Zero Round-Trip Time for the Extended Access Control Protocol

Jacqueline Brendel[1] and Marc Fischlin[2]

Cryptoplexity, Technische Universität Darmstadt, Germany
jacqueline.brendel@cryptoplexity.de marc.fischlin@cryptoplexity.de

Abstract. The Extended Access Control (EAC) protocol allows to create a shared cryptographic key between a client and a server. It is for instance referenced by the International Civil Aviation Organization for securing the communication between machine readable travel documents and terminals, and is also deployed on current German identity cards. Here we discuss how to enhance the EAC protocol by a so-called zero-round trip time (0RTT) mode. Through this mode the client can, without further interaction, immediately derive a new key from cryptographic material exchanged in previous executions. This makes the 0RTT mode attractive from an efficiency viewpoint such that the upcoming TLS 1.3 standard, for instance, will include its own 0RTT mode. Here we show that the EAC protocol can be augmented to support a 0RTT mode, too. Our proposed EAC+0RTT protocol is compliant with the basic EAC protocol and adds the 0RTT mode smoothly on top. We also prove the security of our proposal according to the common security model of Bellare and Rogaway in the multi-stage setting.

1 Introduction

Electronic identity systems in the European Union were put on formal grounds in July 2016, when regulation EU No 910/2014 about electronic identification, authentication, and trust services for electronic transactions (eIDAS) came into effect. While this regulation sets the framework for such systems, concrete implementations yet need to be explored. A promising candidate is the joint effort of the French and German IT security agencies [9] which in turn is based upon the already existing German identity card systems [8] and also referenced by the International Civil Aviation Organization for machine readable travel documents [23].

At the core of the eIDAS proposal is the *extended access control* (EAC) protocol between a client's card (also called chip in this context) and a server (or, terminal). The EAC protocol establishes an authenticated key between both parties over a public channel. For this, both parties run a sophisticated Diffie-Hellman key exchange protocol in which either party deploys its certified long-term key. As such, the EAC protocol is amenable to cryptographic analysis as an authenticated key exchange scheme. Indeed, it has already been confirmed by Dagdelen and Fischlin [13] that the EAC provides strong security in the widely accepted security model of Bellare and Rogaway [1].

1.1 Striving for Zero Round-Trip Time

The EAC protocol consists of two connected phases, the terminal authentication (TA), followed by the chip authentication (CA). Both steps require only a mild number of message exchanges to establish a session key. At the same time, recent efforts in the area of key exchange protocols aim at modes of operations which allow for even faster data delivery. More precisely, it should be possible for a party to re-use cryptographic data from a previous connection to derive a fresh session key without further interaction, thus allowing the party to transmit data immediately. Such a mode is called *zero round-trip time* (0RTT).

The first proposal for a 0RTT-supporting protocol came from Google with its QUIC protocol [19]. The 0RTT mode allows the client to send data to a known server without having to wait for the server's response. This idea was then quickly adopted for the drafts of the new TLS version 1.3, and has been included in the latest drafts in various versions [30,31,29]. Even on a network layer level, the Windows Networking Team recently announced to support 0RTT for TCP connections in order to reduce latency (see [10] for TCP Fast Open description).

The rough idea of the approach taken by QUIC and TLS (for the Diffie-Hellman version [30])[1] is that, upon the first encounter, the server also sends a semi-static public key g^s as part of the authenticated key exchange. Unlike an ephemeral key, which is used only within a single session, and a long-term key which spans over a large amount of sessions, such a semi-static key is valid for a very limited time only. This time period may range from a few seconds to a couple of days. In particular, the semi-static key may be used in multiple sessions.

The next time the client contacts the server, the client may combine a fresh ephemeral key g^c with the server's semi-static key g^s to immediately compute a Diffie-Hellman key g^{cs} and derive an intermediate session key. The client can now send g^c and already deliver data secured under the intermediate session key, without round trip. For both QUIC and TLS the parties then continue the key exchange protocol to switch to full session keys.

It is obvious that the non-interactive derivation of the 0RTT session key comes at a price in terms of security: Since the server cannot contribute to such a key in a per-session manner, an adversary can replay the client's protocol message and data to the server. This is inevitable, but accepted by the designers of QUIC and TLS 1.3 as worthwhile to achieve the desired level of efficiency.

The EAC protocol is at foremost a general key exchange protocol, as QUIC and TLS, such that a 0RTT mode for EAC may be desirable in some settings. More importantly, the EAC protocol is increasingly adopted as an omnipotent solution in related scenarios, for example to secure transactions [28] and for attribute-based access control with smart cards [27]. Especially in the latter case, if deployed in situations where user experience hinges on fast response times, e.g.,

[1] The latest version of the TLS draft [29] focuses on a pre-shared key 0RTT version and has for now dropped the Diffie-Hellman based version; the main EAC protocol only supports a Diffie-Hellman based key exchange, though.

turnstile access in subway stations [18], reducing the latency is important. This requirement has led for instance to the development of the ISO/IEC 24727-6 and ANSI 504-1 standardized "Open Protocol for Access Control Identification and Ticketing with privacY" (OPACITY) for smart cards [32], which uses a related concept of persistent binding for speeding up the key generation process. Unfortunately—and also underlining the importance of rigor—OPACITY has been shown to display cryptographic weaknesses [14].[2]

1.2 Our Contribution and Related Work

Here we show that also the EAC protocol can be augmented to support a 0RTT mode. Interestingly, the extension can be added on top with minimal changes to the original protocol. As in the proposal of QUIC and TLS 1.3 we let the terminal include an additional semi-static key pk_T^{semi} in the regular EAC execution. The key is transmitted as part of the auxiliary data field of the original EAC description, and is thus also authenticated through the terminal's signature in the TA phase.

In the full run of the EAC protocol the semi-static key is still ignored for the session key derivation. Instead, and as in the original EAC description, the chip then receives the terminal's ephemeral key and derives a session key from its certified long-term key and this ephemeral key. The client authenticates through a message authentication code under the session key. In this regard, the slightly modified protocol complies with the original EAC protocol, using the auxiliary data field to transfer an additional key.

If a chip later wants to reconnect to a terminal for which it already holds the semi-static key, it only runs the CA phase again. But instead of receiving a fresh ephemeral key from the terminal, it uses the semi-static key to build the session key. Note that the semi-static key is already authenticated through the previous execution of the EAC protocol. Omitting the transmission of the terminal's ephemeral key turns this step into a non-interactive protocol.

A straightforward idea to improve efficiency further may be to use the terminal's ephemeral key once more for 0RTT, instead of using the semi-static key. The downside is that the terminal would need to store all ephemeral keys in a certain time frame. This is why, both, we here as well as TLS [30] uses semi-static keys instead. Nonetheless we discuss some potential variations of our basic designs in Section 5.

We then show that our EAC+0RTT protocol, which consists of the (augmented) EAC protocol run followed by any number of subsequent 0RTT EAC protocol executions, meets the common security properties of an authenticated key exchange protocol. But we, of course, need to account for the possibility of replay attacks on the 0RTT data. Furthermore, it is convenient to model the possibly many 0RTT EAC handshakes following a single EAC execution in a

[2] Remarkably, the publication of this analysis pre-dates the latest version of SP800-73-4 [11], dated May 2015, which lists OPACITY as a suitable solution for key establishment.

so-called *multi-stage* setting. To this end we adopt the multi-stage extension of the Bellare-Rogaway model in [16].

The proof of security for the EAC+0RTT protocol does not rely on previous results. Nevertheless, we wish to mention the many security analyses of the German identity card protocols and certain eIDAS extensions [4,13,26,12,3,2,21,5,22]. Also, we remark general approaches to build low-latency protocols such as [20] cannot be applied in the context of the EAC protocol without major changes to the protocol.

1.3 Discussion

We emphasize that the design choices of the original EAC protocol are beyond our discussion here. Our goal is to show that a 0RTT version can be implemented based on the existing infrastructure. In particular, it is important that such a solution is "non-invasive" in the sense that it does not require major changes to the existing protocol but is added "on top". Of course, any extension brings some modifications, e.g., in our case both the chip and the terminal must now implement the 0RTT EAC protocol and store semi-static keys. Yet, our proposal for the augmented EAC protocol complies with the original EAC description by using the auxiliary data field for the semi-static key. Furthermore, the 0RTT mode is identical to the plain execution of the CA phase, only that the semi-static key identifier is used instead of the one for the ephemeral key.

We also stress that we do not comment on the security-efficiency trade-off concerning 0RTT modes, but rather offer the option to have such a mode for the EAC protocol in principle. Whether chips and terminals eventually support this mode and tolerate for example the replay problem, is case dependent. Still, the examples of QUIC and TLS 1.3 indicate that, from an engineering perspective, the desire to have such modes exists, and we provide a potential technical solution for EAC.

Finally, let us point out that 0RTT transfers inherently include the small risk that the transmitted data cannot be recovered by the receiver, e.g., if the receiver has switched the semi-static key in the meantime. For common client-server scenarios the client may thus have to re-transmit the data. This problem is often outweighed by the efficiency gain in the regular cases. For smart card applications it may be preferable to have the terminal first signal its support of 0RTT and to communicate the current identifier of the semi-static key, thus saving the card from performing unnecessary operations. This can be done with the transmission of the certificate in the first step of the TA protocol, allowing the card to decide which mode to execute. Strictly speaking, this would effectively support a "lightweight 1RTT" protocol mode, still with significant efficiency advantages.

2 Protocol Description

We next present the Extended Access Control protocol and its extension to support 0RTT. The 0RTT extension should be seen as a particular mode or sub

protocol which co-exists with the original EAC protocol. In particular, many instances of 0RTT EAC may follow a single full EAC protocol run (until pk_T^{semi} changes, in which case the terminal will most likely reject).

2.1 The Extended Access Protocol

The Extended Access Control protocol establishes a secure channel between a chip and a terminal. It is divided in two phases: the Terminal Authentication (TA) and Chip Authentication (CA) as depicted in Figure 1. We integrate the 0RTT EAC protocol to the existing EAC protocol smoothly by using the pre-specified auxiliary data field in which any data can be sent in an authenticated manner to the chip during the TA phase. The auxiliary data field has originally been included to pass further information to the chip such as the current date, and the original EAC protocol ignores any such data if sent under an unknown object identifier. In our case, the terminal can utilize this field to transmit its semi-static key pk_T^{semi} to the chip to enable future 0RTT EAC executions.

Terminal Authentication. The terminal authentication lets the chip C verify the terminal T's identity and its permissions to access sensitive data. This is achieved via the certificate $cert_T$ held by T. This certificate contains not only the terminal's signed public key but also its granted access rights. We assume that each certificate *cert* contains some unique identifier *certID* which can either be the serial number or an identifier like `CertID` or `CertUID`, and that *certID* allows to determine the user identity. Furthermore, as mentioned earlier, the terminal authentication can be used to distribute the terminal's public semi-static key to the chip, thereby permitting future 0RTT EAC executions.

In a first step, the terminal sends its certificate for verification to the chip, which can then either abort, in case of an invalid certificate, or proceed by extracting the terminal's public key pk_T from the valid certificate. If the session was not aborted by C, T generates its ephemeral key pair (epk_T, esk_T) and sends the compressed version of the ephemeral key epk_T to C. This initiates a challenge-response mechanism. The chip replies with a nonce r_C chosen uniformly at random. The terminal authentication is complete, if the chip can then successfully verify the received signature $s_T \leftarrow \mathsf{Sig}(sk_T, id_C||r_C||\mathsf{Compr}(epk_T)||pk_T^{\mathsf{semi}})$ over the chip's identity, chosen nonce and the compressed ephemeral key. Depending on whether the terminal offers support for 0RTT executions, the signature may contain the terminal's semi-static public key pk_T^{semi}.

Chip Authentication. In the second part of the EAC protocol, the chip is authenticated to the terminal and a session key for subsequent encrypted and integrity-protected communications between chip and terminal is established. The chip transmits its credentials to the terminal and receives in response the ephemeral public key epk_T (if the terminal did not abort due to an invalid certificate). After checking epk_T against the compressed value received during the TA phase, the chip can compute the Diffie-Hellman value k from epk_T and its own

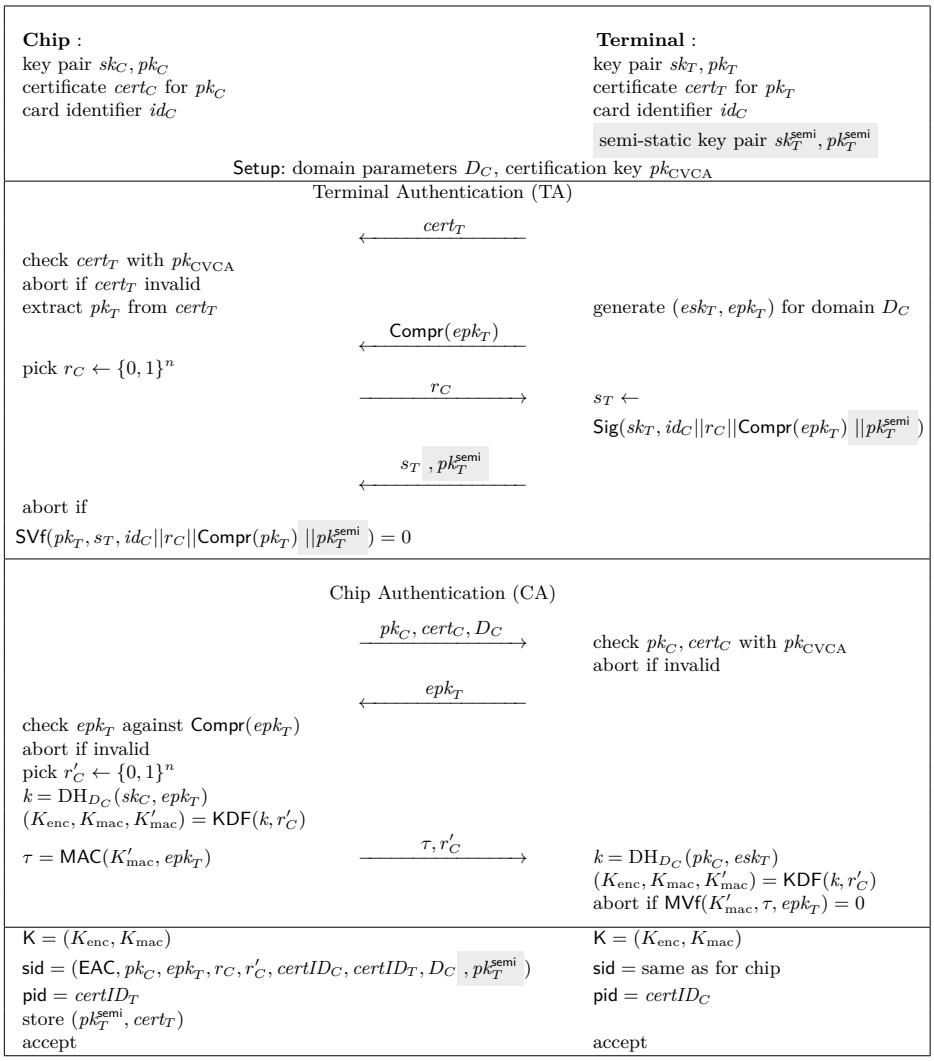

Fig. 1: Terminal Authentication (TA) and Chip Authentication (CA). All operations are modulo q resp. over the elliptic curve. The gray part shows the 0RTT support inserted in the (optional) auxiliary data field.

long-term secret key sk_C. Together with a uniformly random value r'_C, the DH value k is used to derive an encryption key K_{enc}, as well as two authentication keys $K_{\mathrm{mac}}, K'_{\mathrm{mac}}$.[3] For final authentication, the chip uses K'_{mac} to compute a

[3] For the necessity of K'_{mac} in a proof in the Bellare-Rogaway-style we refer to the discussion in [13].

tag τ over the ephemeral public key of the terminal. The tag is then transmitted to the terminal, alongside the random value r'_C used in the key derivation. The terminal is now able to derive the DH key k and subsequently the keys $(K_{\text{enc}}, K_{\text{mac}}, K'_{\text{mac}})$, where the session key K is given by $(K_{\text{enc}}, K_{\text{mac}})$. The terminal aborts the CA phase prematurely if it is not able to verify τ. Otherwise the session identifier and partner identifier are generated on both sides. If C has received a semi-static key, it saves this key along with the terminal's certificate $cert_T$ for further reference. The EAC protocol execution is completed successfully if both parties terminate in accepting state.

2.2 The 0RTT EAC Protocol

Figure 2 shows the modified protocol supporting 0RTT between a chip C and a terminal T. The chip now holds additional information in form of the semi-static public key pk_T^{semi}, which it obtained during a previous EAC protocol interaction with T. In the 0RTT extension of the EAC protocol, C and T perform the following actions, corresponding to a non-interactive version of the CA protocol since the pk_T^{semi} is used instead of epk_T. Thus, the extra communication round in the CA protocol in which T sends the (uncompressed) ephemeral key becomes obsolete.

At first, the chip C picks a random nonce r''_C and computes the DH shared value $k = \text{DH}_{D_C}(sk_C, pk_T^{\mathsf{semi}})$. Using these two values, C then derives the keys $(K_{\text{enc}}, K_{\text{mac}}, K'_{\text{mac}})$ where, as in the EAC protocol, K'_{mac} is an additional authentication key used internally in the 0RTT EAC key exchange (see [13] for a discussion). The session key is then given by $\mathsf{K} = (K_{\text{enc}}, K_{\text{mac}})$. Finally, C computes the MAC-value over the semi-static public key

$$\tau = \mathsf{MAC}(K'_{\text{mac}}, pk_T^{\mathsf{semi}})$$

and sends its first (and only) flight of data to T consisting of

- the authentication token τ,
- the previously chosen nonce r''_C,
- its public key pk_C, as well as its certificate $cert_C$,
- the domain parameter D_C, and
- early application data encrypted under the previously derived key.

Upon receiving the chip's message, T verifies the validity of pk_C and $cert_C$, and aborts if the verification is unsuccessful. Otherwise, T uses the public key, its semi-static secret sk_T^{semi} and the random nonce r''_C to derive K'_{mac} and the 0RTT EAC session key K. T can then check the validity of the authentication token τ and aborts if the tag cannot be verified. If τ is valid, T decrypts the attached early application data. This completes the 0RTT EAC execution.

If the terminal does not support 0RTT, or the semi-static key provided by the chip is outdated or otherwise invalid, the process is aborted and the chip must initiate a fresh execution of the full EAC protocol in order to establish an authenticated secure channel with the terminal. There are, of course, several conceivable ways to recover from failures in the 0RTT handshake. Possible alternatives are described in Section 5.3.

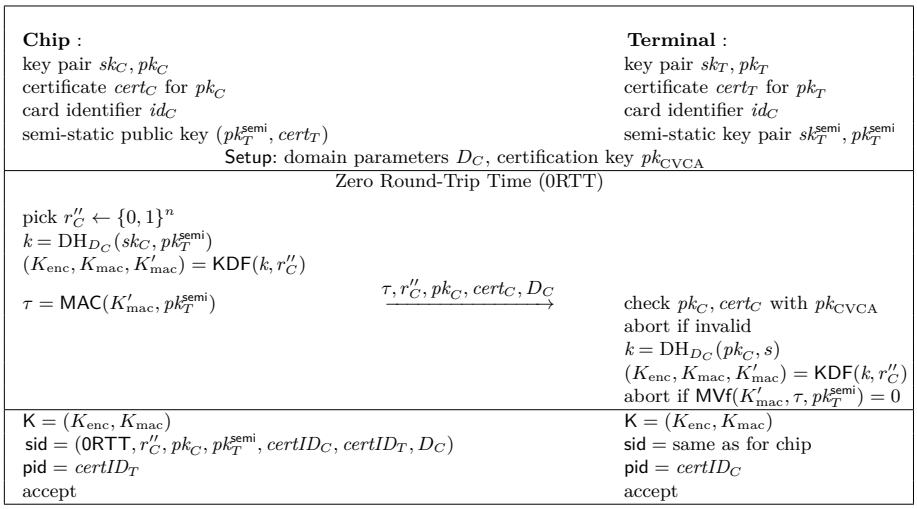

Fig. 2: 0RTT EAC. All operations are modulo q resp. over the elliptic curve. Note that the fields sid and pid are used within the security proof and describe partnered sessions and intended communication partners.

3 Security Model

In this section we present the security model underlying our analysis. As a basis we use the common real-or-random model of Bellare and Rogaway [1] which provides strong security guarantees against active attacks. To capture the notion of zero round-trip time in a multi-stage setting we adapt the model proposed for the analysis of Google's QUIC protocol by Fischlin and Günther [16]. The model will be suitable for protocols which have a special 0RTT mode in which the initiator of a previous session can use a semi-static key of a responder to create a shared session key with a single message.

3.1 Overview

As mentioned beforehand, the security model is situated within the game-based approach of Bellare and Rogaway (BR) [1] in which an adversary, with full control over the network, must be able to distinguish real session keys from independently drawn keys. To this end, the adversary can interact with protocol participants and instances via oracles. To initiate a new session the adversary can call the NewSession oracle, which takes a label to determine which of the two modes (full EAC or 0RTT EAC) to execute. The adversary can query the Send oracle to send protocol message to an instance, immediately getting the party's reply in return. The adversary is furthermore permitted to learn the long-term secret keys of parties through a Corrupt oracle.

Leakage of session keys and semi-static secret keys, which are used to derive 0RTT session keys, is modeled through Reveal and RevealSemiStaticKey queries, respectively. To engage with the BR game, the adversary may perform Test queries for some session(s) of the protocol, resulting in either the receipt of the corresponding session key or of an independently and uniformly chosen key, the choice made at random. In order to win the game, the adversary must now distinguish which kind of key it received. A detailed overview over the queries is given in Section 3.3.

In order to avoid trivial attacks, some restrictions concerning the Test queries apply. Foremost, the party of a tested session must not be corrupt, or else the adversary is trivially able to compute the session key. Analogously, neither the tested session key has been revealed to the adversary nor the party's semi-static secret key in case of the 0RTT mode. Since both communication parties are supposed to derive the same session key in a key exchange protocol, we must also rule out similar trivial attacks on the communication partner of a tested session. Here, communication partners are usually identified through session identifiers which determine sessions belonging together. Details follow.

3.2 Multi-Stage Key Exchange

A single execution of EAC between a chip and a terminal may be followed by multiple 0RTT handshakes between the parties. To model this situation, we adopt the notion of *multi-stage* key exchange as originally introduced in the related QUIC analysis of Fischlin and Günther [16]. This model allows for multiple keys to be established within a single session. As opposed to the multi-stage setting encountered in e.g. QUIC, we can make use of a simplified setting here, since no key derived within a session is used to secure communications in further stages of the same session. Thus, all keys derived in a single session can be seen as independent.

In order to describe sessions formally, the subsequent notations are used, essentially following [16]. The participants in the key exchange protocol are given by elements in the set $\mathcal{U}$. Often, parties fulfill different roles, acting as clients and servers, or in the EAC context both chips and terminals.

Chips are usually denoted by $C \in \mathcal{U}$ and terminals by $T \in \mathcal{U}$. Each party $U \in \mathcal{U}$ holds a long-term public key pk_U with corresponding secret key sk_U and certificate $cert_U$ (under the certification authority's public key pk_{CVCA}). Additionally, terminals T can hold semi-static key pairs $(pk_T^{\mathsf{semi}}, sk_T^{\mathsf{semi}})$. Since the semi-static key may change during the attack (where we even let the adversary determine the point in time when the key is refreshed, via NewSemiStaticKey queries) it is useful to introduce an identifier sskid for the current value. As the semi-static keys can also be revealed to the adversary through a RevealSemiStaticKey query we also introduce a flag $\mathsf{st}_{\mathsf{ssk},\mathsf{sskid}} \in \{\mathsf{fresh}, \mathsf{revealed}\}$ to mark the current state of the semi-static key sskid.

Each participant can run multiple instances of the key exchange protocol in parallel. In the EAC+0RTT case each of these sessions is usually composed out of one execution of the EAC protocol, followed by an arbitrary number of

0RTT EAC executions. We call each of these sub protocol execution a *stage*. For simplicity we assume that M is the maximum number of stages.[4]

Each session can be uniquely referred to by an administrative identifier $\mathsf{label} = (U, k) \in \mathsf{LABELS}$, where $U \in \mathcal{U}$ denotes the *owner* of the session label and $k \in \mathbb{N}$ is used to determine which of the potentially multiple sessions of U is considered.

During the attack a *list of sessions* $\mathsf{List_S}$ is kept which contains a record for any existing session. Each record is updated during the attack and holds the following information:

- $\mathsf{label} \in \mathsf{LABELS}$ is the (administrative) session label.
- $\mathsf{role} \in \{\mathsf{initiator}, \mathsf{responder}\}$ is the session owner's role in this session. For the EAC protocol this corresponds to the distinction between chip (= $\mathsf{initiator}$) and terminal (= $\mathsf{responder}$).[5]
- $\mathsf{stage} \in \{1, \ldots, \mathsf{M}\}$ specifies the current stage, where M is the maximum stage. The stage changes from $i-1$ to i once the state of execution $\mathsf{st}_{\mathsf{exec},i-1}$ changes to either $\mathsf{accepted}$ or $\mathsf{rejected}$.
- mode describes the order of the session's sub protocols to be executed and this vector with M entries is initialized upon creation of the session. For EAC+0RTT case mode can either be $(\mathsf{EAC}, \mathsf{0RTT}, \mathsf{0RTT}, \ldots)$ or it can be $(\mathsf{0RTT}, \mathsf{0RTT}, \mathsf{0RTT}, \ldots)$ to cover the cases that the session should first execute the EAC protocol and then 0RTT EAC runs, or start with 0RTT EAC executions right away (e.g., if the chip already knows the semi-static key from some other session). Each new EAC execution is captured by opening a new session.
- pid_i specifies the intended communication partner in stage i. The default value is $(\perp, \ldots, \perp)$ and for each stage the value may be set to an actual identifier once during the execution.
- ownid identifies, similar to pid, the session owner but remains unchanged between stages.
- sskid is the key identifier for the semi-static key pair used in a session. Default is $\perp$. Note that we let the sskid remain identical among stages of a session since we assume that the key is stored by the party throughout the session.
- $\mathsf{st}_{\mathsf{exec},i} \in \{\mathsf{running}, \mathsf{accepted}, \mathsf{rejected}\}$ denotes the current state of execution in stage i. The default value upon creation of the session and after an increment in stage is $\mathsf{running}$
- $\mathsf{sid} \in (\{0,1\}^* \cup \{\perp\})^{\mathsf{M}}$ indicates the session identifiers where sid_i indicates the session identifier for stage $i \in \{1, \ldots, \mathsf{M}\}$. The default value is $(\perp)^{\mathsf{M}}$.
- $\mathsf{st}_{\mathsf{key}} \in \{\mathsf{fresh}, \mathsf{revealed}\}^{\mathsf{M}}$ indicates the state of the session key K_i for stage $i \in \{1, \ldots, \mathsf{M}\}$. The default value for each stage is fresh.
- $\mathsf{K} \in (\{0,1\}^* \cup \{\perp\})^{\mathsf{M}}$ where K_i indicates the established session key for stage $i \in \{1, \ldots, \mathsf{M}\}$. The default value for each stage is $\perp$.

[4] As in [16], the constant M for the maximum stage is introduced to facilitate the notation. In fact, M may be arbitrarily large.

[5] This is the point of view from the cryptographic protocol flow, although the terminal may technically first need to power the card, of course.

- $\mathsf{tested} \in \{\mathsf{true}, \mathsf{false}\}^{\mathsf{M}}$ keeps track of tested session keys, where tested_i indicates whether K_i has been tested or not. Default value for each key is false.

In order to be able to refer to a specific entry within the tuple for a session in $\mathsf{List_S}$, we use the notation $\mathsf{label}.\langle\mathsf{entry}\rangle$. For example, $\mathsf{label.ownid}$ specifies the owner of the session with unique identifier label. If we compare two sessions with labels $\mathsf{label}, \mathsf{label}'$ then, if clear from the context, we sometimes simply write $\mathsf{sid}, \mathsf{sid}'$ instead of $\mathsf{label.sid}$ and $\mathsf{label}'.\mathsf{sid}$, and similar for other entries.

Partnering of Sessions and Correctness. We call the session label owned by U *partnered* with the session label' owned by V (and vice versa), if

- the sessions share the same session identifier, i.e., for each stage $i \in \{1, \ldots, \mathsf{M}\}$ $\mathsf{sid}_i = \mathsf{sid}'_i$,
- the partner identifiers of both sessions reflect the partnered instances accordingly, e.g., in the EAC protocol C has set $\mathsf{pid} = \mathit{certID}_T$ and T has set $\mathsf{pid}' = \mathit{certID}_C$.

The pairing of partnered sessions is defined via the session identifiers. We require that any execution between honest instances, that was not interfered with, is correctly partnered. Furthermore, as a correctness criterion, partnered sessions should have derived the same session key. We will later demand this explicitly as part of the Match security property. Note, that the own id ownid as well as the partner id pid is set in the EAC+0RTT protocol according to the identities given in the certificates cert_C and cert_T, respectively.

3.3 Adversary Model

We model the adversary by a probabilistic polynomial time (PPT) Turing machine denoted by $\mathcal{A}$. The adversary is active and in full control over the network. This implies in particular that— additional to the interception of messages—the adversary can schedule when (and if) message delivery occurs. Furthermore, the adversary may alter and inject messages. We assume the adversary learns if a participant in the protocol has terminated and/or accepted.

Forward Secrecy. The leakage of a party's long-term secret(s) may result in the insecurity of session keys that were derived using the then compromised key material. Session keys that remain secure, even in the event of a long-term secret compromise, are called *forward secure*. We note that in the 0RTT EAC protocol the corruption of a terminal T and thereby the leakage of its long-term secret key has no impact on the security of session keys derived by T (or its intended partners). This is due to the fact that the long-lived secret sk_T is only involved in the authentication when signing the ephemeral and semi-static key. The long-term secrets of chips, however, are explicitly used to derive session keys, and therefore these keys cannot support forward security. To reflect this one-sided forward security we say that the 0RTT EAC protocol provides *terminal forward secrecy*. More abstractly, we speak of $\mathsf{responder}$ forward secrecy in our model.

Authentication. The authentication property of the EAC+0RTT protocol is *mutual* in the sense that both parties authenticate themselves via certificates. Put differently, none of the parties stays anonymous (as clients in some TLS sub protocol versions do).

Adversarial Queries. As indicated before, in order to break key secrecy, the goal of the adversary is to distinguish real from random-looking session keys. Not all interactions of the adversary with the protocol are admissible at any point. In particular, there are conditions under which the adversary trivially loses the game, e.g. when both revealing and testing session keys of partnered sessions as mentioned before. To keep track if one these cases has occurred, a flag lost is introduced with initial value false. In order to interact with the protocol, the adversary can issue the following queries to the oracles:

$\mathsf{NewSession}(U, \mathsf{role}, \mathsf{prot})$: Establishes a new session with unique new label label for U, stores the role value in $\mathsf{label.role} \leftarrow \mathsf{role}$, and stores the specified type of session prot to be established in $\mathsf{label.mode}$, e.g., for EAC+0RTT we have $\mathsf{prot} \in \{\mathsf{EAC},\mathsf{0RTT}\} \times \{\mathsf{0RTT}\}^{\mathsf{M}-1}$. Returns label.

$\mathsf{NewSemiStaticKey}(T)$: Generates a fresh semi-static key pair $(pk_T^{\mathsf{semi}}, sk_T^{\mathsf{semi}})$ for terminal $T \in \mathcal{U}$ along with corresponding new unique identifier sskid. In particular, $\mathsf{st}_{\mathsf{ssk},\mathsf{sskid}} \leftarrow \mathsf{fresh}$ is set. Returns pk_T^{semi} and sskid. We conservatively assume that the adversary may still run executions with previous semi-static keys, e.g., if servers in a distributed setting do not update the keys perfectly simultaneously.

$\mathsf{Send}(\mathsf{label}, m)$: Causes the message m to be sent to the session label. If there exists no session with label label, the query outputs $\perp$. Else the response of the session owner U upon receipt of message m is returned, and the state of execution $\mathsf{st}_{\mathsf{exec}}$ is updated.

- Should $\mathsf{st}_{\mathsf{exec},i}$ change to $\mathsf{accepted}$ for some i, the current execution of the protocol is suspended and $\mathsf{accepted}$ is returned. The adversary can now, for example, decide whether to test the session key or not, and can later resume the session by sending a special $\mathsf{Send}(\mathsf{label}, \mathsf{cont})$ command.
- Should $\mathsf{st}_{\mathsf{exec},i}$ change to $\mathsf{accepted}$ for some i and there exists a session label' with $\mathsf{sid}_i = \mathsf{sid}'_j$ and $\mathsf{st}'_{\mathsf{key},j} = \mathsf{revealed}$ for some j with $(\mathsf{label}', j) \neq (\mathsf{label}, i)$, then $\mathsf{st}_{\mathsf{key},i}$ is also marked as $\mathsf{revealed}$. This captures for example replay attacks in another session or even within this session here at a different stage.
- Should $\mathsf{st}_{\mathsf{exec},i}$ change to $\mathsf{accepted}$ for some i while there exists a session label' with $\mathsf{sid}_i = \mathsf{sid}'_j$ such that $\mathsf{tested}'_j = \mathsf{true}$ for some j with $(\mathsf{label}', j) \neq (\mathsf{label}, i)$, set $\mathsf{label.K}_i \leftarrow \mathsf{label}'.\mathsf{K}_j$ and $\mathsf{tested}_i \leftarrow \mathsf{true}$.
- Should $\mathsf{st}_{\mathsf{exec},i}$ change to $\mathsf{accepted}$ for some i with an intended communication partner pid_i that was previously corrupted, then set $\mathsf{st}_{\mathsf{key},i} \leftarrow \mathsf{revealed}$.

$\mathsf{Reveal}(\mathsf{label}, i)$: Returns the session key K_i of the session with label label.

- If there exists no session labeled label, or if $i > \mathsf{stage}$, or if the session key K_i has already been tested, or if $\mathsf{st}_{\mathsf{exec},i} \neq \mathsf{accepted}$, then return $\perp$. Otherwise, set $\mathsf{st}_{\mathsf{key},i}$ to $\mathsf{revealed}$ and return K_i to the adversary.

- If there exists a session with label label' in $\mathsf{List_S}$ with $\mathsf{sid}_i = \mathsf{sid}'_j$ for some j, then set $\mathsf{st}'_{\mathsf{key},j} \leftarrow \mathsf{revealed}$.

$\mathsf{RevealSemiStaticKey}(\mathsf{sskid})$: If there exists a semi-static key pair $(pk_T^{\mathsf{semi}}, sk_T^{\mathsf{semi}})$ corresponding to the identifier sskid, then sk_T^{semi} is output and $\mathsf{st}_{\mathsf{ssk},\mathsf{sskid}}$ is set to $\mathsf{revealed}$. To reflect the impact of the revealed semi-static key on sessions that used this key, set $\mathsf{label}.\mathsf{st}_{\mathsf{key},i}$ to $\mathsf{revealed}$ for all $i \in \{1, \ldots, M\}$ and for each session label with $\mathsf{label}.\mathsf{sskid} = \mathsf{sskid}$.

$\mathsf{Corrupt}(U)$: Returns the long-term secret key sk_U of U to the adversary. Sessions owned by U may not be queried any further.

- If U is a chip, forward secrecy is not given and all session keys must be regarded as exposed. Therefore, $\mathsf{label}.\mathsf{st}_{\mathsf{key},i} \leftarrow \mathsf{revealed}$ for each session label owned by U and for all $i \in \{1, \ldots, M\}$.
- If, on the other hand, U is a terminal, terminal forward secrecy is still granted and, thus, no further measures need to be taken.

$\mathsf{Test}(\mathsf{label}, i)$: Tests the session key of stage i in the session corresponding to label. The oracle receives the test bit b_{test} as state. This bit b_{test} is chosen randomly at the outset and then considered permanently set during the game execution.

- If there exists no session with label label or if $\mathsf{label}.\mathsf{st}_{\mathsf{exec},i} \neq \mathsf{accepted}$, the query returns $\perp$.
- If stage i of the session with label label has been tested previously, i.e., if $\mathsf{tested}_i = \mathsf{true}$ for this session, the session key K_i is returned once more.
- In any other case, tested_i is set to true. If $b_{\mathsf{test}} = 0$, a key $\mathcal{K} \xleftarrow{\$} \mathcal{D}$ is sampled at random from the session key distribution $\mathcal{D}$. If $b_{\mathsf{test}} = 1$, on the other hand, $\mathcal{K}$ is set to the actual session key $\mathsf{label}.\mathsf{K}_i$ of stage i. Additionally, for a partnered session label' with $\mathsf{sid}_i = \mathsf{sid}'_j$ (if existent) set $\mathsf{label}'.\mathsf{K}_j \leftarrow \mathsf{label}.\mathsf{K}_i$ and $\mathsf{label}'.\mathsf{tested}_j \leftarrow \mathsf{true}$. Return $\mathcal{K}$.

3.4 Multi-Stage BR Security of 0RTT Key Exchange Protocols

We adopt the approach of Brzuska et al. [7,6] and, Fischlin et al. [16], and therefore separate the required security properties into Match security and BR security. While the conditions on Match security guarantee that the session identifiers sid enable the correct identification of partnered sessions, Multi-Stage BR security refers to Bellare-Rogaway-like key secrecy as introduced earlier in the multi-stage setting.

The subsequent analysis of the EAC+0RTT protocol is based on the following security notions as described in [15] and adapted to our particular setting:

Match Security. The Match security game $G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}}$ is defined as follows.

Definition 1 (Match security). *Let n be the security parameter. Furthermore let KE be a key exchange protocol and $\mathcal{A}$ a PPT adversary interacting with KE via the queries defined in Section 3.3 in the following game $G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}}(n)$:*

Setup. *The challenger generates long-term public/private-key pairs with certificates for each participant $U \in \mathcal{U}$.*

Query. *The adversary $\mathcal{A}$ receives the generated public keys and has access to the queries* NewSession*,* NewSemiStaticKey*,* Send*,* Reveal*,* RevealSemiStaticKey*, and* Corrupt*.*

Stop. *At some point, the adversary stops with no output.*

We say that $\mathcal{A}$ wins the game, denoted by $G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}}(n) = 1$, if at least one of the following conditions hold:

1. *There exist two labels* $\mathsf{label}, \mathsf{label}'$ *and stages* $i, j \in \{1, \ldots, M\}$ *such that* $(\mathsf{label}, i) \neq (\mathsf{label}', j)$ *but* $\mathsf{sid}_i = \mathsf{sid}'_j \neq \perp$*,* $\mathsf{label}.stage \geq i$*,* $\mathsf{label}'.stage \geq j$ *and* $\mathsf{st}_{\mathsf{exec},i} \neq \mathsf{rejected}$*, and* $\mathsf{st}'_{\mathsf{exec}}[j] \neq \mathsf{rejected}$*, but* $\mathsf{K}_i \neq \mathsf{K}'_i$*. (Different session keys in partnered sessions, either within the same session at different stages or across two sessions.)*
2. *There exist two labels* $\mathsf{label}, \mathsf{label}'$ *such that* $\mathsf{sid}_i = \mathsf{sid}'_j \neq \perp$ *for some stages* $i, j \in \{1, \ldots \mathsf{M}\}$*,* $\mathsf{role} = \mathsf{initiator}$*, and* $\mathsf{role}' = \mathsf{responder}$*, but* $\mathsf{label.ownid} \neq \mathsf{label}'.\mathsf{pid}$ *or* $\mathsf{label.pid} \neq \mathsf{label}'.\mathsf{ownid}$*. (Different intended partner.)*
3. *There exist at least three labels* $\mathsf{label}, \mathsf{label}'$ *and* label'' *and stages* i, j, k *such that* (label, i)*,* (label', j)*,* (label'', k) *are pairwise distinct, but* $\mathsf{sid}_i = \mathsf{sid}'_j = \mathsf{sid}''_k \neq \perp$ *and for any two of the three sessions with role* responder *and mode* 0RTT *it holds that the owners are distinct.*
   (More than two sessions share a session id for some stage and this event was not caused by a simple replay attack on the 0RTT protocol for the same responder.)

We say KE *is* Match-secure *if for all PPT adversaries $\mathcal{A}$ the following advantage function is negligible in the security parameter n:* $\mathsf{Adv}^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} := \Pr\left[G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}}(n) = 1\right]$.

Multi-Stage BR Key Secrecy. Next is the key secrecy in the multi-stage setting:

Definition 2 (BR Key Secrecy). *Let n be the security parameter. Furthermore let* KE *be a key exchange protocol with key distribution $\mathcal{D}$ and let $\mathcal{A}$ be a PPT adversary interacting with* KE *via the queries defined in Section 3.3 within the following game $G^{\mathsf{BR},\mathcal{D}}_{\mathsf{KE},\mathcal{A}}(n)$:*

Setup. *The challenger generates long-term public/private-key pairs and certificate for each participant $U \in \mathcal{U}$, chooses the test bit $b_{\mathsf{test}} \xleftarrow{\$} \{0,1\}$ at random, and sets* $\mathsf{lost} \leftarrow \mathsf{false}$*.*

Query. *The adversary $\mathcal{A}$ receives the generated public keys and has access to the queries* NewSession*,* NewSemiStaticKey*,* Send*,* Reveal*,* RevealSemiStaticKey*,* Corrupt*, and* Test*. Note that these queries may set* lost *to* true*.*

Guess. *At some point, $\mathcal{A}$ stops and outputs a guess b_{guess}.*

Finalize. *The challenger sets the 'lost' flag to* $\mathsf{lost} \leftarrow \mathsf{true}$ *if there exist two (not necessarily distinct) labels* $\mathsf{label}, \mathsf{label}'$ *and stages* $i, j \in \{1, \ldots, \mathsf{M}\}$ *such that* $\mathsf{sid}_i = \mathsf{sid}'_j$*,* $\mathsf{label}.\mathsf{st}_{\mathsf{key},i} = \mathsf{revealed}$*, and* $\mathsf{label}'.\mathsf{tested}_j = \mathsf{true}$*. (Adversary has tested and revealed the key in a single session or in two partnered sessions.)*

$\mathcal{A}$ wins the game, denoted by $G^{\mathsf{BR},\mathcal{D}}_{\mathsf{KE},\mathcal{A}} = 1$, if $b_{\mathsf{guess}} = b_{\mathsf{test}}$ and $\mathsf{lost} = \mathsf{false}$. *We say that* Multi-Stage BR key secrecy *holds for* KE *if for all PPT adversaries $\mathcal{A}$ the advantage function*

$$\mathsf{Adv}^{\mathsf{BR},\mathcal{D}}_{\mathsf{KE},\mathcal{A}}(n) := \Pr\left[G^{\mathsf{BR},\mathcal{D}}_{\mathsf{KE},\mathcal{A}}(n) = 1\right] - \frac{1}{2}$$

is negligible in the security parameter n. A key exchange protocol KE *is further called* Multi-Stage BR-secure *if* KE *is both* Match*-secure and* BR *key secrecy for* KE *holds.*

We note that the winning conditions are independent of the forward secrecy property of the KE protocol. Forward secrecy is already taken into account in the formulation of the Reveal and Corrupt queries and the finalization step of the game.

4 Security of the EAC+0RTT Protocol

In this section we will first describe the security assumptions on which we will base the then following security proof of the EAC+0RTT protocol.

4.1 Security Assumptions

In the following we will provide definitions of the basic cryptographic assumptions underlying the security proof of the EAC+0RTT protocol. In particular, we will introduce a double-sided (or symmetric) variant of the PRF-ODH assumption, further referred to as dPRF-ODH. We start by recalling what it means for signatures and certificates to be existentially unforgeable under chosen message attacks:

Definition 3 (EUF-CMA assumption). *Let n be the security parameter. Furthermore let $\mathcal{S} = (\mathsf{SKG}, \mathsf{Sig}, \mathsf{SVf})$ be a signature scheme and let $\mathcal{A}$ be a PPT algorithm. We define the following* EUF-CMA *security game $G^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{A}}(n)$:*

Setup. *Generate a key pair $(pk, sk) \xleftarrow{\$} \mathsf{SKG}(1^n)$ and give pk to the adversary $\mathcal{A}$.*
Query Phase. *In the next phase $\mathcal{A}$ can adaptively query messages $m_1, m_2, \ldots, m_q \in \{0,1\}^*$ with $q \in \mathbb{N}$ arbitrary, which the signing oracle answers with $\sigma_1 \leftarrow \mathsf{Sig}(sk, m_1), \sigma_2 \leftarrow \mathsf{Sig}(sk, m_2), \ldots, \sigma_q \leftarrow \mathsf{Sig}(sk, m_q)$.*
Output. *At some point, $\mathcal{A}$ outputs a message m^* and a potential signature σ^*. Output 1 iff $\mathsf{SVf}(pk, m^*, \sigma^*) = 1$ and $m^* \neq m_i$ for all $i = 1, 2, \ldots, q$.*

We define the advantage function

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{S},\mathcal{A}}(n) := \Pr\left[G^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{A}}(n) = 1\right]$$

We say that a signature scheme $\mathcal{S}$ is EUF-CMA secure*, if for any $\mathcal{A}$ the advantage function is negligible (as a function in n).*

The definitions for certification schemes work analogously. That is, a certification scheme consists of three algorithms $\mathcal{C} = (\mathsf{CKG}, \mathsf{CA}, \mathsf{CVf})$ for creating the authority's key pair, the certification of a public key, and for verifying a public key with respect to a certificate. We allow for multiple certifications of the same public key but assume that each certification requests is accompanied by an identifier id which will be included in *certID*. Then we can define unforgeability as for signatures, implying that the adversary cannot forge a valid certificate for a new public key or for a previously certified key under a new identity. We write $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{C},\mathcal{A}}$ for the advantage of an adversary in the $\mathsf{EUF\text{-}CMA}$ game against a certification scheme. In the EAC protocol the authority's public key is given by pk_{CVCA} and the key generation, certificate creation and certificate verification are often described implicitly only.

Furthermore, we can define message authentication codes (MACs) $\mathcal{M} = (\mathsf{MKG}, \mathsf{MAC}, \mathsf{MVf})$ analogously, except that the key generation algorithm only outputs a single secret key and the adversary does not receive any initial input in the attack. We write $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{M},\mathcal{A}}$ for the advantage of an adversary $\mathcal{A}$ in this game.

Finally, we need that the compression function Compr is collision-resistant. That is, for an adversary $\mathcal{A}$ it should be infeasible to find group elements $X \neq Y$ such that $\mathsf{Compr}(X) = \mathsf{Compr}(Y)$. We write $\mathsf{Adv}^{\mathsf{CR}}_{\mathsf{Compr},\mathcal{A}}$ to denote the advantage of such an adversary $\mathcal{A}$. We remark that we actually need a weaker requirement from Compr, resembling second preimage resistance, namely that for a random group element X it should be hard to find a colliding different Y, when given the discrete logarithm of X with respect to the group.

Next, we define our version of the $\mathsf{PRF\text{-}ODH}$ assumption as a slight extension of the original definition in [24,25]:

Definition 4 ($\mathsf{dPRF\text{-}ODH}$ assumption). *Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order q with generator g, $\mathsf{PRF}\colon \mathbb{G} \times \{0,1\}^* \to \{0,1\}^n$ be a pseudorandom function with keys in $\mathbb{G}$, input strings from $\{0,1\}^*$, and output strings of length n, let $b \in \{0,1\}$ be a bit, and $\mathcal{A}$ be a PPT algorithm.*

We define the following $\mathsf{dPRF\text{-}ODH}$ security game $G^{\mathsf{dPRF\text{-}ODH},\mathbb{G},b}_{\mathsf{PRF},\mathcal{A}}$:

Setup. *The challenger chooses $u \xleftarrow{\$} \mathbb{Z}_q$ at random and gives g^u to $\mathcal{A}$.*

Query Phase 1. *In the next phase $\mathcal{A}$ can ask queries of the form $(A, x) \in \mathbb{G} \times \{0,1\}^*$ to oracle $\mathcal{O}_u$ which the challenger answers with the value $y \leftarrow \mathsf{PRF}(A^u, x)$ if $A \in \mathbb{G}$. Otherwise the challenger returns $\perp$.*

Challenge. *At some point $\mathcal{A}$ asks a challenge query $\hat{x} \in \{0,1\}^*$ upon which the challenger chooses $v \xleftarrow{\$} \mathbb{Z}_q$ at random, sets $y_0 \leftarrow \mathsf{PRF}(g^{uv}, \hat{x})$ and $y_1 \xleftarrow{\$} \{0,1\}^n$, and answers with (g^v, y_b).*

Query Phase 2. *As in the first query phase the adversary $\mathcal{A}$ can ask queries of the form $(A, x) \in \mathbb{G} \times \{0,1\}^*$ to the oracle $\mathcal{O}_u$ which the challenger answers with the value $y \leftarrow \mathsf{PRF}(A^u, x)$, except that now $\mathcal{A}$ is not allowed to query the challenge pair $(g^v, \hat{x})$. Additionally,[6] $\mathcal{A}$ can now ask queries of the form*

[6] This is the extra power which the adversary has in the double-sided version $\mathsf{dPRF\text{-}ODH}$ over the plain $\mathsf{PRF\text{-}ODH}$ assumption.

$(B, x) \in \mathbb{G} \times \{0,1\}^*$ *which are distinct from the pair* $(g^u, \hat{x})$ *to an oracle* $\mathcal{O}_v$. *The challenger answers such queries with the value* $y \leftarrow \mathsf{PRF}(B^v, x)$.

Guess. *Eventually,* $\mathcal{A}$ *stops and outputs a bit* b' *which is also the game's output, denoted by* $G^{\mathsf{dPRF\text{-}ODH},\mathbb{G},b}_{\mathsf{PRF},\mathcal{A}}$.

We define the advantage function $\mathsf{Adv}^{\mathsf{dPRF\text{-}ODH},\mathbb{G}}_{\mathsf{PRF},\mathcal{A}} := \left| \Pr\left[G^{\mathsf{dPRF\text{-}ODH},\mathbb{G},0}_{\mathsf{PRF},\mathcal{A}} = 1\right] - \Pr\left[G^{\mathsf{dPRF\text{-}ODH},\mathbb{G},1}_{\mathsf{PRF},\mathcal{A}} = 1\right] \right|$ *and, assuming a sequence of groups in dependency of the security parameter, we say that the* dPRF-ODH *assumption holds for* PRF *with keys from* $(\mathbb{G}_n)_n$ *if for any* $\mathcal{A}$ *the advantage function is negligible (as a function in* n*, assuming that we have a sequence of groups in* n*).*

4.2 Match Security

In this section we show that the EAC+0RTT protocol achieves Match security, satisfying basic partnering properties. Recall that we defined the session identifiers in the main EAC protocol to be

$$\mathsf{sid} = (\mathsf{EAC}, pk_C, epk_T, r_C, r'_C, certID_C, certID_T, D_C\ , pk^{\mathsf{semi}}_T).$$

and in the 0RTT EAC protocol as

$$\mathsf{sid} = (\mathsf{0RTT}, r''_C, pk_C, pk^{\mathsf{semi}}_T, certID_C, certID_T, D_C).$$

Theorem 1. *The EAC+0RTT protocol is* Match*-secure. For any efficient adversary* $\mathcal{A}$ *we have*

$$\mathsf{Adv}^{\mathsf{Match}}_{EAC,\mathcal{A}} \leq q_p^2 \cdot \max\{2^{-|\mathsf{nonce}|}, \tfrac{1}{q}\}$$

where q_p *is the maximum number of sub protocol executions,* $|\mathsf{nonce}|$ *is the bit-length of each of the nonces* r_C, r'_C, r''_C*, and* q *is the order of the group from which (ephemeral) keys are chosen.*

Note that $q_p \leq \mathsf{M} \cdot q_s$ for the maximal number M of stages and the maximal number q_s of sessions.

Proof. In order to achieve Match Security (cf. Definition 1), we need to show that the following three conditions hold:

(i) Partnered sessions derive the same session key in all stages.
(ii) Sessions are partnered correctly, i.e., with the intended communication partner in all stages.
(iii) No more than two sessions share a session identifier for some stage.

Ad (i): Since partnered sessions share the same session identifiers sid by definition, and in particular the matching sub protocol mode, they necessarily also agree on the session key. This is due to the fact that the session identifier already determines all inputs for key derivation: The entries epk_T (in case of EAC)

resp. pk_T^{semi} (in case of 0RTT EAC), and pk_C and D_C. These values ensure the correct computation of the DH key k on either side by specifying the DH shares for computation in the domain D_C. This DH key k is then used as input to the key derivation function, along with the nonce r'_C (for EAC) resp. r''_C (for 0RTT EAC). Since the corresponding nonce is also contained in the session identifier, and thereby guaranteed to be identical in both the chip's and terminal's key derivation, both sides will output the same value for the session key $\mathsf{K} = (K_{\text{enc}}, K_{\text{mac}})$.

Ad (ii): Both certificate identifiers $certID_C$ and $certID_T$ are included in the session identifier. Therefore, agreement on the session identifier yields agreement on the intended partner's identity, as reflected in the respective certificate.

Ad (iii): We distinguish two cases, according to the sub protocol.

If all three session identifiers agree for the EAC protocol, then it must be that any combination of two sessions of honest users with identical identifiers yields a third collision only if the honest chip or honest terminal in the third session picks its random value r_C, r'_C (in case of a chip) resp. epk_T (in case of a terminal) such that it collides. This can only happen with probability at most $\max\{2^{-2n}, 1/q\}$ for the length n of r_C and r'_C resp. the group size q. Since there are at most q_p^2 many combinations of the initial two sessions, where q_p denotes the maximal number of sub protocol executions, we derive a bound of $q_p^2 \cdot \max\{2^{-2n}, 1/q\}$ in this case.

The other case is when the threefold collision occurs for 0RTT EAC sessions. Note that the adversary can only win if at least two of the three sessions are owned by a chip (with $\mathsf{role} = \mathsf{initiator}$). However, the probability that any two sub executions in chip sessions match on r''_C is at most $q_p^2 \cdot 2^{-n}$. □

Multi-Stage BR Key Secrecy. Similarly, we can show key secrecy, and even argue forward secrecy with respect to subsequent terminal corruptions. We note that forward secrecy with respect to chip corruptions is impossible to achieve for EAC since the chip does not generate ephemeral keys for executions but rather uses the long-term secrets:

Theorem 2. *The EAC+0RTT protocol provides key secrecy (with* $\mathsf{responder}$ *forward secrecy). That is, for any efficient adversary $\mathcal{A}$ there exist efficient adversaries $\mathcal{B}_3, \mathcal{B}_4, \mathcal{B}_5, \mathcal{B}_{10/11}$ such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{BR},\mathcal{D}}_{\mathsf{KE},\mathcal{A}}(n) \leq\ & 3q_p^2 \cdot \max\{2^{-|\mathsf{nonce}|}, \tfrac{1}{q}\} + \mathsf{Adv}^{\mathsf{CR}}_{\mathsf{Compr},\mathcal{B}_3} \\ & + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{C},\mathcal{B}_4} + q_T \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{S},\mathcal{B}_5} \\ & + 4q_p \cdot q_C \cdot \max\{q_p, q_{\mathsf{sskid}}\} \cdot \mathsf{Adv}^{\mathsf{dPRF\text{-}ODH}}_{\mathcal{B}_{10/11}}\end{aligned}$$

where q_p is the maximum number of sub protocol executions, q_s is the maximal number of sessions, q_C is the maximal number of chips, q_T is the maximal number of terminals, $|\mathsf{nonce}|$ is the bit-length of each of the nonces r_C, r'_C, r''_C, and q is the order of the group from which (ephemeral) keys are chosen.

Proof. In the proof it is convenient to make a restriction on the adversary to make a single Test query only, and then distinguish between the cases that the now unique test session is for a chip or for a terminal, and whether it happens in an EAC execution or a 0RTT EAC execution. However, since all cases share a common structure we first exclude some general attack strategies with a game-hopping technique, starting with the original BR key secrecy.

Game 0. The original BR key secrecy game.

Game 1. As the original game, but this time abort the game, declaring the adversary to lose, if there exist two stages in chip sessions with the same session identifier. In particular, it follows that no chip sessions in mode EAC coincide for r_C, r'_C, and in mode 0RTT EAC match on r''_C.

Similar to Match security we obtain

$$\mathsf{Adv}^{G_0}_{\mathsf{KE},\mathcal{A}} \leq \mathsf{Adv}^{G_1}_{\mathsf{KE},\mathcal{A}} + q_p^2 \cdot 2^{-|\mathsf{nonces}|}.$$

Game 2. Is identical to the previous game, only that the adversary this time loses immediately, if there exist two stages in sessions of honest terminals picking the same ephemeral key epk_T or the same semi-static key pk_T^{semi}.

Similar to Match security and considering the two options for collisions, we obtain

$$\mathsf{Adv}^{G_1}_{\mathsf{KE},\mathcal{A}} \leq \mathsf{Adv}^{G_2}_{\mathsf{KE},\mathcal{A}} + 2 \cdot q_p^2 \cdot \tfrac{1}{q}.$$

Game 3. As the previous game, but this time the adversary loses if a stage in a session of an honest terminal sends $\mathsf{Compr}(epk_T)$ for some ephemeral key epk_T, but such that a session of an honest chip receives a value $epk'_T \neq epk_T$ with $\mathsf{Compr}(epk_T) = \mathsf{Compr}(epk'_T)$.

Any such incidence can be used to break the collision-resistance of Compr. To this end we construct an adversary $\mathcal{B}_3$ which runs a black-box simulation of the key exchange attack of $\mathcal{A}$ according to game G_2, generating all parameters with known secrets. If the above happens, then $\mathcal{B}_3$ can easily output such a collision. It follows that

$$\mathsf{Adv}^{G_2}_{\mathsf{KE},\mathcal{A}} \leq \mathsf{Adv}^{G_3}_{\mathsf{KE},\mathcal{A}} + \mathsf{Adv}^{\mathsf{CR}}_{\mathsf{Compr},\mathcal{B}_3}.$$

Game 4. As the previous game, but this time we abort the game with a loss for the adversary if a stage of a session of an honest party accepts a certificate for a long-term key pk_C or pk_T and identifier *certID* as valid, even though the certificate authority has not issued a certificate for that pair.

This can be straightforwardly used to mount a successful attack against the certificate scheme. That is, one builds via a black-box reduction an adversary $\mathcal{B}_4$ against the certificate scheme $\mathcal{C}$ by letting this adversary $\mathcal{B}_4$ simulate all other steps of the key exchange protocol internally, and only using the scheme's public key pk_{CVCA} and the oracle to create certificates. If there is some session where a new pair (pk, id) for *cert* is verified as valid, then we have $\mathcal{B}_5$ output the corresponding certificate as a forgery. It follows that

$$\mathsf{Adv}^{G_3}_{\mathsf{KE},\mathcal{A}} \leq \mathsf{Adv}^{G_4}_{\mathsf{KE},\mathcal{A}} + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{C},\mathcal{B}_4}.$$

Game 5. As the previous game, but this time we abort if a stage of a chip session accepts a signature for value $id_C||r_C||\mathsf{Compr}(epk_T)||pk_T^{\mathsf{semi}}$ which has not yet been signed by an honest terminal at this point.

This time we construct an adversary $\mathcal{B}_5$ against the unforgeability of the underlying signature scheme. Adversary $\mathcal{B}_5$ receives a public key pk as input and initially guesses the index of a terminal for which a forgery should happen. It creates all other parameters for the key exchange internally, but sets $pk_T = pk$ for the guessed index. Any signature creation for this terminal is performed by querying the signature oracle, all other steps can be carried out by $\mathcal{B}_5$ itself. If at some point a chip accepts a signature for a previously unsigned message, then $\mathcal{B}_5$ outputs this message-signature pair as a forgery attempt. With probability $1/q_T$ this will be for the predicted index of the terminal, such that

$$\mathsf{Adv}^{G_4}_{\mathsf{KE},\mathcal{A}} \leq \mathsf{Adv}^{G_5}_{\mathsf{KE},\mathcal{A}} + q_T \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{S},\mathcal{B}_5}.$$

Game 6. Identical to the previous game but in which the adversary makes only a single Test query and announces the stage and index of the only test session according to the order of $\mathsf{NewSession}$ queries at the beginning of its attack; if the adversary later tests a different session we declare a loss for the adversary.

A straightforward hybrid argument, guessing the index in advance, shows that this can decrease the adversary's success probability by a factor $1/q_p$ of the total number of stages in sessions. Hence, we have

$$\mathsf{Adv}^{G_5}_{\mathsf{KE},\mathcal{A}} \leq q_p \cdot \mathsf{Adv}^{G_6}_{\mathsf{KE},\mathcal{A}}.$$

Game 7. As the previous game, but this time let the adversary also announce in advance if the unique Test query is for an EAC execution or for a 0RTT EAC execution.

Since we can guess which of the two cases happens with probability at least $\frac{1}{2}$ we obtain

$$\mathsf{Adv}^{G_6}_{\mathsf{KE},\mathcal{A}} \leq 2 \cdot \mathsf{Adv}^{G_7}_{\mathsf{KE},\mathcal{A}}.$$

Game 8. As the previous game, but this time let the adversary also announce in advance a chip (and thus its long-term public key) such that a stage key of a session of the chip is tested, or partnered with a terminal stage which is tested.

Since we can guess the right chip index with with probability at least $\frac{1}{q_C}$ we obtain

$$\mathsf{Adv}^{G_7}_{\mathsf{KE},\mathcal{A}} \leq q_C \cdot \mathsf{Adv}^{G_8}_{\mathsf{KE},\mathcal{A}}.$$

Game 9. As the previous game, but this time let the adversary also announce the following in advance:

- In case it declares at the beginning to test an EAC execution (according to game G_7), the adversary announces the stage and index of a terminal session (hence determining the terminal's ephemeral key) which is tested, or partnered with the tested chip session; and

– In case it declares to test a 0RTT EAC execution, the index of semi-static key of a terminal according to the order of NewSemiStaticKey queries such that the terminal's session is tested or partnered to a tested session.

Once more, a guessing strategy yields

$$\mathsf{Adv}^{G_8}_{\mathsf{KE},\mathcal{A}} \leq \max\{q_p, q_{\mathsf{sskid}}\} \cdot \mathsf{Adv}^{G_9}_{\mathsf{KE},\mathcal{A}}.$$

Now we can make our case distinction. Note that at this point $\mathcal{A}$ makes a single Test query and announces in advance the corresponding key indices and the type of session. By the above games, if a chip session gets tested, then it has at most one (honest) partner. If it is a terminal session, it may have more than one partner, in case of a replay attack on a terminal. However, the partnered sessions all compute the same session key due to the identical session identifiers. Also, the adversary cannot make a Reveal query to either of the partnered sessions without forcing a loss. In this sense we say that each test session has at most one partner, meaning that all partners compute the same secret key.

Furthermore, if the owner of the tested session is an (at that point uncorrupt) terminal, then we may assume that the partnered chip, if existing, is not corrupt either. The reason is that, otherwise, the chip's session key would be set to revealed through the Corrupt query and the adversary would lose the game. It follows that the Diffie-Hellman key used to compute the session key is based on an uncorrupt long-term chip key pk_C and a genuine ephemeral key epk_T resp. semi-static key pk_T^{semi} of the honest terminal. The latter holds by the above game hopping, since the adversary can no longer at this point have sent an invalid epk_T (since the signature in the TA phase ranges over $\mathsf{Compr}(epk_T)$ and one can only transmit the actual epk_T in the CA phase), nor an invalid pk_T^{semi} (which is signed in clear in the TA phase). Moreover, any RevealSemiStaticKey query for pk_T^{semi} would set the session key to be revealed (and ephemeral keys cannot be leaked).

Analogously, if the owner of the tested session is an honest chip then the partnered terminal (if existing) must have honestly completed its execution, else the session identifier on the terminal side would not have been set. In particular, the contribution to the deployed Diffie-Hellman key is an authenticated key epk_T or pk_T^{semi}. Note that a subsequent corruption of the terminal does not reveal either of the two types of keys, and a RevealSemiStaticKey for the semi-static key would cause the adversary to lose anyway.

In summary, the shares of the Diffie-Hellman key in the test session (and the potentially partnered sessions) are picked by honest parties and are never revealed in the course of the attack. This allows us now to replace the key triple $(K_{\mathrm{enc}}, K_{\mathrm{mac}}, K'_{\mathrm{mac}})$ in the at most two sessions by random elements via the dPRF-ODH assumption. We start with the case that the test session is in a 0RTT EAC execution.

Case A: Test of a session key in a 0RTT EAC execution.

Game A.10. As Game 9, but in this step we replace the tuple $(K_{\text{enc}}, K_{\text{mac}}, K'_{\text{mac}})$ in the tested stage of the session and in any partnered session by the same independent random values $(\widetilde{K}_{\mathsf{enc}}, \widetilde{K}_{\mathsf{mac}}, \widetilde{K}'_{\mathsf{mac}})$.

We show that if there exists an adversary $\mathcal{A}$ that can distinguish Game 9 from Game A.10, then there necessarily exists an adversary $\mathcal{B}_{10}$ which can win the dPRF-ODH game with non-negligible advantage. The dPRF-ODH adversary $\mathcal{B}_{10}$, which will simulate the BR challenger for $\mathcal{A}$, is constructed as follows:

- Algorithm $\mathcal{B}_{10}$ first obtains a group element g^u in the dPRF-ODH game. Algorithm $\mathcal{B}_{10}$ skips the first query phase and directly asks a challenge query $\hat{x} = r''_C$ for random r''_C. It receives the dPRF-ODH challenge (g^v, y_b) where $y_0 = \mathsf{PRF}(g^{uv}, \hat{x})$ and $y_1 \xleftarrow{\$} \{0,1\}^n$ and $b \xleftarrow{\$} \{0,1\}$ are chosen uniformly at random by the challenger.
- To simulate the environment for $\mathcal{A}$, algorithm $\mathcal{B}_{10}$ must provide the long-term public keys of all participants. To this end, $\mathcal{B}_{10}$ generates keys (including pk_{CVCA}) and corresponding certificates for all users $U \in \mathcal{U}$. For the partnered chip identity which was output by $\mathcal{A}$ at the outset, algorithm $\mathcal{B}_{10}$ sets the respective public key to the previously received value g^u.
- $\mathcal{A}$ has now access to the queries specified in Definition 2. In particular, $\mathcal{A}$ may trigger the generation of semi-static keys for any terminal T via $\mathsf{NewSemiStaticKey}(T)$. Algorithm $\mathcal{B}_{10}$ uses g^v as the semi-static key of the tested session announced by $\mathcal{A}$.
- Whenever $\mathcal{B}_{10}$ then has to execute a key derivation, i.e., to answer a Reveal query of $\mathcal{A}$, or to compute or verify the final MAC with K'_{mac} —the case of a Test query is dealt with below— then $\mathcal{B}_{10}$ acts as follows. If $\mathcal{B}_{10}$ holds at least one of the secret keys for computing the Diffie-Hellman key of the session, then $\mathcal{B}_{10}$ computes the keys locally and performs the requested action.
  If it lacks knowledge of both secret keys then it must be either the case that they correspond to the keys g^v, g^u from the challenge, or one or both of the keys have been picked by the adversary $\mathcal{A}$. In the first case, if the nonce in the session also equals $\hat{x}$ then $\mathcal{B}$ uses the challenge value y_b as keys. In any other case it can query the corresponding dPRF-ODH oracle to get the actual value for the keys.
  We remark that the following inconsistency of an unpartnered session, which "accidentally" derives the same keys but where $\mathcal{B}_{10}$ falsely uses the possibly random challenge value y_b, cannot happen: If the session is not partnered then it must be because of the certificate identifier $certID_C$ or $certID_T$ and then the (verified) certificate is for a different party. In this case, however, adversary $\mathcal{B}_{10}$ has chosen the secret key itself (which happens to coincide with the challenge key) and uses the secret key to correctly compute the session keys.
- At some point, the BR adversary $\mathcal{A}$ issues a $\mathsf{Test}(\mathsf{label}, i)$ query to the BR challenger (simulated by $\mathcal{B}_{10}$). The dPRF-ODH adversary $\mathcal{B}_{10}$ forwards its previously received challenge y_b (with K'_{mac} removed) as challenge to $\mathcal{A}$.
- Eventually, $\mathcal{A}$ stops and outputs a guess b_{guess}. Algorithm $\mathcal{B}_{10}$ outputs 1 if and only if $b_{\mathsf{guess}} = b_{\mathsf{test}}$ (for the random value b_{test} chosen by $\mathcal{B}_{10}$ as part

of the attack on the key exchange protocol) as its output in the dPRF-ODH game.

Since we presumed that $\mathcal{A}$ can efficiently distinguish between Game 9 and Game A.10, adversary $\mathcal{A}$ is able to detect whether $\hat{y}_b$ is a random value or the output of a pseudorandom function. Thus, by simply running $\mathcal{A}$ and simulating all other involved parties faithfully, algorithm $\mathcal{B}_{10}$ will be able to efficiently win the dPRF-ODH game with non-negligible advantage. Hence, we can bound the advantage by

$$\mathsf{Adv}^{G_9}_{\mathsf{KE},\mathcal{A}} \leq \mathsf{Adv}^{G_{A.10}}_{\mathsf{KE},\mathcal{A}} + \mathsf{Adv}^{\mathsf{dPRF\text{-}ODH}}_{\mathcal{B}_{10}}$$

The final step is now to "undo" the replacement of the extra MAC key K'_{mac}—recall that we only aim to replace the actual session keys $K_{\text{enc}}, K_{\text{mac}}$ by random values. This requires another game hop, as before:

Game A.11. As Game A.10, but in this step we replace only the pair $(K_{\text{enc}}, K_{\text{mac}})$ in the tested stage of the session and by any partnered session by the same independent random values $(\widetilde{K}_{\mathsf{enc}}, \widetilde{K}_{\mathsf{mac}})$.

We construct another adversary $\mathcal{B}_{11}$ as in the previous game, only this time we let $\mathcal{B}_{11}$ always replace the components for K_{enc} and K_{mac} in the challenge value $\hat{y}_b$ by random values $\widetilde{K}_{\mathsf{enc}}, \widetilde{K}_{\mathsf{mac}}$. If $\hat{y}_b$ is random then we still get random values (as in Game A.10), and if y_b we now get only random entries in $K_{\text{enc}}, K_{\text{mac}}$ (as in Game A.11). It follows again by the dPRF-ODH assumption that

$$\mathsf{Adv}^{G_{A.10}}_{\mathsf{KE},\mathcal{A}} \leq \mathsf{Adv}^{G_{A.11}}_{\mathsf{KE},\mathcal{A}} + \mathsf{Adv}^{\mathsf{dPRF\text{-}ODH}}_{\mathcal{B}_{11}}$$

We recall that in this game, tested and partnered session key pairs (only) are consistently replaced by independent random values. This implies in particular, that the adversary $\mathcal{A}$ can gain no additional information on the bit b_{test} encoded in the challenge as the response to the Test query is independent of it. Thus, $\mathcal{A}$ can perform no better than to guess, i.e., $\mathsf{Adv}^{G_{A.11}}_{\mathsf{KE},\mathcal{A}} \leq 0$, which completes the proof for this case.

Case B: Test of a session key in an EAC execution. The case of a test happening in an EAC sub protocol execution is almost identical to 0RTT EAC case. The only difference is that $\mathcal{A}$ now announces the terminal's *ephemeral* key and we inject the challenge key in the dPRF-ODH assumption as an ephemeral key.

$$\mathsf{Adv}^{G_9}_{\mathsf{KE},\mathcal{A}} \leq 2 \cdot \mathsf{Adv}^{\mathsf{dPRF\text{-}ODH}}_{\mathcal{B}_{10/11}}$$

This proofs the bound for the other case and therefore the bound for the overall protocol. □

Remark 1. It may come as a surprise that the unforgeability of the MAC does not enter the security bound. This is due to the fact that we are "only" interested in key secrecy in the above theorem, stating that *at most* the intended partner can compute the session key and that seeing other session keys does not facilitate

this task. The former is ensured by the certification of the chip's long-term key and the fact that one cannot corrupt the chip, and the latter is already captured by the dPRF-ODH assumption, saying that learning related values of the PRF does not help to distinguish the challenge value from random.

Remark 2. Note that our analysis does not provide any form of key confirmation nor entity authentication. Indeed, the final MAC can be seen as providing exactly these properties [17].

5 Variations

There exist several alternatives to implement 0RTT executions. For example, the 0RTT keys may be established either in the fashion of a Diffie-Hellman key exchange or—forgoing forward secrecy— rather from pre-shared keys (derived as additional key material in the previous round). It is also interesting to investigate different ways of handling negotiation failures in the 0RTT case. In the following, we therefore present different choices for the 0RTT flow.

5.1 Diffie-Hellman Variant

The 0RTT EAC extension presented in Section 2.2 is based on a Diffie-Hellman style key agreement. Similar implementations can also be found in Google's QUIC protocol and in earlier draft versions of TLS 1.3 (draft 12 [30] and earlier).

5.2 Pre-Shared Key Variant

From draft 13 [31] onward, TLS 1.3 replaces the DH-based variant of 0RTT handshakes by a pre-shared key (PSK) alternative. The pre-shared key is established either out of band or, more commonly, in a preceding interaction between server and client. Once a full handshake has been completed, the client receives a so-called PSK identity from the server. The PSK was derived in the initial handshake and can then be used by the client to derive keys for future (0RTT) handshakes. To initiate a 0RTT handshake, the client simply incorporates the `early_data` and `pre_shared_key` extension in the `ClientHello`, followed by the application data. After the successful processing of the data, the server then responds with the `ServerHello` and a forward-secret key is then derived as in the ordinary handshake.

In principle, one could also imagine a similar approach for the EAC protocol, using the pre-shared keying material instead of the shared Diffie-Hellman key. Note, however, that this may require further changes to the EAC protocol (for the additional keying material) and that, unlike the Diffie-Hellman version, this does not provide any (terminal) forward secrecy.

5.3 Error Handling

Zero Round-Trip Time may not be supported by all servers, or there may occur errors in trying to decrypt the early data. Here we discuss how such problems are dealt with in other settings, and how one can proceed in the EAC case.

Google's QUIC Protocol. From a design perspective, all handshakes in QUIC are also 0RTT handshakes, of which some may fail. The server replies with a `ServerHello` if all necessary information to complete the handshake was contained in the preceding `ClientHello`. If this was not the case, the server sends a rejection message encompassing information that allows the client to make progress in a next handshake attempt. The type and extent of information sent along with the rejection message can be chosen individually by the server but must not prevent clients from establishing a valid handshake within a reasonable time frame.

TLS 1.3 Draft 18. Upon receiving a 0RTT handshake request with encrypted early data, the server can answer in three ways: It may either disregard the 0RTT extension and return no response, causing the client to fall back to the standard 1RTT handshake. Or it may return the empty extension, thereby signalling to the client that prior validation checks were successful and that the server intends to process the received early data. Furthermore, the server may send a `HelloRetryRequest` to the client asking it to send a `ClientHello` without the `early_data` extension.

0RTT EAC. In case of failure, we expected the client to fall back to a full EAC protocol execution consisting of terminal and chip authentication. This may seem like an expensive step in view of performance, especially if the semi-static key used by the client is simply outdated. If the terminal does not support 0RTT, fall back to full EAC is clearly inevitable.

Furthermore, we emphasize that it is in general not possible for terminals to identify outdated keys. In order for a terminal to detect this (i.e., to distinguish unknown keys from outdated keys), it must keep at least the last used value of pk_T^{semi} when updating to a new value $pk_T^{\mathsf{semi}\prime}$. Keeping state is commonly seen as not recommendable, if not infeasible, in most use cases. However, we note that a chip receives all the data it needs to initiate future 0-RTT handshakes with a 0RTT-supporting terminal during the terminal authentication phase of the EAC protocol. Therefore, it is sufficient for the chip to carry out the TA phase before the 0RTT handshake can be re-tried. In light of this, it is also conceivable for terminals to proceed similarly to the mechanism deployed in the QUIC protocol and to reply with the current authenticated semi-static key, i.e., to send $cert_T, pk_T^{\mathsf{semi}}, s_T$ where $s_T \leftarrow \mathsf{Sig}(sk_T, pk_T^{\mathsf{semi}})$.

6 Conclusion

The extended access control (EAC) protocol is an omnipotent solution for key establishment between two parties. In this work, we presented a 0RTT mode for the EAC protocol which allows to reduce the latency of recurring connections. It is noteworthy that this 0RTT mode can be added as an extension with minimal changes to the original protocol. We further showed that EAC+0RTT can be proven secure in the multi-stage setting of the Bellare-Rogaway model.

Thus, the modified protocol still achieves the common security properties of an authenticated key exchange protocol.

Acknowledgments

This work has been [co-]funded by the DFG as part of project D.2 within the RTG 2050 "Privacy and Trust for Mobile Users", as well as part of project S4 within the CRC 1119 CROSSING.

References

1. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) Advances in Cryptology – CRYPTO'93. Lecture Notes in Computer Science, vol. 773, pp. 232–249. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 22–26, 1994) (Cited on pages 1 and 8.)
2. Bender, J., Dagdelen, Ö., Fischlin, M., Kügler, D.: Domain-Specific Pseudonymous Signatures for the German Identity Card, pp. 104–119. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-33383-5_7 (Cited on page 4.)
3. Bender, J., Dagdelen, Ö., Fischlin, M., Kügler, D.: The PACE|AA Protocol for Machine Readable Travel Documents, and Its Security, pp. 344–358. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-32946-3_25 (Cited on page 4.)
4. Bender, J., Fischlin, M., Kügler, D.: Security Analysis of the PACE Key-Agreement Protocol, pp. 33–48. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04474-8_3 (Cited on page 4.)
5. Bender, J., Fischlin, M., Kügler, D.: The PACE|CA Protocol for Machine Readable Travel Documents, pp. 17–35. Springer International Publishing, Cham (2013), http://dx.doi.org/10.1007/978-3-319-03491-1_2 (Cited on page 4.)
6. Brzuska, C.: On the Foundations of Key Exchange. Ph.D. thesis, Technische Universität Darmstadt, Darmstadt, Germany (2013), http://tuprints.ulb.tu-darmstadt.de/3414/ (Cited on page 13.)
7. Brzuska, C., Fischlin, M., Warinschi, B., Williams, S.C.: Composability of Bellare-Rogaway key exchange protocols. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) ACM CCS 11: 18th Conference on Computer and Communications Security. pp. 51–62. ACM Press, Chicago, Illinois, USA (Oct 17–21, 2011) (Cited on page 13.)
8. BSI (Bundesamt für Sicherheit in der Informationstechnik, Federal Office for Information Security): Technical Guideline TR-03110: Advanced Security Mechanisms for Machine Readable Travel Documents: Extended Access Control (EAC), Password Authenticated Connection Establishment (PACE), and Restricted Identification (RI. BSI-TR-03110 (2008), version 2.0 (Cited on page 1.)
9. BSI (Bundesamt für Sicherheit in der Informationstechnik, Federal Office for Information Security): Technical Guideline TR-03110-2: Advanced Security Mechanisms for Machine Readable Travel Documents and eIDAS Token, Part 2, Protocols for electronic IDentification, Authentication and trust Services (eIDAS). BSI-TR-03110 (2015), version 2.2 (Cited on page 1.)
10. Cheng, Y., Chu, J., Radhakrishnan, S., Jain, A.: TCP Fast Open (December 2014), RFC 7413, Internet Engineering Task Force (IETF) (Cited on page 2.)

11. Cooper, D., Ferraiolo, H., Mehta, K., Francomacaro, S., Chandramouli, R., Mohler, J.: Interfaces for Personal Identity Verification – Part 1: PIV Card Application Namespace, Data Model and Representation (May 2015) (Cited on page 3.)
12. Coron, J.S., Gouget, A., Icart, T., Paillier, P.: Supplemental Access Control (PACE v2): Security Analysis of PACE Integrated Mapping, pp. 207–232. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-28368-0_15 (Cited on page 4.)
13. Dagdelen, Ö., Fischlin, M.: Security Analysis of the Extended Access Control Protocol for Machine Readable Travel Documents. In: Information Security - 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6531, pp. 54–68. Springer (2010) (Cited on pages 1, 4, 6, and 7.)
14. Dagdelen, Ö., Fischlin, M., Gagliardoni, T., Marson, G.A., Mittelbach, A., Onete, C.: A cryptographic analysis of OPACITY - (extended abstract). In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013: 18th European Symposium on Research in Computer Security. Lecture Notes in Computer Science, vol. 8134, pp. 345–362. Springer, Heidelberg, Germany, Egham, UK (Sep 9–13, 2013) (Cited on page 3.)
15. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In: Ray, I., Li, N., Kruegel:, C. (eds.) ACM CCS 15: 22nd Conference on Computer and Communications Security. pp. 1197–1210. ACM Press, Denver, CO, USA (Oct 12–16, 2015) (Cited on page 13.)
16. Fischlin, M., Günther, F.: Multi-stage key exchange and the case of Google's QUIC protocol. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 14: 21st Conference on Computer and Communications Security. pp. 1193–1204. ACM Press, Scottsdale, AZ, USA (Nov 3–7, 2014) (Cited on pages 4, 8, 9, 10, and 13.)
17. Fischlin, M., Günther, F., Schmidt, B., Warinschi, B.: Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In: 2016 IEEE Symposium on Security and Privacy. pp. 452–469. IEEE Computer Society Press, San Jose, CA, USA (May 22–26, 2016) (Cited on page 24.)
18. Gilson, B., Baldridge, T.: PKI (CAK) — Enabled PACS with PIV Card: PACS Lessons Learned and Need for Speed. http://csrc.nist.gov/groups/SNS/piv/fips_201-2_march_2015/day_one/gilson_baldridge_piv-cak_enabled_pacs_fips201-2_2015.pdf (May 2015), presentation at FIPS 201-2 Supporting Special Publications Workshop (Cited on page 3.)
19. Google: QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic (2016) (Cited on page 2.)
20. Hale, B., Jager, T., Lauer, S., Schwenk, J.: Speeding: On low-latency key exchange. Cryptology ePrint Archive, Report 2015/1214 (2015), http://eprint.iacr.org/2015/1214 (Cited on page 4.)
21. Hanzlik, L., Krzywiecki, Ł., Kutyłowski, M.: Simplified PACE|AA Protocol, pp. 218–232. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-38033-4_16 (Cited on page 4.)
22. Hanzlik, L., Kutyłowski, M.: Restricted identification secure in the extended canetti-krawczyk model. Journal of Universal Computer Science 21(3), 419–439 (2015) (Cited on page 4.)
23. ICAO: Machine Readable Travel Documents, Part 11, Security Mechanisms for MRTDs. Doc 9303 (2015), seventh Edition (Cited on page 1.)
24. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology

– CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 273–293. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2012) (Cited on page 16.)
25. Krawczyk, H., Paterson, K.G., Wee, H.: On the security of the TLS protocol: A systematic analysis. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology – CRYPTO 2013, Part I. Lecture Notes in Computer Science, vol. 8042, pp. 429–448. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 18–22, 2013) (Cited on page 16.)
26. Kutyłowski, M., Krzywiecki, Ł., Kubiak, P., Koza, M.: Restricted Identification Scheme and Diffie-Hellman Linking Problem, pp. 221–238. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-32298-3_15 (Cited on page 4.)
27. Morgner, F., Bastian, P., Fischlin, M.: Attribute-based Access Control Architectures with the eIDAS Protocols. In: SSR 2016: Security Standardisation Research. Lecture Notes in Computer Science, Springer Verlag (2016) (Cited on page 2.)
28. Morgner, F., Bastian, P., Fischlin, M.: Securing Transactions with the eIDAS Protocols. In: 10th WISTP International Conference on Information Security Theory and Practice. Lecture Notes in Computer Science, Springer Verlag (2016) (Cited on page 2.)
29. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3 – draft-ietf-tls-tls13-18. https://tools.ietf.org/html/draft-ietf-tls-tls13-18 (October 2016) (Cited on page 2.)
30. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3, draft-ietf-tls-tls13-12. https://tools.ietf.org/html/draft-ietf-tls-tls13-12 (March 2016) (Cited on pages 2, 3, and 24.)
31. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3, draft-ietf-tls-tls13-13. https://tools.ietf.org/html/draft-ietf-tls-tls13-13 (May 2016) (Cited on pages 2 and 24.)
32. Smart Card Alliance: Industry Technical Contributions: OPACITY. http://www.smartcardalliance.org/smart-cards-contributions-opacity/ (Cited on page 3.)