A Robust Efficient Two-factor Authentication Protocol

ABSTRACT

Two-factor authentication (2FA) is a widely adopted method for securing access to high-value online accounts, especially in scenarios involving financial transactions. In these setups, successful authentication necessitates the presence of knowledge factors (such as a PIN) and the possession of a hardware token. Nonetheless, the existing 2FA protocols expose the PIN to vulnerability in the event of a compromise to the server, user's computer, or hardware token.

This paper presents a robust 2FA protocol designed to withstand attacks from adversaries who can (a) intercept the communication between a user and the server and (b) gain physical access to the user's hardware token, its web browser, its associated PIN, or even breach the server. In comparison to prior methodologies, our protocol reduces the development cost of authentication tokens by eliminating the need for tamper-resistant hardware or direct hardware token linkage to the computer. Moreover, our protocol reduces communication overhead by up to 40% compared to state-of-the-art solutions, all while using efficient symmetric-key primitives.

CCS CONCEPTS

• **Security and privacy → Cryptography**.

KEYWORDS

Multi-factor authentication, forward security

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 (PSD2) [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. Nonetheless, current systems cannot meet these requirements.

2FA requires the combination of two factors: knowledge (like a PIN or password), possession (of a hardware device/token) or biometrics (such as a fingerprint). Dedicated 2FA tokens equipped with biometric sensors incur higher costs compared to their counterparts without such sensors. The integration of biometric authentication necessitates specialized hardware, including fingerprint scanners, iris scanners, or facial recognition cameras, contributing to the increased expense associated with these tokens. These sensors must be of high quality to ensure accurate and reliable readings, ultimately leading to an increased price for biometric-based tokens. Furthermore, the creation of accurate and secure biometric algorithms demands substantial development efforts. These algorithms must reliably identify individuals while resisting various forms of spoofing. Biometrics possess other drawbacks too, including irrevocability, limited applicability, and privacy issues. Thus, this paper concentrates on merging knowledge and possession factors.

Within this paper, we introduce a novel 2FA protocol that not only addresses the shortcomings of existing systems but also greatly enhances them. Consequently, this protocol holds the potential to significantly advance the wider adoption of 2FA, particularly in authenticating online access and authorizing *financial transactions*. Our primary criteria for the 2FA protocol are outlined as follows:

- **Security**: The protocol must exhibit robustness in thwarting unauthorized access across a wide range of scenarios, including compromises to the server, hardware token, user's computer (including its **web browser**), and communication network.
- **Usability**: The protocol must minimize user and environmental disruptions, eliminating the need for specialized software installation on user computers and relieving users from the task of memorizing lengthy passwords or inputting extensive strings on computers or hardware tokens.
- **Cost-efficiency:** The affordability of the hardware token holds paramount importance, effectively obviating the requirement for tamper-resistant trusted hardware or complex computational capabilities.

Also, 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 the ability to check the transaction details (e.g., the amount of transfer and the details of the recipient in the context of online banking) and so be able to detect if malware on their computer has tampered with the transaction details shown on screen.

Researchers and companies have put forth diverse 2FA solutions that rely on the combination of PIN and token possession. Certain solutions within this spectrum provide strong security assurances against adversaries capable of (a) monitoring the communication between a user and server, and (b) gaining physical access to the user's token, its associated PIN, or even breaching the server.

However, these solutions exhibit a subset of the following drawbacks: (i) demanding users to recall multiple secret values (instead of a PIN) for identity verification, thereby undermining usability, (ii) requiring users to entrust their personal computers where they input their PINs, (iii) using a trusted chipset, (iv) entailing numerous exponentiations that deplete the token's battery swiftly, or (v) being proven secure only in the non-standard random oracle model.

Our Contributions. In this work, we introduce a 2FA protocol that resists the strong adversary outlined earlier, concurrently rectifying the aforementioned limitations and reducing communication overhead. To elucidate further, our protocol:

- requires a user to memorize and input a singular PIN into tokens.
- enables tokens to produce a succinct authentication message.
- eliminates the need for any modular exponentiations.
- is proved secure in a standard model.
- refrains from placing any security assumptions on users' personal computers.

- imposes communication costs that are up to 40% lower compared to prevailing state-of-the-art protocols designed to withstand the described adversary.

To achieve its objectives, our protocol refrains from employing any trusted chipset; instead, it relies on a novel combination of the following three approaches.

Firstly, neither the server nor the hardware token can 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, the protocol mandates that both the server and token utilize key-evolving symmetric-key encryption, which constitutes the blend of forward-secure pseudorandom bit generation and authenticated encryption.

Thirdly, our protocol, for the first time, harnesses the intrinsic data-deletion capability of tokens. This necessitates the immediate discarding of used keys post-utilization.

The combination of the aforementioned approaches ensures the secrecy of the communication between the parties and guarantees that adversaries cannot ascertain the PIN, even if they intercept the parties' communication and subsequently breach either the token or the server. We formally prove the security of this protocol.

2 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 mechanisms that rely 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:

1. *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 need to ensure the random challenges themselves remain confidential in the presence of an eavesdropping adversary.
2. *an internal counter*: the solutions that use this approach need to take into account the situation where the token-side counter becomes out of synchronization.
3. *the current accurate time*: this approach requires the authentication server and token to use a synchronized clock and the two endpoints may get out of synchronization 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

2.2.1 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. Yet, there exists no single widely deployed 2FA that everyone in the industry uses. 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 into 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 through manual analysis, such as those in [7, 21, 25] or through formal evaluation, as shown [11].

Additionally, for these devices to function, corresponding software must be installed on the computer. This requirement may not be feasible for shared devices or computers that adhere to a corporate IT policy. If any specialized software is required, it must be developed and implemented for each supported operating system and processor architecture. This entails periodic updates, leading to increased development and maintenance 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, 27, 34]) along with a few works that identify vulnerabilities of existing solutions, *e.g.*, in [8, 32, 33].

However, the existing smart card-based solutions (*e.g.*, in [13, 27, 34]) 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.

2.2.2 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.

Category 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 does the token have to have a synchronized 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

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

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 (as a result it does not deal with offline dictionary attacks).

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 mandates users to input their passwords/PINs into their own (personal) computers (referred to as client C in the paper) instead of entering them into the designated hardware token. This poses a concern since the PINs become more vulnerable to exposure by attackers. This is because users' computers are nearly always connected to the Internet, serve multiple functions, and are more susceptible to unauthorized access, (4) to maintain the security of the protocol, it is necessary for users' personal computers to be fully trusted, especially in situations where the token or the server is compromised. However, this introduces an additional assumption that may not always be desirable.

Moreover, an alternative authentication protocol, presented in [35], operates independently of a trusted chipset. However, it has been specifically tailored for "federated identity systems" and thus is not well-suited for two/multi-factor authentication scenarios.

Category 2: Mobile Phone-based Tokens, exemplified by the solutions introduced in [14, 19, 20]. Protocols have been developed where a mobile phone serves as a hardware token to generate a One-Time Password (OTP). These solutions frequently leverage the additional capabilities inherent in mobile phones, including features like a Trusted Execution Environment (TEE), direct server communication, and 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.

Mobile phone-based OTP tokens have limitations: they require network coverage, may not be suitable for all users due to privacy concerns when sharing phone numbers, and not everyone owns a smartphone or is willing to install special software on it. Moreover, smartphones have a large attack surface that can be exploited to extract authentication secrets.

For major banks, it is often not commercially viable to refuse service to customers unable or unwilling to use smartphone-based authentication, and in countries that impose a universal service obligation on certain banks (e.g., the United Kingdom) it would not be legal to do so. Therefore, it is established practice for banks to offer dedicated hardware-based authentication tokens, at least to customers who request these.

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, and 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 also assume that a user will not use its hardware token after it has been stolen. All 2FA schemes that depend on hardware tokens "implicitly" rest on this assumption. Without it, an attacker could steal a token, extract its content, modify the token to retain the user's PIN, and return it to the user.

Subsequently, when the user employs the compromised token and enters their PIN, the adversary regains access to the token and obtains both authentication factors. Table 1 presents a summary of notations used in this paper.

Table 1: Notation Table.

Symbol	Description
$\mathsf{PRF}_k(.)$	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. Initialized 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 st_0 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 synchronizing FS-PRG state and detecting replayed messages. Initialized 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. Initialized 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.

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 with two algorithms and two positive integers; namely, $\mathsf{FS\text{-}PRG} = \Big((\mathsf{FS\text{-}PRG.KGen}, \mathsf{FS\text{-}PRG.next}), (b, n)\Big)$, as defined in [3]. The probabilistic key generation algorithm FS-PRG.KGen takes a security parameter as 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. Appendix A restates a formal definition and construction of FS-PRG.

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 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 MODEL AND SYSTEM DESIGN

A 2FA scheme involves two players, (1) User (U): an honest party that 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 enable server-to-hardware token communication through the user's computer (the client), assuming the token has a camera for scanning 2-D barcodes containing messages sent by the server via the client, similar to previous works such the ones in [15, 24, 30]. In this work, we denote instances of user and server by U^i and S^j respectively. Each instance is called an oracle. Multiple instances of these parties may run concurrently.

Figure 1 outlines the message flow of our 2FA scheme during the authentication phase. At a high level, the authentication phase

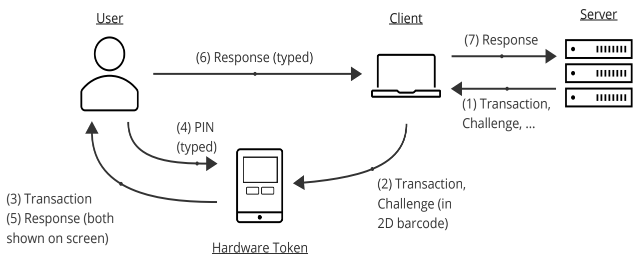

Figure 1: Protocol participants and message flows

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 use the model introduced in [2].

4.0.1 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 a set of oracles, and then running the adversary by letting it ask a polynomial number of queries, including `Test(I)` query, which 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 $\mathcal{A}$. At the end of game $Game^{ake}$, $\mathcal{A}$ outputs its guess b' for bit b involved in the Test-query.

Semantic security. Informally, it requires that the privacy of a session key be preserved in the presence of $\mathcal{A}$. We say that $\mathcal{A}$ wins if it manages to correctly guess (bit b associated with) the session key. We denote its advantage as the probability that $\mathcal{A}$ can correctly guess b, such an advantage is denoted as $Adv_{\pi}^{ss}(\mathcal{A})$. 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. 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 $\mathcal{A}$ is negligible in the security parameter, *i.e.*, $Adv_{\pi}^{aut}(\mathcal{A}) \leq \mu(\lambda)$.

In our scheme, the user must verify a message (e.g., a bank transaction) during the key agreement and authentication phase. To facilitate deterministic verification, especially for the scheme's proof, we define a predicate $y \leftarrow \phi(m, \gamma)$, where $y \in \{0, 1\}$. This predicate, taking a message m (*e.g.*, bank's transactions) and policy γ (*e.g.*, a user's policy specifying a payment amount and destination account number), validates whether the message conforms to the policy, yielding 1 for a match, and 0 otherwise.

Appendix B provides a more in-depth formal discussion of the threat model.

5 THE PROTOCOL

We wish to build an authentication protocol for which the server can verify that the PIN has been entered correctly but 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]. In the real world, users' credentials are sometimes leaked to the public or sold on the black market by hackers who have breached servers [31].

Our protocol consists of three phases: (i) a setup phase, performed only 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. This initial key loading assumption is common to all hardware tokens based around symmetric cryptography, such as the RSA SecureID, and tokens based around asymmetric cryptography have an equivalent certificate loading process, so this assumption does not affect the feasibility of the protocol. Moreover, this key is not generated for any specific user.

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.

Note, in Figure 2, the random ID U_{ID}, serves as the token's serial number. U_{ID} is not specifically generated for any individual user, and the security of our protocol does not rely on U_{ID}'s confidentiality.

User's token		Server
1 :		$k \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
2 :		$st_0 \xleftarrow{\$} \mathsf{FS\text{-}PRG.KGen}(1^\lambda)$
3 :		$U_{\mathrm{ID}} \leftarrow \mathsf{IDGen}(1^\lambda)$
	$\xleftarrow{(k, st_0, U_{\mathrm{ID}})}$	
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 :	$\mathrm{store}(U_{\mathrm{ID}}, k^U, st^U, ct^U, sa^U)$	store $(U_{\mathrm{ID}}, k^S, st^S, ct^S)$

Figure 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 outline how this phase operates. The server initially 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 prompts the user for a PIN, ensuring it matches the user's intention, often by requesting it twice and verifying the consistency of the inputs.

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 synchronizes 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 synchronization 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

User's token		Server
	$\xrightarrow{(\bar{U}_{ID}, \text{enrolment})}$	
1 :		If $\bar{U}_{ID} \neq U_{ID}$, then go to 23
2 :		$ct^S \leftarrow ct^S + 1$
3 :		$kt_1^S, st^S \leftarrow \mathsf{Update}(st^S, 1)$
4 :		$N^S \xleftarrow{\$} \{0,1\}^\lambda$
5 :		$p = N^S \| ct^S$
6 :		$(M, t) \leftarrow \mathsf{Enc}_{k^S}(p)$
	$\xleftarrow{(M, t)}$	
7 :	$(p, b) \leftarrow \mathsf{Dec}_{k^U}(M, t)$	
8 :	If $b \neq 1$, then go to 18	
9 :	Parse p which yields N^M, ct^M	
10 :	If $ct^M \leq ct^U$, then go to 18	
11 :	Request PIN^U from the user	
12 :	$v^U \leftarrow \mathsf{PRF}_{sa^U}(\mathsf{PIN}^U)$	
13 :	$d = ct^M - ct^U$	
14 :	$ct^U \leftarrow ct^U + d$	
15 :	$kt_1^U, st^U \leftarrow \mathsf{Update}(st^U, d)$	
16 :	$p' = N^M \| v^U$	
17 :	$(M', t') \leftarrow \mathsf{Enc}_{kt_1^U}(p')$	
	$\xrightarrow{(M', t')}$	
18 :	$\mathtt{Discard}(\mathsf{PIN}^U, v^U, kt_1^U, N^U)$	$(p', b') \leftarrow \mathsf{Dec}_{kt_1^S}(M', t')$
19 :		If $b' \neq 1$, then go to 23
20 :		Parse p' yielding N^M, v^M
21 :		If $N^M \neq N^S$, then go to 23
22 :		Store v^M as v^S
23 :		$\mathtt{Discard}(kt_1^S, N^S)$

Figure 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.

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, 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 synchronized their 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**. Figure 4 presents the authentication phase in detail.

At a high level, this phase operates as follows. The server initially updates the FS-PRG's state and corresponding counter, which results in a new state st^S, a new random value kt_2^S, and a new temporary counter tmp_{ct^S}.

The server updates the state and the counter one more time which yields a new state st^S, a new random value kt_3^S, and a new counter ct^S. The server generates a random challenge and two ciphertexts, $\ddot{M}$ and $\hat{M}$. The former ciphertext consists of the random challenge and the description of the transaction, encrypted under key kt_2^S. The latter ciphertext contains the counter tmp_{ct^S}, encrypted under key k^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 client, 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) enables the token to deal with the aforementioned 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 Appendix 7, this check will succeed with high probability. The token synchronizes 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 (note that the token can also generate 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, 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

	User's token	**Server**
	$\xrightarrow{(\bar{U}_{ID},\ \text{authentication})}$	
1 :		If $\bar{U}_{ID} \neq U_{ID}$, then go to 21
2 :		$ct^S \leftarrow ct^S + 1$
3 :		$kt_2^S, st^S \leftarrow \mathsf{Update}(st^S, 1)$
4 :		$tmp_{ct^S} \leftarrow ct^S,\ ct^S \leftarrow ct^S + 1$
5 :		$kt_3^S, st^S \leftarrow \mathsf{Update}(st^S, 1)$
6 :		$N^S \xleftarrow{\$} \{0,1\}^\lambda,\ p = N^S \| t^S$
7 :		$(\ddot{M}, \ddot{t}) \leftarrow \mathsf{Enc}_{kt_2^S}(p)$
8 :		$(\hat{M}, \hat{t}) \leftarrow \mathsf{Enc}_{k^S}(tmp_{ct^S})$
	$\xleftarrow{(\ddot{M}, \ddot{t}),\ (\hat{M}, \hat{t})}$	
9 :	$(tmp_{ct^M}, b') \leftarrow \mathsf{Dec}_{k^U}(\hat{M}, \hat{t})$	
10 :	If $b' \neq 1$, go to 21	
11 :	If $tmp_{ct^M} \leq ct^U$, go to 21	
12 :	$d = tmp_{ct^M} - ct^U;\ ct^U \leftarrow ct^U + d + 1$	
13 :	$kt_2^U, st^U \leftarrow \mathsf{Update}(st^U, d)$	
14 :	$(p, b) \leftarrow \mathsf{Dec}_{kt_2^U}(\ddot{M}, \ddot{t})'$; if $b \neq 1$, go to 21	
15 :	Parse p yielding N^M, t^M	
16 :	Display t^M for user to check	
17 :	If user rejects t^M, go to 21	
18 :	Request PIN^U from user	
19 :	$v^U \leftarrow \mathsf{PRF}_{sa^U}(\mathsf{PIN}^U)$	
20 :	$kt_3^U, st^U \leftarrow \mathsf{Update}(st^U, 1)$	
21 :	$p'' = N^M \| t^M \| v^U$	
22 :	$response^U \leftarrow \mathsf{PRF}_{kt_3^U}(p'' \| 1)$	
23 :	$sk^C \leftarrow \mathsf{PRF}_{kt_3^U}(p'' \| 2)$	
	$\xrightarrow{response^U}$	
24 :	$\mathtt{Discard}(\mathsf{PIN}^U, v^U, kt_2^U, kt_3^U, N^M, t^M, tmp_{ct^M}, d)$	
25 :		$expected^S \leftarrow \mathsf{PRF}_{kt_3^S}(N^S \| t^S \| v^S \| 1)$
26 :		If $response^U \neq expected^S$, go to 28
27 :		$sk^S \leftarrow \mathsf{PRF}_{kt_3^S}(N^S \| t^S \| v^S \| 2)$
28 :		$\mathtt{Discard}(kt_2^S, kt_3^S, N^S, t^S, tmp_{ct^S})$

Figure 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. Steps 23 and 27 are optional and can be used when session keys are needed.

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 can generate 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 the adversary's negligible advantage in breaking the authentication of the above protocol.

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 eavesdropping. 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 Π, PRF, and FS-PRG are secure, in the protocol ψ, $Adv^{aut}_{\psi}(\mathcal{A})$ has the upper bound of:

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

Please consult Appendix C for the formal security proofs of the aforementioned theorems.

Informally, our protocol will maintain security even in the event of **client's browser (or general user's computer) compromise**. From a security perspective, compromising the client falls under the **Man-in-the-Middle (MITM)** attack category, where the adversary intercepts and reads the traffic, including the "response".

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 adversary described above 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, its success probability depends on the bit-length of r and if it is set to an appropriate size, then the probability will be negligible. We refer readers to Appendix D, for the full version of the enhanced protocol.

6 INFORMAL SECURITY ANALYSIS

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

6.1 Threats and Protection Objectives

In this section, we outline the threats our protocol must resist.

- <u>*T.DEV: token access.*</u> An adversary may steal the authentication token, gaining access to k^U, sa^U and the current values of ct^U and st^U as we assume that the token does not rely on a trusted chipset.
- <u>*T.MITM: Man-in-the-middle.*</u> An adversary may have access to the traffic exchanged between the client and server.
- <u>*T.PIN: Knowledge of PIN.*</u> An adversary may know the PIN entered by a user, for example from observing them type it in.
- <u>*T.SRV: Server compromise.*</u> The server, being the party that relies on authentication, should not be considered entirely malicious, i.e., an active adversary. However, it is reasonable to assume that the server's database could be compromised, potentially revealing k^S, v^S, and the current values of ct^U and st^U.

Next, we outline our protocol's key security objective.

- <u>*O.AUTH: Authentication.*</u> If the server considers the authentication to have succeeded, then the correct token was used and the correct PIN was entered.
- <u>*O.TRAN: Transaction authentication.*</u> 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.
- <u>*O.PIN: PIN protection.*</u> 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) <u>*O.AUTH against T.PIN and T.MITM.*</u> In this case, the adversary does not have access to the authentication token but does

[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 `Der(.)` that returns a random value of size $\log_2(r)$ bits and (2) sends $r' = x \oplus r$ to the server.

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 that will generate a valid response is the token itself (when the PIN is provided) at line 22 of Figure 4 on page 7. 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. Hence, 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. However, 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.

(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, if the transaction's description, that the server sends to it, has been tampered with.

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

6.3 Excluded Scenarios

We exclude some scenarios (for instance, because they do not make sense or are very hard to efficiently deal with without introducing additional trusted third parties).

- *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. Hence, 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.

We highlight that the existing 2FA schemes also exclude the above scenarios from their threat models.

7 SYNCHRONIZATION

A user's token needs to be synchronized 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 backward.

Specifically, we must provide assurance that the server's state remains at the same 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 synchronization. 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 $(U_{ID}$, 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 forgotten, then users need to prove their identity (e.g., by providing their identity card) to the server which would allow them to enroll again, if the server approves their identity.

9 EVALUATION

In this section, we analyze and compare the 2FA protocol we presented in Section 5 with the smart-card-based protocol proposed in [34], the hardware token-based protocol in [15], and the symmetric-key-based scheme in [22]. We consider the protocols in [15, 34] 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 summarizes the analysis result.

9.1 Computation Cost

We start by analyzing 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 analyze the computation cost of the protocol in [34]. 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

Table 2: Comparison of efficient 2FA protocols.

Features	Operation	Our Protocol	[34]	[15]	[22]
Computation cost	Sym-key	18	19	7	9
	Expo.	0	5	12	0
Communication cost	–	2804-bit	3136-bit	3900-bit	896-bit
Not multiple PINs	–	✓	×	✓	✓
Not modular expo.	–	✓	×	×	✓
Not trusted terminal or chipset	–	✓	×	×	×
Considering Corrupt Server/Token	–	✓	✓	✓	×
Security assumption	–	Standard	Random oracle	Random oracle	Standard

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 analyze 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 initialization 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]. The scheme is highly efficient. It 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, along with those in [15, 22, 34] uses a constant number of symmetric-key primitive invocations. However, our protocol and the protocol in [22] avoid modular exponentiations, resulting in a lower cost compared to the protocols [15, 34].

When executed on the extremely low-power microcontrollers common in dedicated hardware authentication tokens, **asymmetric cryptography consumes approximately 1000 times more energy than its corresponding symmetric counterpart** [26]. These tokens are designed to function for several years without battery replacements, typically lacking user-replaceable or rechargeable batteries to cut manufacturing costs and enhance tamper resistance. **The power saving that results from using solely symmetric primitives is therefore important for feasibility**.

For the commercial product that this protocol was implemented for, there was no microcontroller available on the market that was both cost-effective and able to execute asymmetric cryptography with a speed that would result in a good user experience.

9.2 Communication Cost

We next analyze 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 transmits 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. Hence, the total communication cost of our protocol is about 2804 bits.

Next, we evaluate the cost of the protocol in [34]. 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 [34]. 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 analyze 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 initialization 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 [34] 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.

When communication is implemented using a 2-D barcode, this reduction in message size as compared to the state of the art allows for an increased level of error correction in the generated barcodes. Consequently, the decoding process becomes more resilient to inaccurate alignment and issues like reflections on the client screen, therefore improving user experience through a reduction in time needed to capture a valid barcode.

9.3 Other Features

In our protocol, a user is required to know and input only a single secret, specifically a PIN. Also, we do not make any trust assumptions regarding users' personal computers, or their web browsers. Hence, they can be corrupted at any time, for example, simultaneously with the corruption of another party, such as the token or server.

In contrast, the protocol outlined in [34] necessitates that a user possesses and enters an extra secret, namely a random ID. As demonstrated by Scott [29], this approach does not maintain its security, even if only the user's ID is exposed. Additionally, in this scheme, users are required to input their PINs into a separate device, specifically a card reader. To ensure the security of the scheme, it is essential that the card reader is fully trusted, especially in situations where the server or smart card is compromised.

Likewise, the protocol described in [15] requires that users enter their PINs into their personal computers instead of inserting them into the dedicated hardware token. This approach poses a significant problem, as the PINs are at a higher risk of exposure to attackers. This is due to the fact that users' computers are (i) frequently connected to the Internet and (ii) used for various purposes, making them more susceptible to breaches. To ensure the security of the protocol, it is necessary for these computers to be trusted, even when either the token or the server is compromised. In contrast, our protocol mandates users to input their PIN into the hardware token, exclusively dedicated to authentication and consistently disconnected from the Internet.

In the protocol presented in [22] a user is only required to know a single PIN. However, it has a reliance on the complete trustworthiness of the server and utilizes a trusted chipset. Both our protocol and the one described in [22] are secure within the standard model, whereas the protocols presented in [15, 34] operate in the non-standard random oracle model.

10 CONCLUSION AND FUTURE WORK

In this work, we have introduced a 2FA protocol designed to withstand a formidable adversary who may (a) eavesdrop on the communication between a user and the server, and (b) gain physical access to the user's hardware token, its PIN, or compromise the server. Our protocol offers a distinctive combination of key features that are not found in state-of-the-art schemes. Specifically, our protocol (i) mandates the user to remember only a single PIN, (ii) relies solely on symmetric-key primitives, (iii) operates within a standard model, (iv) does not assume trust in the user's computer (and accordingly its browser), and (v) incurs minimal communication costs. This protocol is the first of its kind to offer these features without necessitating tamper-resistant hardware.

It would be intriguing to augment the security of the proposed protocol to withstand an even more formidable adversary than the one considered in this paper – an adversary capable of compromising both a user's token and the server.

REFERENCES

[1] 2015. Payment Services (PSD 2). Directive 2015/2366/EU of the European Parliament and of the Council.

[2] Mihir Bellare, David Pointcheval, and Phillip Rogaway. 2000. Authenticated Key Exchange Secure against Dictionary Attacks. In *EUROCRYPT*.

[3] Mihir Bellare and Bennet S. Yee. 2003. Forward-Security in Private-Key Cryptography. In *CT-RSA*.

[4] Alex Biryukov, Joseph Lano, and Bart Preneel. 2003. Cryptanalysis of the alleged SecurID hash function. In *International Workshop on Selected Areas in Cryptography*.

[5] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. 2003. Security proofs for an efficient password-based key exchange. In *ACM CCS*.

[6] C-C Chang and T-C Wu. 1991. Remote password authentication with smart cards. *IEE Computers and Digital Techniques* (1991).

[7] Donghoon Chang, Sweta Mishra, Somitra Kumar Sanadhya, and Ajit Pratap Singh. 2017. On Making U2F Protocol Leakage-Resilient via Re-keying. *IACR Cryptol. ePrint Arch.* (2017), 721.

[8] Ankita Chaturvedi, Ashok Kumar Das, Dheerendra Mishra, and Sourav Mukhopadhyay. 2016. Design of a secure smart card-based multi-server authentication scheme. *J. Inf. Secur. Appl.* (2016).

[9] Emiliano De Cristofaro, Honglu Du, Julien Freudiger, and Greg Norcie. 2013. 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. 2022. Moving the U.S. Government Toward Zero Trust Cybersecurity Principles. https://www.whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf.

[11] Haonan Feng, Hui Li, Xuesong Pan, and Ziming Zhao. 2021. A Formal Analysis of the FIDO UAF Protocol. In *NDSS*. The Internet Society.

[12] Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. 2006. A Method for Making Password-Based Key Exchange Resilient to Server Compromise. In *CRYPTO*.

[13] BB Gupta, Varun Prajapati, Nadia Nedjah, P Vijayakumar, Ahmed A Abd El-Latif, and Xiaojun Chang. 2021. 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] Abdullah Imran, Habiba Farrukh, Muhammad Ibrahim, Z. Berkay Celik, and Antonio Bianchi. 2022. SARA: Secure Android Remote Authorization. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1561–1578. https://www.usenix.org/conference/usenixsecurity22/presentation/imran

[15] Stanislaw Jarecki, Mohammed Jubur, Hugo Krawczyk, Nitesh Saxena, and Maliheh Shirvanian. 2021. Two-factor Password-authenticated Key Exchange with End-to-end Security. *ACM Trans. Priv. Secur.* (2021).

[16] Ari Juels, Nikolaos Triandopoulos, Marten Van Dijk, John Brainard, Ronald Rivest, and Kevin Bowers. 2016. Configurable one-time authentication tokens with improved resilience to attacks. US Patent 9,270,655.

[17] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography, Second Edition.* CRC Press.

[18] Sang-Kyun Kim and Min Gyo Chung. 2009. More secure remote user authentication scheme. *Computer Communications* (2009).

[19] Dmitry Kogan, Nathan Manohar, and Dan Boneh. 2017. T/Key: Second-Factor Authentication From Secure Hash Chains. In *CCS*. ACM.

[20] Radhesh Krishnan Konoth, Björn Fischer, Wan J. Fokkink, Elias Athanasopoulos, Kaveh Razavi, and Herbert Bos. 2020. SecurePay: Strengthening Two-Factor Authentication for Arbitrary Transactions. In *EuroS&P*.
[21] Ijlal Loutfi and Audun Jøsang. 2015. FIDO Trust Requirements. In *NordSec 2015*.
[22] Shin'ichiro Matsuo, Daisuke Moriyama, and Moti Yung. 2011. Multifactor Authenticated Key Renewal. In *Trusted Systems - Third International Conference, INTRUST 2011, (Lecture Notes in Computer Science)*. Springer.
[23] Steven J. Murdoch, Ingolf Becker, Ruba Abu-Salma, Ross J. Anderson, Nicholas Bohm, Alice Hutchings, M. Angela Sasse, and Gianluca Stringhini. 2016. 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)*.
[24] OneSpan. [n.d.]. *Digipass 770 Datasheet*. https://www.onespan.com/resources/digipass-770/datasheet.
[25] Christoforos Panos, Stefanos Malliaros, Christoforos Ntantogian, Angeliki Panou, and Christos Xenakis. 2017. A Security Evaluation of FIDO's UAF Protocol in Mobile and Embedded Devices. In *TIWDC*.
[26] Nachiketh R. Potlapally, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. 2003. Analyzing the Energy Consumption of Security Protocols. In *International Symposium on Low Power Electronics and Design* (Seoul, Korea). ACM, 30–35.
[27] Niranchana Radhakrishnan and Amutha Prabakar Muniyandi. 2022. Dependable and Provable Secure Two-Factor Mutual Authentication Scheme Using ECC for IoT-Based Telecare Medical Information System. *Journal of Healthcare Engineering* (2022).
[28] RSA Security LLC. 2021. RSA SecurID Hardware Authenticators, Technical Specifications. https://www.securid.com/wp-content/uploads/2021/11/rsa-securid-hardware-tokens-technical-specifications-012621.pdf.
[29] Michael Scott. 2012. Cryptanalysis of a recent two factor authentication scheme. *IACR Cryptol. ePrint Arch.* (2012), 527.
[30] Thales. 2020. Gemalto SWYS QR Reader Eco. https://www.thalesgroup.com/sites/default/files/database/document/2020-12/fs-QR-code-reader.pdf.
[31] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. 2019. Protecting accounts from credential stuffing with password breach alerting. In *USENIX*. USENIX Association, 1556–1571.
[32] Youliang Tian, Qiuxian Li, Jia Hu, and Hui Lin. 2020. Secure limitation analysis of public-key cryptography for smart card settings. *World Wide Web* (2020).
[33] Ding Wang, Qianchen Gu, Haibo Cheng, and Ping Wang. 2016. The Request for Better Measurement: A Comparative Evaluation of Two-Factor Authentication Schemes. In *AsiaCCS*.
[34] Ding Wang and Ping Wang. 2018. Two Birds with One Stone: Two-Factor Authentication with Security Beyond Conventional Bound. *IEEE Trans. Dependable Secur. Comput.* (2018).
[35] Zhenfeng Zhang, Yuchen Wang, and Kang Yang. 2020. Strong Authentication without Temper-Resistant Hardware and Application to Federated Identities.. In *NDSS*.

Appendix A 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. However, 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.

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

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

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].

Appendix B FORMAL THREAT MODEL

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.

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 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.* [34] 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 $\texttt{Send}(U^i, \text{start})$ initializes 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.

B.0.1 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 $Adv^{ss}_{\pi}(\mathcal{A}) = 2Pr[b = b'] - 1$, 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.*, $Adv^{ss}_{\pi}(\mathcal{A}) \leq \mu(\lambda)$.

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.*, $Adv^{aut}_{\pi}(\mathcal{A}) \leq \mu(\lambda)$.

Appendix 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.

Proof. 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 ($\texttt{Enc}$ and $\texttt{Dec}$) 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:

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

Similar to the security proof in [5], we assume that if any of the games halt 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 $\mathsf{Send}(.)$ 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 5 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 6 and 7 respectively. Also, we present the simulators for the rest of the queries in Figure 8.

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 $\mathsf{Send}(S^j, \bar{response}^U)$ 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 $\Big((U_{ID}, \text{enrolment}), (U_{ID}, \text{authentication}), (\bar{M}, \bar{t}), (\bar{M}', \bar{t}'), (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'), \bar{response}^U\Big) \in \overrightarrow{L}$.
  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 $\mathsf{Send}(S^j, \bar{response}^U)$, 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 $\Big((U_{ID}, \text{enrolment}), (U_{ID}, \text{authentication}), (\bar{M}, \bar{t}), (\bar{M}', \bar{t}'), (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'), \bar{response}^U\Big) \in \overrightarrow{L}$. 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,

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

- ***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 $\mathsf{Send}(U^i, (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'))$ as follows.
  (1) compute $\bar{response}^U \leftarrow \mathsf{PRF}_{\ddot{t}'}(\ddot{M}'||1)$.
  (2) compute $\bar{sk}^U \leftarrow \mathsf{PRF}_{\ddot{t}'}(\ddot{M}'||2)$.
  We also amend the way the server compiles query $\mathsf{Send}(S^j, \bar{response}^U)$ 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,
  $$|Pr[S_7] - Pr[S_6]| \le Pr[\hat{Auth}_7]$$
  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,
  $$|Pr[S_7] - Pr[S_6]| \le \frac{2q_s}{2^\lambda} \tag{10}$$
  Moreover, the session key and authenticator are random values, as they are the outputs of PRF whose secret key is not known. Therefore, $Pr[S_7] = \frac{1}{2}$.
  By summing up all the above relations 4-10, we would have
  $$|Pr[S_7] - Pr[S_0]| \le (q_s + q_p)\Big(Adv^{\mathsf{PRF}}(\mathcal{A}) + Adv^{Enc}(\mathcal{A})\Big) + \frac{4(q_s + q_p)}{2^\lambda} + \frac{4q_s}{2^\lambda} \tag{11}$$
  By combining Equations 3 and 11, we would have:

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

□

<u>Pseudorandom function</u>

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 $f(r)$.

Figure 5: Pseudorandom function's simulator.

<u>$\mathsf{Send}(U^i, .)$</u>

This query is dealt with as below:
- if the user's instance is not in the "expecting" state and it receives query $\mathsf{Send}(U^i, \mathsf{start}, \mathsf{phase})$, where $\mathsf{phase} \in \{\mathsf{enrolment}, \mathsf{authentication}\}$ 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 $\mathsf{Send}(U^i, (\bar{M}, \bar{t}))$, 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.*, $b \ne 1$), 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: $\forall i, 1 \le i \le ct^M - ct^U$: (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 $\mathsf{Send}(U^i, (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'))$, 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., $b' \ne 1$), 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: $\forall i, 1 \le i \le tmp_{ct^M} - ct^U$: (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.*, $b \ne 1$), 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 $\Big((U_{ID}, \mathsf{enrolment}), (U_{ID}, \mathsf{authentication}), (\bar{M}, \bar{t}), (\bar{M}', \bar{t}'), (\ddot{M}', \ddot{t}'), (\hat{M}', \hat{t}'), \bar{response}^U\Big) \in \vec{L}$.

Figure 6: Simulators for Send query to a user's instance.

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

This query is dealt with as below:

- upon receiving $\underline{\mathsf{Send}(S^j, (U_{ID}, \text{enrolment}))}$, 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 $\underline{\mathsf{Send}(S^j, (\bar{M}', \bar{t}'))}$, 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 $\underline{\mathsf{Send}(S^j, (U_{ID}, \text{authentication}))}$, 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 $\underline{\mathsf{Send}(S^j, \bar{response}^U)}$, 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.

Figure 7: Simulators for Send **query to a server's instance.**

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

This query is dealt with as below:

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

$\underline{\underline{\mathtt{Reveal}(I)}}$

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.

$\underline{\underline{\mathtt{Test}(I)}}$

This query is processed as below.

(1) $sk \leftarrow \mathtt{Reveal}(I)$.
(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.

Figure 8: 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 $\bar{aut}$.

Proof. 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:

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

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:

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

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 them with Equation 12:

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

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 $\bar{\mathrm{Cpt}}_2 = \texttt{Corrupt}(C, 2)$. We argue that in this case, the upper bound of $\mathcal{A}$'s advantage will be changed as follows: $Adv_{\psi,\bar{\mathrm{Cpt}}_2}^{aut}(\mathcal{A}) \leq \frac{q_s}{N}$.

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.

Appendix D MORE ENHANCED PROTOCOL

In this section, we outline a more enhanced version of the protocol presented in Section 5. 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[5] 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 bit-length of r and if it is set to an appropriate size, then the probability will be negligible. We present the new versions of the setup, enrolment, and authentication phases in Figures 9, 10, and 11. 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 :			$U_{\mathrm{ID}} \leftarrow \mathsf{IDGen}(1^\lambda)$
		$\xleftarrow{(k,\, st_0,\, U_{\mathrm{ID}})}$	
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 :	$r \xleftarrow{\$} \{0,1\}^\iota$		
9 :	$\mathsf{store}(U_{\mathrm{ID}}, k^U, st^U, ct^U, sa^U, r)$		$\mathsf{store}\ (U_{\mathrm{ID}}, k^S, st^S, ct^S)$

Figure 9: Setup phase.

[5]For the token to securely send r to the server, the token (1) derives a fresh random value x from kt_3^U using a key derivation function $\texttt{Der}(.)$ that returns a random value of size $\log_2(r)$ bits and (2) sends $r' = x \oplus r$ to the server.

	User's token		**Server**
		$\xrightarrow{(U_{ID}, \text{enrolment})}$	
1 :			$ct^S \leftarrow ct^S + 1$
2 :			$kt_1^S, st^S \leftarrow \mathsf{Update}(st^S, 1)$
3 :			$N^S \xleftarrow{\$} \{0,1\}^\lambda$
4 :			$p = N^S \| ct^S$
5 :			$(M, t) \leftarrow \mathsf{Enc}_{k^S}(p)$
		$\xleftarrow{(M, t)}$	
6 :	$(p, b) \leftarrow \mathsf{Dec}_{k^U}(M, t)$		
7 :	If $b \neq 1$, then go to 18		
8 :	Parse p which yields N^M, ct^M		
9 :	If $ct^M \leq ct^U$, then go to 18		
10 :	Request PIN^U from the user		
11 :	$v^U \leftarrow \mathsf{PRF}_{sa^U}(\mathsf{PIN}^U)$		
12 :	$d = ct^M - ct^U$		
13 :	$ct^U \leftarrow ct^U + d$		
14 :	$kt_1^U, st^U \leftarrow \mathsf{Update}(st^U, d)$		
15 :	$w^U \leftarrow \mathsf{PRF}_r(v^U)$		
16 :	$p' = N^M \| v^U \| w^U$		
17 :	$(M', t') \leftarrow \mathsf{Enc}_{kt_1^U}(p')$		
		$\xrightarrow{(M', t')}$	
18 :	$\mathtt{Discard}(\mathsf{PIN}^U, v^U, kt_1^U, N^U)$		$(p', b') \leftarrow \mathsf{Dec}_{kt_1^S}(M', t')$
19 :			If $b' \neq 1$, then go to 23
20 :			Parse p' yielding N^M, v^M, w^M
21 :			If $N^M \neq N^S$, then go to 23
22 :			Store v^M as v^S and w^M as w^S
23 :			$\mathtt{Discard}(kt_1^S, N^S)$

Figure 10: Enrolment phase.

	User's token		**Server**
		$\xrightarrow{(U_{ID}, \text{authentication})}$	
1 :			$ct^S \leftarrow ct^S + 1$
2 :			$kt_2^S, st^S \leftarrow \mathsf{Update}(st^S, 1)$
3 :			$tmp_{ct^S} \leftarrow ct^S,\ ct^S \leftarrow ct^S + 1$
4 :			$kt_3^S, st^S \leftarrow \mathsf{Update}(st^S, 1)$
5 :			$N^S \xleftarrow{\$} \{0,1\}^\lambda,\ p = N^S \| t^S$
6 :			$(\ddot{M}, \ddot{t}) \leftarrow \mathsf{Enc}_{kt_2^S}(p)$
7 :			$(\hat{M}, \hat{t}) \leftarrow \mathsf{Enc}_{k^S}(tmp_{ct^S})$
		$\xleftarrow{(\ddot{M}, \ddot{t}), (\hat{M}, \hat{t})}$	
8 :	$(tmp_{ct^M}, b') \leftarrow \mathsf{Dec}_{k^U}(\hat{M}, \hat{t})$; if $b' \neq 1$, go to 21		
9 :	If $tmp_{ct^M} \leq ct^U$, go to 21		
10 :	$d = tmp_{ct^M} - ct^U$; $ct^U \leftarrow ct^U + d + 1$		
11 :	$kt_2^U, st^U \leftarrow \mathsf{Update}(st^U, d)$		
12 :	$(p, b) \leftarrow \mathsf{Dec}_{kt_2^U}(\ddot{M}, \ddot{t})$; if $b \neq 1$, go to 21		
13 :	Parse p yielding N^M, t^M		
14 :	Display t^M for user to check		
15 :	If user rejects t^M, go to 21		
16 :	Request PIN^U from user		
17 :	$v^U \leftarrow \mathsf{PRF}_{sa^U}(\mathsf{PIN}^U)$		
18 :	$kt_3^U, st^U \leftarrow \mathsf{Update}(st^U, 1)$		
19 :	$p'' = N^M \| t^M \| v^U$; $response^U \leftarrow \mathsf{PRF}_{kt_3^U}(p'' \| 1)$		
20 :	$x \leftarrow \mathtt{Der}(kt_3^U), r' = x \oplus r$; $sk^C \leftarrow \mathsf{PRF}_{kt_3^U}(p'' \| 2)$		
		$\xrightarrow{response^U, r'}$	
21 :	$\mathtt{Discard}(\mathsf{PIN}^U, v^U, kt_2^U, kt_3^U, N^M, t^M, tmp_{ct^M}, d, x)$		
22 :			$x \leftarrow \mathtt{Der}(kt_3^U), r = x \oplus r'$
23 :			$w^M \leftarrow \mathsf{PRF}_r(v^S)$
24 :			If $w^M \neq w^S$, go to 28
25 :			$expected^S \leftarrow \mathsf{PRF}_{kt_3^S}(N^S \| t^S \| v^S \| 1)$;
26 :			If $response^U \neq expected^S$, go to 28
27 :			$sk^S \leftarrow \mathsf{PRF}_{kt_3^S}(N^S \| t^S \| v^S \| 2)$
28 :			$\mathtt{Discard}(kt_2^S, kt_3^S, N^S, t^S, tmp_{ct^S}, x, r)$

Figure 11: Authentication phase.