. The token synchronises its state and counter using the server's messages. Next, the token displays the transaction for the user to check. If the user does not accept the transaction (*e.g.*, due to an attempted man-in-the-browser attack), then the protocol halts immediately. Assuming the user is willing to proceed, then the token prompts for the PIN, and computes the verifier v^U using the key sa^U. If the user enters the correct PIN, the verifier will be the same as the one sent to the server during the enrolment phase.

For the token to generate the response message, it first updates its state one more time, which results in a pseudorandom value kt_3^U. Then, it derives a pseudorandom value, $response^U$, from a combination of the random challenge N^M, transaction t^M, verifier v^U, and $x = 1$ using PRF and kt_3^U. The token generates a session key, using the above combination and key with a difference that now $x = 2$. The response message is truncated to be a convenient length (e.g., 6–8 digits). It is displayed on the screen of the token. The user types it into the client which forwards it to the server. The token 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 that was set during the enrolment phase. The server then compares the expected response with the response sent by the client. Only if they match, the authentication is considered to have succeeded. If the response does not match the one the server expects, the server concludes that the message was tampered with or that the user entered an incorrect PIN. If the authentication succeeds, the server generates the session key in the same way as the token does. The server also discards the FS-PRG key, the challenge, and the transaction's description.

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 $Adv^{ss}_{\pi}(\mathcal{A})$ be $\mathcal{A}$'s advantage (in breaking the semantic security of an AKE scheme π) as defined in Section 4. If authenticated encryption Π, pseudorandom function PRF*, and forward-secure pseudorandom generator* FS-PRG *are secure, then $Adv^{ss}_{\pi}(\mathcal{A})$ for protocol ψ has the following upper bound:*

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

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 $Adv^{aut}_{\pi}(\mathcal{A})$ be $\mathcal{A}$'s advantage (in breaking the authentication of an AKE scheme π) as defined in Section 4. If authenticated encryption Π, pseudorandom function* PRF*, and forward-secure pseudorandom generator* FS-PRG *are secure, then in the protocol ψ, $Adv^{aut}_{\psi}(\mathcal{A})$ has the following upper bound:*

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

We refer readers to Appendix C for the formal security proof of the above two theorems.

5.4 An Extension

In this section, we outline a more enhanced version of the protocol. This enhanced protocol ensures that even if an adversary penetrates the server and accesses the server's local data, it cannot authenticate itself to the server later on (and as before it cannot learn users' PIN).

To deal with the adversary that breaks into the server and then tries to authenticate itself, we use the following idea (and additional steps). We require the token (at the setup phase) to generate and store a new key r of PRF. In the enrolment phase, the token derives a pseudorandom value w^U using the verifier v^U (generated in the original protocol) and r. In this phase, w^U is also securely sent to the server which stores it locally. Only the server needs to store w^U.

In the authentication phase, the client also securely[2] sends r to the server which re-generates w^U using r and the stored v^U and checks whether w^U equals the w^U that it locally maintains. It then deletes r.

Intuitively, the above approach guarantees security against the above adversary because having access to the local data of the server, the adversary still needs to know r to authenticate itself; since r has been picked uniformly at random and is not stored in the server, then its success probability depends on the

[2] For the token to securely send r to the server, the token (1) derives a fresh random value x from kt^U_3 using a key derivation function $\mathtt{Der}(.)$ that returns a random value of size $\log_2(r)$ bits and (2) sends $r' = x \oplus r$ to the server.

bit-length of r and if it is set to an appropriate size, then the probability will be negligible. We refer readers to Appendix B, for the full version of the enhanced protocol.

6 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.

6.1 Threats and Protection Objectives

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

- *T.DEV: token access.* An adversary may steal the authentication token. The adversary will then know k^U, sa^U and the current values of ct^U and st^U because we assume the token does not take advantage of a trusted chipset.
- *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 user, 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^U and st^U.

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

6.2 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 token but does know the user's PIN and communication between the client and server. For the adversary to perform a successful authentication, it must compute $\mathsf{PRF}_{kt_3^U}(N^M \| t^M \| v^U || 1)$. Nevertheless, it does not know kt_3^U or the

state from which kt_3^U has been generated; since kt_3^U is an output of PRF and is sufficiently large, it is computationally indistinguishable from a truly random value; thus, the probability of finding kt_3^U is negligible in the security parameter. The adversary may also try to guess the truncated response; however, its probability of success is at most $\frac{1}{\lambda}$, where λ is the bit-length of the truncated response.[3]

Thus, the only party which will generate a valid response is the token itself (when the PIN is provided) at line 22 of Figure 4. We have already assumed that the adversary does not have access to the token; therefore, it cannot generate a valid response.

2. *O.AUTH against T.DEV and T.MITM.* In this scenario, the adversary has compromised the user's token (but not its PIN), has records of previous messages, and wishes to impersonate the user. Note, in this case, the adversary does not learn v^U, t^M, PIN^U, kt_1^U, kt_2^U, kt_3^U or previous values of st^U; as these are all discarded when the protocol terminates, i.e., when `Discard`(.) is executed. 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 replay attack would not work. The adversary can use the token to discover 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 token 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 token and wishes to obtain the user's PIN. As with Scenario 2, the PIN cannot be obtained from the token, 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 user'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 token. 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 feasible 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 token were equal. Even this minimal information leakage can be removed if the server rejects the PINs that were used before.

[3] For instance, for 8-digit response its probability of success is $\frac{1}{2^{28}}$.

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 token. Furthermore, due to the security of the authenticated encryption, the token can detect (except for a negligible probability) if the transaction's description, that the server sends to it, has been tampered with.

6.3 Excluded Scenarios

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

- *Compromised PIN and token.* If the adversary has compromised both factors of a 2FA protocol, then the server cannot distinguish between the adversary and the legitimate user.
- *Authentication on compromised server.* 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 token.* If the adversary has compromised the server and the token, then the PIN can be trivially brute-forced with knowledge of v^S and sa^U.

7 Synchronisation

A user's token needs to be synchronised with the server in order for the server to check the correctness of the response generated by the token. 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 token's state, or that the server is ahead of the token, *i.e.*, $ct^S \geq ct^U$. Then, as challenge messages always contain the current value of the server's counter, the token 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 token never advances its state directly, but only advances to the point that the server currently is at. Thirdly, by requiring the token 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 token would not advance the state and would be behind the server. Nevertheless, this would not cause any issues, because the server's next challenge message will include the new value of the counter and the token will advance its state until it matches the server's state. Hence, in the authentication phase (in Figure 4), inequality $ct^S \geq ct^U$ must hold in the following three cases:

- **Case 1: no messages are dropped.** This is a trivial case and boils down to the correctness and security of the authentication protocol (presented in Figure 4). Specifically, in this case, inequality $ct^S \geq ct^U$ always holds because the token advances its state only after it receives a valid challenge from the server that has already advanced its state. An adversary would be able to convince the token to advance its state (before the server does so) if it could generate a valid encrypted challenge; however, its probability of success is negligible in the security parameter, due to the security of Authenticated Encryption (AE).
- **Case 2: the server's challenges are dropped.** If the adversary drops the server's encrypted challenge message $(\ddot{M}, \ddot{t})$, then the token would not advance its state. Because the token advances its state (at line 13) only after it receives the encrypted messages, checks their validity and makes sure its state is smaller than the server's state. The server advanced its state, regardless of whether its message will be dropped. Thus, inequality $ct^S \geq ct^U$ always holds in Case 2 as well.
- **Case 3: the user responses are dropped.** If the adversary drops the user's response (*i.e.* $response^U$), then the user cannot authenticate itself to the server, but it can re-execute the authentication protocol, by sending to the server message $(U_{ID}, \text{authentication})$ again. In Case 3, both the token and server have updated their state before the message is dropped; therefore, inequality $ct^S \geq ct^U$ holds in this case too.

Note that the FS-PRG advance process is fast; thus, multiple invocations of this will not create a noticeable delay. In the case where the enrolment's response message is dropped, the PIN will remain unchanged; as a result, the user 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.

8 System Feasibility

Usability is of critical importance for an effective authentication system as otherwise, users will refuse to use it or implement insecure workarounds [9]. As we stated in Section 4, in our protocol, the server interacts with the token via the client. To accommodate usability and let the token receive the server's message, we require the token to be able to receive a few hundred bytes from the server. This functionality is already present on any token capable of transaction authentication because it must be able to receive a description of the transaction to show to the user. Typically this communication functionality is implemented by an inexpensive camera such as in the Gemalto SWYS QR [30] or OneSpan Digipass 770 [24], which both scan a 2D barcode shown on the screen of the client. The response from the token to the server can be safely truncated because the protocol ensures offline brute-force attacks are not possible. So, the response can be manually typed without any special hardware required for this direction of communication. The response length should be selected to reduce the chance of success of an online brute force attack to an acceptable level, taking into consideration the rate-limiting implemented on the server. This security-usability

trade-off is not specific to our protocol and exists in all hardware token-based multi-factor authentication schemes that do not assume a high-bandwidth communication channel from the authentication token to the server.

Another consideration is handling mistyped or forgotten PINs. As we highlighted in Section 5, when setting the PIN, the token can ask the user to confirm their PIN by entering the PIN twice and alerting the user if they do not match. However, because we assume that the token has no trusted hardware we cannot store the PIN in the token. Therefore, during authentication, if the wrong PIN is entered, the user will only be alerted after the response code has been verified by the server. To enhance usability by detecting mistyped PINs earlier in the protocol, at some cost of security, the token could show the user an image computed as a function of the PIN entered to help the user detect a mistyped PIN, effectively serving as a checksum. To prevent someone observing the token from discovering the PIN from the image, the function could be designed to have a large number of collisions. When the PIN is totally forgotten, then users need to prove their identity (e.g., by providing their identity card) to the server which would allow them to enrol again, if the server approves their identity.

9 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], the hardware token-based protocol in [15], and the symmetric-key-based scheme in [22]. We consider the protocols in [33,15] because they are relatively efficient, do not rely on secure hardware, and consider the same security threats as we do, *i.e.*, resistance against card/token loss, against an offline attack, and against a corrupt server. We also include the scheme in [22] in our analysis because it is highly efficient, only uses symmetric-key primitives, and is in the standard model.

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

Features	Operation	Our Protocol	[33]	[15]	[22]
Computation cost	Sym-key	18	19	7	9
	Modular expo.	0	5	12	0
Communication cost	–	2804-bit	3136-bit	3900-bit	896-bit
Not requiring multiple PINs	–	✓	×	✓	✓
Not requiring modular expo.	–	✓	×	×	✓
Not requiring trusted terminal or chipset	–	✓	×	×	×
Secure Against Corrupt Server or Token	–	✓	✓	✓	×
Security assumption	–	Standard	Random oracle	Random oracle	Standard

9.1 Computation Cost

We start by analysing our protocol's computation cost. First, we focus on the protocol's enrolment phase. The user'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 user 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 user-side operations. The enrolment phase involves 3 and 2 invocations of a hash function at the user and server sides respectively. This protocol has an additional phase called login which costs the user 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 user to perform 1 modular exponentiation and invoke the hash function 3 times.

Now, we analyse the computation cost of the protocol presented in [15]. 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 user to perform a single modular exponentiation and invoke a hash function 2 times. It also involves, as a subroutine, the initialisation of the asymmetric "password-authenticated key exchange" (PAKE) proposed in [12], which involves at least 2 modular exponentiations, 1 invocation of hash function and symmetric-key encryption. In the login phase, the user 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).

Now, we focus on the protocol proposed in [22]. As we previously mentioned the scheme is highly efficient. The scheme assumes the user and server have already agreed on the user's password. In total, the user (and the trusted chipset) invokes the verification algorithm of a Message Authentication Code (MAC) scheme once and the pseudorandom function three times. The server cost is similar to the user's cost, with the difference that the server also invokes the tag generator algorithm of the MAC scheme once. The scheme does not involve any modular exponentiations.

Thus, our protocol and the ones in [33,15,22] involve a constant number of symmetric-key primitive invocations; however, our protocol and the protocol in [22] do not involve any modular exponentiations, whereas the protocol in [33,15] involves a constant number of modular exponentiations which leads to a higher cost.

9.2 Communication Cost

We next analyse our protocol's communication cost. In the enrolment phase, the user only sends two pairs of messages: (U_{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 user only sends three messages: (U_{ID}, authentication, $response^U$), 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 user's total communication cost in the enrolment and login phases is 1792 bits. Note that we set the user'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 user sends to the server a single value of size 160 bits. In the verification, the server sends to the user 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 [15]. As before, in our cost evaluation, we estimate the protocol's minimum cost. In the enrolment phase, a user 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 user 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 user to send out at least one signature of size 1024 bits. The device sends to the user a ciphertext of asymmetric-key encryption which is of size 1024 bits along with a 20-bit message. Thus, the user-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.

Next, we focus on the communication cost of the protocol in [22]. In total, the user for each authentication sends $(r_A, ID_A, Auth_A)$ to the server. The server also sends $(r_A, r_B, ID_B, Auth_B)$ to the user, where each message is of size 128-bit. Thus, the protocol's total communication cost is 896 bits.

Hence, our protocol imposes a 10% and 40% lower communication cost than the protocols in [33] and [15] that are secure against a corrupted token or server. The protocol in [22] has the lowest communication cost but it is not secure against a corrupted server.

9.3 Other Features

In our protocol, a user needs to know and type into the token only a single secret, *i.e.*, a PIN. Furthermore, we do not put any trust assumptions on users' personal computers (or clients). Therefore, they can be corrupted at any time, e.g., simultaneously when another party such as the token or server is corrupted.

In contrast, in the protocol in [33], a user has to know and type an additional secret, *i.e.*, a random ID. As shown by Scott [29], this scheme will not remain

secure, even if only the user's ID is revealed. Also, in this scheme, users type in their PINs into another device, i.e., a card reader. For the security of the scheme holds, it is required that the card reader be fully trusted, in the case where the server or smart card is breached.

Similarly, the protocol in [15] requires users to insert their PINs into their personal computers instead of inserting them into the dedicated hardware token; this approach is problematic as the PINs will be at a higher risk of being exposed to attackers because users' computers are (i) often connected to the Internet, and (ii) used for various purposes, consequently, they are more likely to be broken into. Furthermore, the protocol relies on an additional trusted party as well, i.e., trusted users' computers. Specifically, for the protocol's security to hold, it is required that users' personal computers are fully trusted, in the case where the token or the server is corrupted, which is not desirable. In contrast, our protocol requires users to insert their PIN into the hardware token that is used only for authentication and is always disconnected from the Internet.

In the protocol in [22] a user needs to know only a single PIN. It requires the server to be fully trusted and it uses a trusted chipset. So, it relies on the strongest security assumption. Our protocol and the one in [22] are secure in the standard model whereas the protocols in [33,15] are in the non-standard random oracle model.

10 Conclusion and Future Work

We have presented a 2FA protocol that resists a strong adversary who may (a) observe the traffic between a user and server, and (b) have physical access to the user's hardware token, 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 user to remember only one secret/PIN, (ii) is based on only symmetric-key primitives, (iii) is in a standard model, (iv) does not assume the user's computer is trusted, and (v) imposes low communication costs. This is the first protocol that offers the aforementioned features without requiring tamper-resistant hardware. Future research could investigate the usability of a hardware token that embeds our 2FA protocol.

References

1. Payment Services (PSD 2). Directive 2015/2366/EU of the European Parliament and of the Council (November 2015)
2. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: EUROCRYPT (2000)
3. Bellare, M., Yee, B.S.: Forward-security in private-key cryptography. In: CT-RSA (2003)
4. Biryukov, A., Lano, J., Preneel, B.: Cryptanalysis of the alleged SecurID hash function. In: International Workshop on Selected Areas in Cryptography (2003)
5. Bresson, E., Chevassut, O., Pointcheval, D.: Security proofs for an efficient password-based key exchange. In: ACM CCS (2003)
6. Chang, C.C., Wu, T.C.: Remote password authentication with smart cards. IEE Computers and Digital Techniques (1991)

7. 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)
8. 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)
9. De Cristofaro, E., Du, H., Freudiger, J., Norcie, G.: A comparative usability study of two-factor authentication. arXiv preprint arXiv:1309.5344 (2013)
10. Executive Office of the President: Office of Management and 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
11. Feng, H., Li, H., Pan, X., Zhao, Z.: A formal analysis of the FIDO UAF protocol. In: NDSS. The Internet Society (2021)
12. Gentry, C., MacKenzie, P.D., Ramzan, Z.: A method for making password-based key exchange resilient to server compromise. In: CRYPTO (2006)
13. 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)
14. Imran, A., Farrukh, H., Ibrahim, M., Celik, Z.B., Bianchi, A.: SARA: Secure android remote authorization. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 1561–1578. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/imran
15. 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)
16. 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
17. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. CRC Press (2014)
18. Kim, S.K., Chung, M.G.: More secure remote user authentication scheme. Computer Communications (2009)
19. Kogan, D., Manohar, N., Boneh, D.: T/Key: Second-factor authentication from secure hash chains. In: CCS. ACM (2017)
20. 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)
21. Loutfi, I., Jøsang, A.: FIDO trust requirements. In: NordSec 2015 (2015)
22. Matsuo, S., Moriyama, D., Yung, M.: Multifactor authenticated key renewal. In: Trusted Systems - Third International Conference, INTRUST 2011,. Lecture Notes in Computer Science, Springer (2011)
23. Murdoch, S.J., Becker, I., Abu-Salma, R., Anderson, R.J., Bohm, N., Hutchings, A., Sasse, M.A., Stringhini, G.: Are payment card contracts unfair? (short paper). In: Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers. Lecture Notes in Computer Science (2016)
24. OneSpan: Digipass 770 datasheet, https://www.onespan.com/resources/digipass-770/datasheet
25. 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)

26. 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)
27. Reardon, J., Basin, D.A., Capkun, S.: Sok: Secure data deletion. In: IEEE Symposium on Security and Privacy, SP 2013. IEEE Computer Society (2013)
28. 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
29. Scott, M.: Cryptanalysis of a recent two factor authentication scheme. IACR Cryptol. ePrint Arch. p. 527 (2012)
30. Thales: Gemalto SWYS QR Reader Eco (2020), https://www.thalesgroup.com/sites/default/files/database/document/2020-12/fs-QR-code-reader.pdf
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 Definition and Construction of Forward-Secure Pseudorandom Bit Generator and Keyed Pseudorandom Function

In this section, we restate the formal definition of the forward-secure pseudorandom bit generator (taken from [3]), briefly explain how it can be constructed, and also define a keyed pseudorandom function PRF. 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, 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.

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

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

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

$st_0 \xleftarrow{\$} \mathsf{FS\text{-}PRG.KGen}(1^\lambda)$
$i \leftarrow 0; h \leftarrow \text{aux}$
Repeat
$i \leftarrow i + 1$
$(out_i, st_i) \leftarrow \mathsf{FS\text{-}PRG.next}(st_{i-1})$
$out_i \xleftarrow{\$} \{0,1\}$
$(I, h) \leftarrow \mathcal{A}(\text{find}, out_i, h)$
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.* [3] 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 [3] for further discussion.

Definition 1. *Let* $\mathsf{PRF} : \{0,1\}^\psi \times \{0,1\}^\eta \rightarrow \{0,1\}^\lambda$ *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 [17].

B More Enhanced Protocol

In this section, we present a more enhanced version of the protocol that we presented in Section 5. The proposed protocol ensures that even if an adversary penetrates the server and accesses the server's local data, it cannot authenticate itself to the server later on and as before it cannot learn users' PIN.

We present the new versions of the setup, enrolment and authentication phases in Figures 5, 6, and 7. We have highlighted the new steps in blue.

	User's token		**Server**
1 :			$k \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
2 :			$st_0 \xleftarrow{\$} \mathsf{FS\text{-}PRG.KGen}(1^\lambda)$
3 :			$U_{\text{ID}} \leftarrow \mathsf{IDGen}(1^\lambda)$
		$(k,\, st_0,\, U_{\text{ID}})$ $\longleftarrow$	
4 :	set k^U to k		set k^S to k
5 :	set st^U to st_0		set st^S to st_0
6 :	$ct^U \leftarrow 0$		$ct^S \leftarrow 0$
7 :	$sa^U \xleftarrow{\$} \{0,1\}^\psi$		
8 :	$r \xleftarrow{\$} \{0,1\}^\iota$		
9 :	store$(U_{\text{ID}}, k^U, st^U, ct^U, sa^U, r)$		store $(U_{\text{ID}}, k^S, st^S, ct^S)$

Fig. 5: Setup phase.

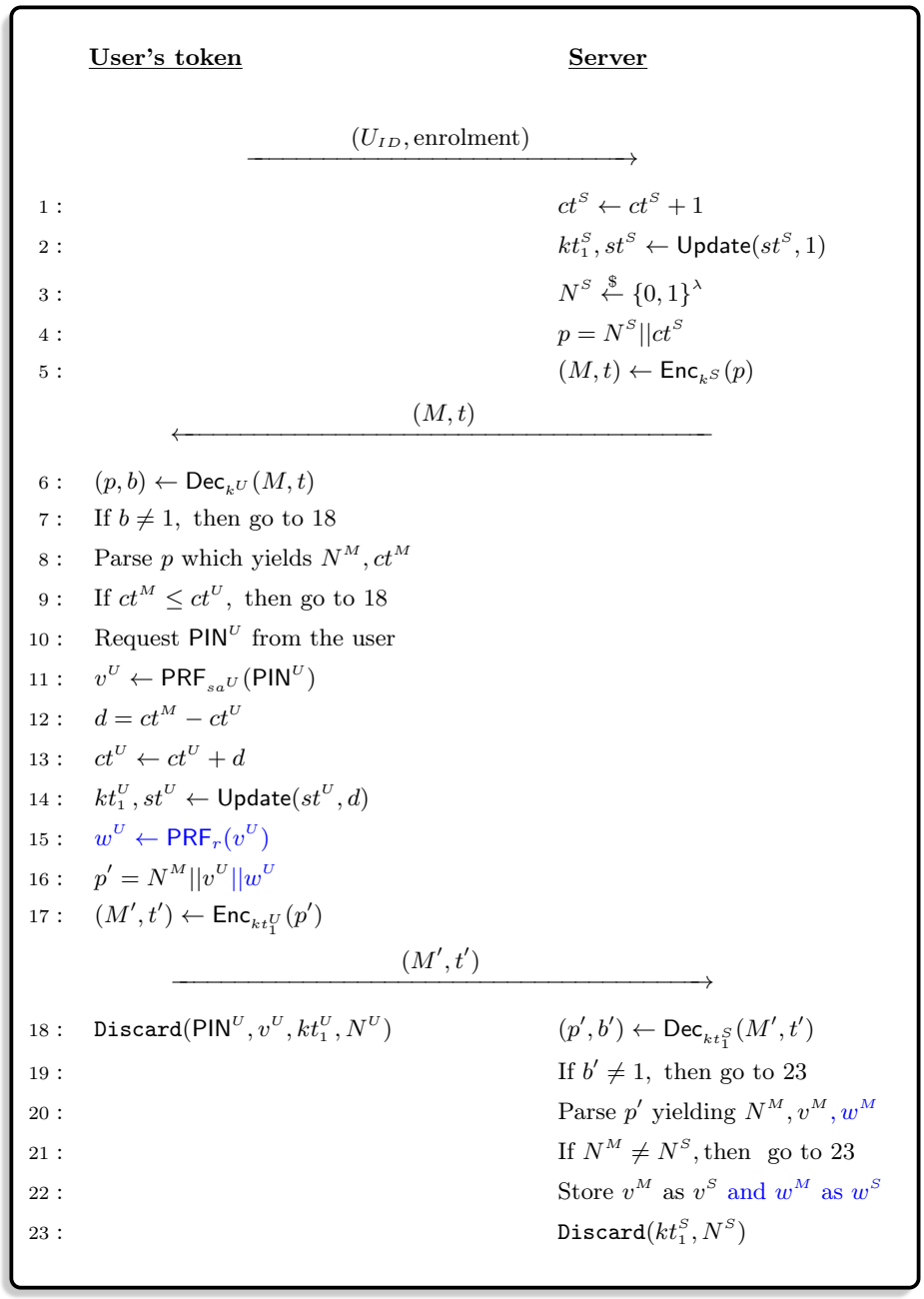

Fig. 6: Enrolment phase.

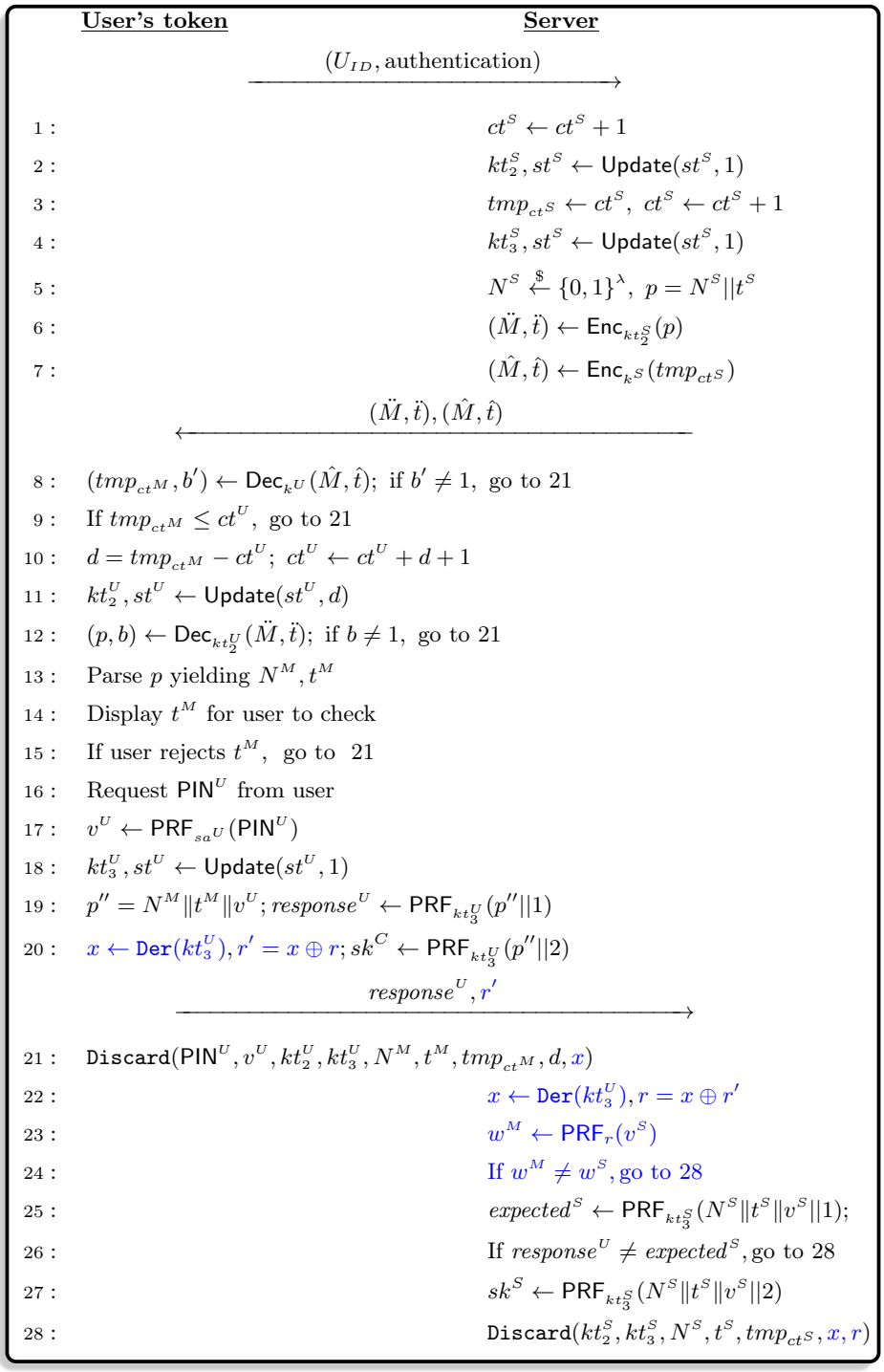

Fig. 7: Authentication phase.

C 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.

C.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 (`Enc` and `Dec`) and all instances U^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 [5], 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 `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 8 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 the user's and server's `Send` queries in Figures 9 and 10 respectively. Also, we present the simulators for the rest of the queries in Figure 11. By definition, ϕ is a deterministic function, given the transaction t^U and policy γ, it always returns the same output as the user's token does when verifying t^U in the previous game. Therefore, both ϕ and the user's token would output identical values, given pair (t^U, 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 $\texttt{Send}(S^j, \bar{response}^U)$ as follows:
  1. computes $expected^S \leftarrow \mathsf{PRF}_{\bar{kt}_3^S}(\ddot{N}^S \| t^S \| v^S || 1)$.
  2. checks if $\bar{response}^U = expected^S$. It proceeds to the next step if the equation holds.
  3. checks if $\Big((U_{ID}, \text{enrolment}), (U_{ID}, \text{authentication}), (\bar{M}, \bar{t}),\ (\bar{M}',\ \bar{t}'),\ (\ddot{M}',\ \ddot{t}'),\ (\hat{M}', \hat{t}'), \bar{response}^U\Big) \in \vec{L}$.
  4. checks if $\bar{response}^U \in L_{\mathcal{A}}$.
  5. if both checks in steps 3 and 4 fail, then it rejects authenticator $\bar{response}^U$ 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 $\mathtt{Send}(S^j, \bar{response}^U)$, 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 $\bar{response}^U = expected^S$. It proceeds to the next step if the equation holds.
  3. checks if $\Big((U_{ID}, \text{enrolment}), (U_{ID}, \text{authentication}), (\bar{M}, \bar{t}),\ (\bar{M}',\ \bar{t}'),\ (\ddot{M}',\ \ddot{t}'),\ (\hat{M}', \hat{t}'), \bar{response}^U\Big) \in \vec{L}$. If this check fails, then it rejects authenticator $\bar{response}^U$ and terminates, without accepting any key.
  4. checks if $(\ddot{N}^S \| t^S \| *, \bar{response}^U) \in L_{\mathcal{A}}$.
  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 $\bar{response}^U$. 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 user's token processes query $\mathtt{Send}(U^i, (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'))$ as follows.
  1. compute $\bar{response}^U \leftarrow \mathsf{PRF}_{\bar{t}'}(\ddot{M}' || 1)$.
  2. compute $\bar{sk}^U \leftarrow \mathsf{PRF}_{\bar{t}'}(\ddot{M}' || 2)$.

  We also amend the way the server compiles query $\mathtt{Send}(S^j, \bar{response}^U)$ as follows.
  a checks if $(\ddot{M}' || 1, \bar{response}^U) \in L_{\mathcal{A}}$ or $(\ddot{M}' || 2, \bar{sk}^U) \in L_{\mathcal{A}}$.
  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 $\bar{response}^U$ or $q = \ddot{M}'||2$ to PRF results in $\bar{sk}^U$. 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. 8: Pseudorandom function's simulator.

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

This query is dealt with as below:

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

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

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

• upon receiving $\underline{\texttt{Send}(U^i, (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'))}$, it:
  1. authenticates and decrypts the ciphertext $\hat{M}'$ as follows: $(p', b') \leftarrow \mathsf{Dec}_{k^U}(\hat{M}', \hat{t}')$. If the authentication fails (i.e., $b' \neq 1$), it halts.
  2. extracts (tmp_{ct^M}, ct^M) from p' and checks if $tmp_{ct^M} > ct^U$. If the check fails, it halts.
  3. updates its state as follows: $\forall i, 1 \leq i \leq tmp_{ct^M} - ct^U$: (a) $ct^U \leftarrow ct^U + 1$ and (b) $(k, st^U) \leftarrow \mathsf{Update}(st^U, ct^U)$.
  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, $y \leftarrow \phi(t^M, \gamma)$. If $y = 0$, it halts.
  7. generates v^U using sa^U and PIN^U as follows, $v^U \leftarrow \mathsf{PRF}_{sa^U}(\mathsf{PIN}^U)$.
  8. updates its state one more time as follows, $(k', st^U) \leftarrow \mathsf{Update}(st^U, ct^M)$.
  9. computes the authenticator: $\bar{response}^U \leftarrow \mathsf{PRF}_{k'}(N^M || t^M || v^U || 1)$ and session key: $\bar{sk}^U \leftarrow \mathsf{PRF}_{k'}(N^M || t^M || v^U || 2)$.
  10. responds the send query with $\bar{response}^U$. It makes the user'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 $\Big((U_{ID}, \text{enrolment}), (U_{ID}, \text{authentication}), (\bar{M}, \bar{t}), (\bar{M}', \bar{t}'), (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'), \bar{response}^U\Big) \in \vec{L}$.

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

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

This query is dealt with as below:

- upon receiving $\underline{\texttt{Send}(S^j, (U_{ID}, \text{enrolment}))}$, it:
  1. increments its counter as $ct^S \leftarrow ct^S + 1$, updates its state as $\bar{kt}_1^S, st^S \leftarrow \mathsf{Update}(st^S, ct^S)$, 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 $(p', b') \leftarrow \mathsf{Dec}_{\bar{kt}_1^S}(\bar{M}', \bar{t}')$. 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 $\underline{\texttt{Send}(S^j, (U_{ID}, \text{authentication}))}$, it:
  1. increments its counter $ct^S \leftarrow ct^S + 1$, updates its state $\bar{kt}_2^S, st^S \leftarrow \mathsf{Update}(st^S, ct^S)$, temporarily stores this counter $tmp_{ct^S} \leftarrow ct^S$, increments the counter again $ct^S \leftarrow ct^S + 1$, updates its state again $\bar{kt}_3^S, st^S \leftarrow \mathsf{Update}(st^S, ct^S)$, and picks a random value $\ddot{N}^S \xleftarrow{\$} \{0,1\}^\lambda$.
  2. generates two pairs of ciphertext and tag as follows, $(\ddot{M}', \ddot{t}') \leftarrow \mathsf{Enc}_{\bar{kt}_2^S}(\ddot{N}^S || t^S)$ 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 $\underline{\texttt{Send}(S^j, \bar{response}^U)}$, it:
  1. computes $expected^S \leftarrow \mathsf{PRF}_{\bar{kt}_3^S}(\ddot{N}^S \| t^S \| v^S || 1)$. It checks whether $\bar{response}^U = expected^S$. If the equality does not hold, the server instance terminates without accepting any session key.
  2. generates the session key $\bar{sk}^S \leftarrow \mathsf{PRF}_{\bar{kt}_3^S}(\ddot{N}^S \| t^S \| v^S || 2)$. It accepts the key and terminates.

Fig. 10: Simulators for Send query to a server's instance.

$\texttt{Execute}(U^i, S^j)$

This query is dealt with as below:

1. $(U_{ID}, \text{enrolment}) \leftarrow \texttt{Send}(U^i, \text{start}, \text{enrolment})$.
2. $(\bar{M}, \bar{t}) \leftarrow \texttt{Send}(S^j, (U_{ID}, \text{enrolment}))$.
3. $(\bar{M}', \bar{t}') \leftarrow \texttt{Send}(U^i, (\bar{M}, \bar{t}))$.
4. $(U_{ID}, \text{authentication}) \leftarrow \texttt{Send}(U^i, \text{start}, \text{authentication})$.
5. $(\ddot{M}', \ddot{t}', \hat{M}', \hat{t}') \leftarrow \texttt{Send}(S^j, (U_{ID}, \text{authentication}))$.
6. $\bar{response}^U \leftarrow \texttt{Send}(U^i, (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'))$.
7. outputs the following transcript: $[(U_{ID}, \text{enrolment}), (\bar{M}, \bar{t}), (\bar{M}', \bar{t}'), (U_{ID}, \text{authentication}), (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'), \bar{response}^U]$.

$\texttt{Reveal}(I)$

This query is processed as follows.
• returns session key $\bar{sk}^I$ (computed by $I \in \{U, S\}$), if I has already accepted the key.

$\texttt{Test}(I)$

This query is processed as below.

1. $sk \leftarrow \texttt{Reveal}(I)$.
2. $b \xleftarrow{\$} \{0,1\}$.
3. sets v as follows:
$$v = \begin{cases} sk, & \text{if } b = 1 \\ r \xleftarrow{\$} \{0,1\}^\iota, & \text{otherwise} \end{cases}$$
4. returns v.

Fig. 11: Simulators for `Execute`, `Reveal`, and `Test` queries.

C.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 user, U; we denote such a case with $a\bar{ut}$. The Authentication proof relies on the semantic security proof (and games) we presented in Section C.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:

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

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-key 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^U. [4] Thus, when we take the forward security into account, the advantage of the adversary (due to the union bound) is as follows:

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

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

[4] The case where $\mathcal{A}$ has the additional capability to send query $\texttt{Corrupt}(U, 2)$ was never discussed and analysed in [5]. However, we noticed that $\mathcal{A}$ in that scheme would have the same upper bound advantage as $\mathcal{A}$ in our scheme does.