Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3

Marc Fischlin Felix Günther
Cryptoplexity, Technische Universität Darmstadt
Darmstadt, Germany
marc.fischlin@cryptoplexity.de, guenther@cs.tu-darmstadt.de

Benedikt Schmidt
IMDEA Software Institute
Madrid, Spain
benedikt.schmidt@imdea.org

Bogdan Warinschi
University of Bristol
Bristol, UK
csxbw@bristol.ac.uk

***Abstract*—Key exchange protocols allow two parties at remote locations to compute a shared secret key. The common security notions for such protocols are secrecy and authenticity, but many widely deployed protocols and standards name another property, called key confirmation, as a major design goal. This property should guarantee that a party in the key exchange protocol is assured that another party also holds the shared key. Remarkably, while secrecy and authenticity definitions have been studied extensively, key confirmation has been treated rather informally so far.**

In this work, we provide the first rigorous formalization of key confirmation, leveraging the game-based security framework well-established for secrecy and authentication notions for key exchange. We define two flavors of key confirmation, full and almost-full key confirmation, taking into account the inevitable asymmetry of the roles of the parties with respect to the transmission of the final protocol message. These notions capture the strongest level of key confirmation reasonably expectable for the two communication partners of the key exchange.

We demonstrate the benefits of having precise security definitions for key-confirmation by applying them to the next version of the Transport Layer Security (TLS) protocol, version 1.3, currently developed by the Internet Engineering Task Force (IETF). Our analysis shows that the full handshake as specified in the TLS 1.3 draft `draft-ietf-tls-tls13-10` achieves desirable notions of key confirmation for both clients and servers. While key confirmation is generally understood and in the TLS 1.3 draft described as being obtained from the `Finished` messages exchanged, interestingly we can show that the full TLS 1.3 handshake provides key confirmation even without those messages, shedding a formal light on the security properties different handshake messages entail.

We further demonstrate the usefulness of rigorous definition by revisiting a folklore approach to establish key confirmation (as discussed for example in SP 800-56A of NIST). We provide a formalization as a generic protocol transformation and show that the resulting protocols enjoy strong key confirmation guarantees, thus confirming its beneficial use in both theoretical and practical protocol designs.

I. Introduction

Key exchange is one of most widely deployed cryptographic protocols to date, bootstrapping confidential and authenticated data exchange in virtually any secure communication protocol. The most basic security properties are key secrecy and entity authentication: the former property guarantees that no other party learns information about the key whereas the later ensures that the key is shared with the intended partner. The seminal work of Bellare and Rogaway [4] provided rigorous security definitions for these two notions and thus grounded the design and analysis of such protocols on solid foundations. The huge body of follow-up work refined, extended, and applied their results. These include extensions to the asymmetric setting [8], more refined attacker models [14], dealing with different variants of entity authentication [28], etc. There is also a large body of work, e.g., [22], [24], [7], [5], [18], applying and extending these results to real-world security protocols such as TLS [17] or SSH [32].

A. Key Confirmation

An intuitively desirable security property that has so far escaped a rigorous treatment is *key confirmation*: the idea that when a party accepts locally a key, it has the guarantee that some other party has precisely the same key. The property is often mentioned in scientific papers on the subject of key exchange [3], [23], [25] but the typical reference for a definition is the "Handbook of Applied Cryptography" [27, Definition 12.7] which describes key confirmation as the property

> "whereby one party is assured that a second (possibly unidentified) party actually has possession of a particular secret key."

Other references include the refinement proposed by Blake-Wilson and Menezes [9], [10] who further distinguish between *explicit* key confirmation, where one party is assured that the other party holds the key, and *implicit* key confirmation, where the other party can compute the key.

One may speculate that the reason for a lack of rigorous definitions is that absence of key confirmation does not seem to open parties to attacks: a party may send messages encrypted with an (unconfirmed) key which no-one can decrypt. This may be a waste of resources but not an obvious security risk. Another possibility is that it might seem "clear" when a protocol has key confirmation, so a formal definition appears to be an overkill. For example, protocols like TLS 1.2 [17] and EMV [20] utilize the derived key during the execution of the protocol, so receiving a a message encrypted with the shared key provides key-confirmation assurances.

This level of informal understanding is also reflected by other folklore protocol transformations that can boost a secure key exchange protocol to also provide key confirmation

DOI 10.1109/SP.2016.34

guarantees. A popular proposal (which we refer to as "refresh-then-MAC") is to extend a key exchange protocol as follows: use the established key first to derive two additional keys; the first will be set as the session key whereas the latter will be used to compute a message authentication code (MAC) value over the transcript of the protocol so far. Exchanging valid MACs should then guarantee that parties have also locally computed the associated session key.

The status of key confirmation as an important security property is still unclear. On the one hand, some practitioners seem to be convinced that key confirmation messages (messages that use the key derived to ensure that the parties have agreed on the same key) do improve authentication[1], yet others struggle to understand the benefits that key confirmation messages bring to protocols[2]. Nevertheless, many security protocol specifications name key confirmation as an explicit goal to achieve, including, e.g., the recommendations for key establishment schemes by NIST [2], [1], [21], or the draft specifications for the next TLS version 1.3 [29].

This paper is motivated by the observation that any serious discussion of this issue is moot in the absence of clear security definitions and that such definitions should also ground design decisions and security analysis which nowadays simply rely on the type of lore outlined above. To support the argument, note that the recommendation of NIST concerning key confirmation in [2], [1] basically follows the informal definition above, ignoring that one of the protocol participants must accept first and thus gets a different strength of confirmation guarantee than the one accepting last.

B. Our Results

SECURITY DEFINITIONS. We propose security definitions (in Section III) that aim to capture the established intuition behind key confirmation. First, we note that we do not attempt to distinguish between explicit and implicit key confirmation: distinct computational interpretation to "has the key" and "can compute the key" seem difficult to provide. This follows the line of reasoning by Blake-Wilson and Menezes [9] who argue that "for all practical purposes, the assurances [of implicit and explicit key confirmation] are in fact the same," especially since one cannot guarantee that a party forgets a key between its derivation and its first time usage.

Second, it is clear that key confirmation guarantees are asymmetric: the party that receives the last message obtains the stronger guarantees; such guarantees do not hold for the party that sends the last message since the message can be dropped by an adversary. Accordingly, we distinguish between *full key confirmation* and *almost-full key confirmation*. The former property guarantees that when a party accepts a key, there exists some other party that has already accepted precisely that key. The latter property ensures that when a party accepts a key, there is some session which, *if it accepts*, then it accepts the same key. Formalizing sound key confirmation notions turns out to be more challenging than one might expect given the common informal understanding: As we explain later in the paper, although we rely on compelling intuition, the definitions need to be carefully crafted to avoid some potential pitfalls which we outline.

A prime feature of our definitions is modularity. Following Blake-Wilson and Menezes [9] as well as NIST [2], [1], we chose to disentangle key confirmation from the other security concerns specific to key exchange. In particular, our notion only ensures that there exists *some party* that accepted the same key: the notion does not guarantee that it is the expected communication partner. However, the desired property follows by combining key confirmation and implicit key authentication (i.e., classical key secrecy with mutually authenticating parties [4]): a protocol with both these properties has explicit key authentication. Informally, key confirmation can be interpreted as guaranteeing the lower bound that "at least one other (unspecified) party holds the key" whereas implicit key authentication ensures the upper bound that "at most one (namely the expected) party holds the key." Together, the notions entail explicit key authentication: "exactly the expected party holds the key." Note that the definitional modularity also allows us to "swap" the key confirmation steps and property in and out, depending on the protocol's security requirements (as in the recommendations of NIST), and to independently argue about this additional security feature.

APPLICATION TO TLS. We use the rigorous security models that we develop to shed light on the key-confirmation properties of the recent TLS 1.3 draft `draft-ietf-tls-tls13-10` [29] (short: `draft-10`) in Section IV. As in previous TLS versions, TLS 1.3 `draft-10` leverages `Finished` messages essentially consisting of a message authentication code (MAC) computed over the transcript of the key exchange and sent both by the client and the server. It is hence not surprising that our analysis confirms that (the full, (EC)DHE handshake of) TLS 1.3 indeed achieves the strongest expectable key confirmation guarantees, i.e., full key confirmation for the server (which accepts after the client) and almost-full key confirmation for the client (which accepts first).

Perhaps surprisingly, we show that key confirmation does not (necessarily) rely on the `Finished` messages exchanged, but can actually be shown to hold even in a shortened variant of the full `draft-10` handshake which omits these messages. This becomes possible due to the `CertificateVerify` messages sent in the full `draft-10` handshake, which are essentially an online signature under the parties' long-term secret signing keys over (the hash of) all messages exchanged (i.e., the transcript or "session hash", as denoted in TLS 1.3).

This result deepens the understanding of the far-reaching security guarantees achievable with the session hash concept (originally introduced to counter the triple handshake attack [6] in TLS 1.2) and online signatures. While it

[1]Adam Langley publicly proclaimed at the Real World Cryptography Workshop (RWC) 2014: "Key-confirmation messages are here to stay." [26]

[2]See the discussion on the TLS mailing list [31] on removing the confirmation message from the design of TLS.

might at first glance seem to open up a discussion of whether `Finished` messages become obsolete in presence of `CertificateVerify` messages already establishing key confirmation, we remark that TLS 1.3 `draft-10` provisions further, abbreviated handshake variants which omit the `CertificateVerify` messages for performance reasons and fully rely on the `Finished` messages for key confirmation as well as authentication.

GENERIC CONSTRUCTION. As explained above, one idea used to obtain key confirmation deployed in existing protocols is to somehow explicitly involve the key in the operations of the key-exchange protocol. This message plays a double role: on the one hand the MAC "ties" together the messages that belong to one session. On the other hand, the message is sent over the channel that is being established, or in other words, it is encrypted with the session key: receiving this message would therefore show that the other party already holds the key.

It is by now well-known that unmitigated use of the session key (for either encrypting or MAC-ing) immediately destroys key secrecy, and better transformations have been proposed and used in protocol design. For example, Chen and Kudla study the impact that such a transformation has on a particular protocol, but still need to rely on intuition to conclude that the resulting protocol satisfies key confirmation [15]. With precise definitions in place, we are in a position where these proposals can be accurately analyzed.

We deploy our rigorous notions to analyze the popular "refresh-then-MAC" transformation (also recommended by NIST [2], [21]) and confirm that the intuition behind the construction is indeed correct: when applied to a key-exchange protocol that ensures key secrecy the transformation yields a protocol which, in addition, also satisfies key confirmation (and preserves (implicit) authentication). Interestingly, as a by-product of our formalism, we can now also show that the simpler version where one sends some additionally derived key material in clear, without computing a MAC, would serve the same purpose and already provide key confirmation.

II. KEY EXCHANGE PROTOCOLS AND THEIR SECURITY

In this section we define key exchange protocols and their security, following essentially the approach of Bellare and Rogaway [4]. We recall the basic security properties of key secrecy, i.e., that session keys look random, and Match security as defined in [13], [12], which guarantees soundness of session partnering (e.g., that a successful session has a unique partner session).

In the model we will introduce the notion of a *key-confirmation identifier* kcid, pivotal for our formalization of almost-full key confirmation. Essentially, once set, the identifier kcid ensures that the party will eventually derive the same key as any other party with that identifier, even though the party has not accepted and the session identifier sid has not been set yet. In other words, one may interpret the setting of the key-confirmation identifier as stating that, upon receiving the partner's confirmation message, a session will have enough information to compute the (same) key. We elaborate on the choices for setting key-confirmation identifiers in a protocol further when introducing the notion of almost-full key confirmation that relies on them.

A. Protocol Syntax

We consider two-party protocols where participating parties belong to either a set of clients $\mathcal{C}$ or a set of servers $\mathcal{S}$. Each set has associated a long-term key generation algorithm $\mathsf{KG}_{\mathtt{client}}$ resp. $\mathsf{KG}_{\mathtt{server}}$ (one of these algorithms can be trivial, for the case when a set of parties does not have long term keys). We let $\mathcal{I} = \mathcal{C} \cup \mathcal{S}$ denote the set of all identities in the system.

Our focus is on key exchange protocols which are defined by an (interactive) program Π that parties execute locally. Between invocations, the program maintains a state $\mathsf{st} = (\mathsf{crypt}, \mathsf{status}, \mathsf{role}, \mathsf{id}, \mathsf{pid}, \mathsf{sid}, \mathsf{kcid}, \mathsf{key})$, where the different components are as follows:

- $\mathsf{crypt} \in \{0,1\}^*$ is some protocol-specific state, e.g., secret Diffie-Hellman values. It will be set to the participants cryptographic keys upon initialization.
- $\mathsf{status} \in \{\mathtt{accept}, \mathtt{reject}, \perp\}$ is a variable that indicates the status of the key-exchange phase. Initially, $\mathsf{status} = \perp$ and may change to `accept` in accepted executions, or `reject` in runs in which the party rejects. We assume that once the status has been set to `accept` or `reject`, the value does not change anymore and the adversary immediately learns the value of status.
- $\mathsf{role} \in \{\mathtt{client}, \mathtt{server}\}$ is the role of the participant. If $\mathsf{id} \in \mathcal{C}$ then $\mathsf{role} = \mathtt{client}$ and if $\mathsf{id} \in \mathcal{S}$ then $\mathsf{role} = \mathtt{server}$.
- $\mathsf{id} \in \mathcal{I}$ is the identity of the party that "owns" this session.
- pid is a partner identifier which is assigned a value in $\mathcal{I} \cup \{\perp, *\}$ once, where initially $\mathsf{pid} = \perp$. We use $*$ to indicate an unspecified identity for the case when that party does not have long term keys, and use $\mathsf{pid} = i$ for some $i \in \mathcal{I}$, otherwise[3].
- $\mathsf{sid} \in \{0,1\}^* \cup \{\perp\}$ is a session identifier, initialized to $\perp$ and then set upon changing to accepting status $\mathsf{status} = \mathtt{accept}$. Once it is set to some string, sid cannot be changed anymore. We assume throughout the paper that session identifiers are public in the sense that they are determined by the incoming and outgoing messages for that sessions.
- $\mathsf{kcid} \in \{0,1\}^* \cup \{\perp\}$ is a so-called key-confirmation identifier, initialized to $\perp$ and usually set at some point during the execution.
- $\mathsf{key} \in \{0,1\}^* \cup \{\perp\}$ is the key locally derived in this session, also called session key. Here, key may consist of multiple elements, e.g., one key for encryption and one key for authentication. Initially the session key is set to $\perp$. We assume that a key is set upon changing to the accepting status $\mathsf{status} = \mathtt{accept}$.

[3]We work therefore in the post-specified peer setting where, as is the case of TLS, the owner of a potentially partnered session is determined on the fly. The pre-specified peer setting can be easily obtained by demanding that pid is set as soon as a session starts its execution.

Formally, the program Π is a function which takes a state st as above, an input message m, and returns a new state st' and a message m'. Given some state st and a message m we write $(\mathsf{st}', m') \leftarrow \Pi(\mathsf{st}, m)$ for one step in the execution of the protocol that processes message m to yield a new state st' and an answer m'.

B. Security Model

We first describe the execution model in which the adversary interacts with the participants, all running protocol Π.

EXECUTION MODEL. Initially, all parties $i \in \mathcal{I}$ generate long-term keys by running either $\mathsf{KG}_{\texttt{client}}$ or $\mathsf{KG}_{\texttt{server}}$, depending on their role.[4] They store the secret keys sk_i for further use, and the adversary gets to learn the public keys pk_i of all participants $i \in \mathcal{I}$.

As usual, we assume that the adversary $\mathcal{A}$ controls the communication network. The adversary runs an execution of multiple instances of protocol Π, starting a new session of party $i \in \mathcal{I}$ by calling a $\mathsf{NewSession}$ oracle for i. This immediately creates a globally unique administrative label ℓ, and a freshly initialized state st for the party. We denote the individual components of that session by $\ell.\mathsf{crypt}$, $\ell.\mathsf{status}$, $\ell.\mathsf{role}$ etc. Note that this sets $\ell.\mathsf{id} = i$ and $\ell.\mathsf{role}$ accordingly, and all other components are initialized as described above. The adversary can now interact with this new session Π_i^ℓ of the protocol Π via the label ℓ. We note that different options are possible for how the intended peer of the session is specified. The adversary can specify $\ell.\mathsf{pid}$ upon session creation (yielding the pre-specified peer setting) or leave it undefined (yielding the post-specified peer setting).

The adversary may deliver messages to sessions using queries of the form $\mathsf{Send}(\ell, m)$, where ℓ is a session label (of a session which has been initialized before), and m is an arbitrary message. The session executes $(\mathsf{st}', m') \leftarrow \Pi(\ell.\mathsf{st}, m)$, sets the local session state to st', and returns m' to the adversary. We assume that the adversary also learns if the session changes its status $\ell.\mathsf{status}$ to `accept` or `reject`.

The adversary $\mathcal{A}$ may corrupt the long-term key of user i by issuing a query $\mathsf{Corrupt}(i)$. The key sk_i is returned to the adversary and identity i is added to an initially empty set Corr of corrupt parties. From now on, the adversary cannot call oracles related to that party anymore. We call parties in Corr corrupt, whereas all parties still in $\mathcal{I} \setminus \mathrm{Corr}$ are called honest. We define the set $\mathrm{Corr}_{\mathsf{Cl}}$ of corrupted clients and the set $\mathrm{Corr}_{\mathsf{Sv}}$ of corrupted servers as the intersections of Corr with $\mathcal{C}$ and $\mathcal{S}$.

The adversary may learn the session key via query $\mathsf{RevealKey}(\ell)$. This returns the session key $\ell.\mathsf{key}$ to the adversary where, potentially, the key key is still undetermined yet and equal to $\perp$.

Finally, for capturing key secrecy we give the oracle access to a Test oracle. This oracle is initialized with a secret bit $b \leftarrow_{\$} \{0, 1\}$. Upon querying the oracle with some label ℓ it returns $\perp$ to the adversary if $\ell.\mathsf{status} \neq \texttt{accept}$; otherwise it returns $\ell.\mathsf{key}$ if $b = 0$, or a fresh random key chosen from some distribution according to the protocol specification if $b = 1$. The task of the adversary is to determine b. We assume for simplicity that the adversary can query the Test oracle only once. This can be extended to the case of multiple queries and security follows from a hybrid argument for public session identifiers (with some careful consistency stipulations).

PARTNERING. A crucial ingredient for capturing security of key exchange protocols is the notion of *partnering*. Roughly speaking, partnering formalizes which sessions of the protocol intend to communicate with each other. This notion is necessary to identify trivial attacks in which the adversary reveals a session key of the intended partner of a tested session. We follow the paradigm of [3] and define partnering via session identifiers sid, set by each party in the course of the protocol, when the status changes to `accept`.

We now say that two distinct sessions (oracles) in an execution are *partnered* if their local session identifier variables have the same value (different from $\perp$). Formally, we use a predicate $\mathsf{Partners}$ which has as input two sessions labels ℓ and ℓ' and evaluates to true if and only if $\ell.\mathsf{sid} = \ell'.\mathsf{sid} \neq \perp \wedge \ell \neq \ell'$.

Definition 2.1 (Partnered sessions): Two sessions with labels ℓ, ℓ' are partnered in an execution if $\mathsf{Partners}(\ell, \ell') = \mathsf{true}$.

Note that, since session identifiers can be set only once, two sessions which are partnered in an execution always remain partnered.

FRESHNESS. We need to exclude trivial attacks in which the adversary tests the key of a session and reveals the key of a partnered session. Since we will later demand that identical session identifiers imply identical keys, the partnered session will hold the same key and the adversary could easily distinguish the tested key from random. To identify the sessions where testing is still admissible we use the freshness predicate $\mathsf{Fresh}(\ell)$. This predicate is evaluated at the end of the execution and yields true if and only if all of the following conditions hold:

1) $\ell.\mathsf{status} = \texttt{accept}$, i.e., the test session has accepted at some point.
2) The adversary has not issued a query $\mathsf{RevealKey}(\ell)$ to the test session during the execution.
3) $\ell.\mathsf{id} \notin \mathrm{Corr}$, i.e., the owner of the test session has not been corrupted in the execution.
4) For any ℓ' such that $\mathsf{Partners}(\ell, \ell') = \mathsf{true}$ the adversary has not issued a query $\mathsf{RevealKey}(\ell')$; no partner of the tested session has been asked to reveal the session key.
5) $\ell.\mathsf{pid} \notin \mathrm{Corr}$ and $\ell.\mathsf{pid} \neq *$, i.e., the session has not been partnered with a party (potentially) controlled by the adversary.

Definition 2.2 (Freshness): A session with label ℓ is *fresh* if $\mathsf{Fresh}(\ell) = \mathsf{true}$.

Note that the way we define the freshness property indicates that we are not concerned with forward secrecy. That is, no matter when the party of the session (or a partner) has been

[4] Note that we allow, e.g., that client long-term keys are empty or remain unused to model that clients may not possess long-term secret keys.

corrupted, it is assumed that this endangers the security of the session key, even for sessions which have already been completed.

AUTHENTICATION. Authentication guarantees the identity of the partner for some session. We consider both one-way and mutual authentication which we define formally using the asymmetric predicate $\mathsf{auth}(\ell, \ell')$. Informally, the predicate says that session ℓ authenticates the owner of session ℓ' (as its intended partner). Formally, we define $\mathsf{auth}(\ell, \ell') = \mathsf{true}$ if and only if $\ell.\mathsf{pid} \neq *$ and $\ell.\mathsf{pid} = \ell'.\mathsf{id}$. The former basically demands that some authentication took place and the latter requires that it points to the intended partner. Mutual authentication is then expressed as $\mathsf{mauth}(\ell, \ell') = \mathsf{auth}(\ell, \ell') \wedge \mathsf{auth}(\ell', \ell)$.

We furthermore say a protocol is unilaterally or mutually authentication if any partnered sessions correctly authenticate the server resp. both sides. Note that we however do not formalize (implicit) authentication as a distinguished security experiment, but follow the common approach to encapsulate implicit authentication within key secrecy.

Definition 2.3 ((Implicit) Authentication): A session ℓ unilaterally authenticates a partnered session ℓ' if $\mathsf{auth}(\ell, \ell') = \mathsf{true}$. The sessions ℓ, ℓ' authenticate mutually if $\mathsf{mauth}(\ell, \ell') = \mathsf{true}$.

A key exchange protocol Π provides unilateral resp. mutual authentication if for all partnered sessions ℓ, ℓ' with $\ell.\mathsf{role} = \mathtt{client}$ and $\ell'.\mathsf{role} = \mathtt{server}$ it holds that $\mathsf{auth}(\ell, \ell') = \mathsf{true}$ resp. $\mathsf{mauth}(\ell, \ell') = \mathsf{true}$.

Notice that our notion of freshness above is general enough to deal with the interplay between authentication and key secrecy, in that it immediately captures the case of *unilateral authentication* when the partner is anonymous (which we model by setting $\mathsf{pid} = *$).

As explained earlier, our model seamlessly treats pre- and post-specified partners.

C. Traditional Security Properties

Before we move on to key confirmation, which is the security notion that is the focus of this work, we recall two notions, crucial for the security of key exchange protocols. One is key secrecy, which basically requires that (fresh) keys look random and are only available to the (implicitly) authenticated partners; the other one is called Match security in [13] and, as a counter balance ensuring soundness of session partnering, captures for example functional properties such as that partnered sessions derive the same keys, and security properties such as quasi uniqueness of pairs of session identifiers.

KEY SECRECY. Key secrecy demands that any efficient adversary cannot do significantly better than guessing in distinguishing actual keys (of fresh sessions) from random.

Definition 2.4 (Key secrecy): A key exchange protocol Π provides key secrecy if for any PPT adversary $\mathcal{A}$ there exists a negligible function $\mathrm{negl}(n)$ such that

$$\Pr\left[\mathbf{Exp}^{\mathsf{scrcy}}_{\Pi,\mathcal{A}}(n) = 1\right] \leq \frac{1}{2} + \mathrm{negl}(n)$$

for the key secrecy experiment in Figure 1.

Match SECURITY. Next we define Match security. For this we use a symmetric predicate $\mathsf{samekey}(\ell, \ell')$ which is set to true if and only if $\ell.\mathsf{key} = \ell'.\mathsf{key}$.

Define the predicate Match which returns true if and only if all of the following conditions holds.

1) For all sessions ℓ, ℓ' with $\mathsf{Partners}(\ell, \ell') = \mathsf{true}$, it holds that $\mathsf{samekey}(\ell, \ell') = \mathsf{true}$, i.e., partnered sessions derive the same key.
2) For all sessions ℓ, ℓ', ℓ'' with $\mathsf{Partners}(\ell, \ell') = \mathsf{true}$ and $\mathsf{Partners}(\ell, \ell'') = \mathsf{true}$, it holds that $\ell' = \ell''$, i.e., there is at most one partnered session for each session.
3) For all sessions ℓ, ℓ' with $\mathsf{Partners}(\ell, \ell') = \mathsf{true}$, it holds that $\ell.\mathsf{role} \neq \ell'.\mathsf{role}$, i.e., two partnered sessions adopt the client-server relationship.
4) For all ℓ, ℓ' with $\mathsf{Partners}(\ell, \ell') = \mathsf{true}$, it holds that $\mathsf{auth}(\ell, \ell')$ or $\ell.\mathsf{pid} = *$, i.e., the partnered session has the intended owner (where $\ell.\mathsf{pid} = *$ allows for an arbitrary owner).[5]

The adversary's attack on Match-security is now similar to the one against key secrecy, only this time the Test oracle disappears from the setting, and the experiment finally evaluates the predicate Match on the resulting execution state. The adversary wins if the predicate evaluates to false, implying that the adversary has managed to create a state which violates the Match-security properties. Since we use a similar structure for our key confirmation experiments, where only a different predicate is evaluated, it is convenient to define the experiment generically with an abstract predicate Pred and plug in the corresponding predicate for the security notion in question.

*Definition 2.5 (*Match *security):* A key exchange protocol Π provides Match security if for any PPT adversary $\mathcal{A}$ there exists a negligible function $\mathrm{negl}(n)$ such that

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}}_{\Pi,\mathcal{A}}(n) = 1\right] \leq \mathrm{negl}(n)$$

for the generic security experiment in Figure 2 with the predicate set to $\mathsf{Pred} = \mathsf{Match}$.

III. DEFINING KEY CONFIRMATION

In this section we discuss and develop our notions for key confirmation. Per the discussion in the introduction we do not distinguish between explicit and implicit key confirmation.

We design two security notions which capture strong forms of key confirmation, one that corresponds to the guarantees of the party that receives the last message in the protocol, and a second one for the party that sends this last message. We give logical formulas that directly capture the basic intuition behind key confirmation and then turn the formulas into their

[5]The difference to unilaterally or mutually authenticated protocols is that we generally allow for $\ell.\mathsf{pid} = *$ here. This is to require soundness of the pid set; authentication requirements follow from Definition 2.3.

Experiment $\mathbf{Exp}^{\mathsf{scrcy}}_{\Pi,\mathcal{A}}(n)$

1 : **foreach** $i \in \mathcal{I}$ **do**
2 : **if** $i \in \mathcal{C}$ **then** $(sk_i, pk_i) \leftarrow \mathsf{KG}_{\mathtt{client}}(1^n)$ **fi**
3 : **if** $i \in \mathcal{S}$ **then** $(sk_i, pk_i) \leftarrow \mathsf{KG}_{\mathtt{server}}(1^n)$ **fi**
4 : **endforeach**
5 : $b \leftarrow\!\!\$\ \{0,1\}$
6 : $pks \leftarrow \{(i, pk_i) \mid i \in \mathcal{I}\}$
7 : $b' \leftarrow \mathcal{A}^{\mathsf{NewSession}(\cdot),\mathsf{Send}(\cdot,\cdot),\mathsf{Test}(b,\cdot),\mathsf{Corrupt}(\cdot),\mathsf{RevealKey}(\cdot)}(1^n, pks)$
8 : **return** $(b = b' \wedge \mathsf{Fresh}(\ell_{\text{test}}))$ / where ℓ_{test} only Test-query

Figure 1: Key Secrecy Experiment.

Experiment $\mathbf{Exp}^{\mathsf{Pred}}_{\Pi,\mathcal{A}}(n)$

1 : **foreach** $i \in \mathcal{I}$ **do**
2 : **if** $i \in \mathcal{C}$ **then** $(sk_i, pk_i) \leftarrow \mathsf{KG}_{\mathcal{C}}(1^n)$ **fi**
3 : **if** $i \in \mathcal{S}$ **then** $(sk_i, pk_i) \leftarrow \mathsf{KG}_{\mathcal{S}}(1^n)$ **fi**
4 : **endforeach**
5 : $pks \leftarrow \{(i, pk_i) \mid i \in \mathcal{I}\}$
6 : $\mathcal{A}^{\mathsf{NewSession}(\cdot),\mathsf{Send}(\cdot,\cdot),\mathsf{Corrupt}(\cdot),\mathsf{RevealKey}(\cdot)}(1^n, pks)$
7 : $b \leftarrow \mathsf{Pred}$ / evaluate predicate Pred on execution state
8 : **return** $\overline{b}$

Figure 2: Generic security experiment for predicate Pred, capturing in dependence of Pred the notions of Match security and all versions of key confirmation.

associated security notions. As we explain, care needs to be taken to rule out superficially correct, but in fact misleading definitions.

A. Full Key Confirmation

First, we treat the simpler case of *full* key confirmation. These are the guarantees obtained by the party that receives the last message of the protocol: the protocol ensures full key confirmation (for that party) if, when it receives this last message (and therefore accepts the locally derived key), it has the guarantee that there is a (partnered) session of the protocol that has accepted precisely the same key.

Since a protocol cannot achieve the full confirmation property for all sessions simultaneously—in each pair of sessions one party has to finish first—it is convenient to restrict the sessions under considerations to some subset. Since a session is fully described by its label ℓ, including for example the identity of the party running the session, we usually identify the sessions according to their label ℓ which should belong to some set $\mathcal{L}$. Slightly overloading notation (but extending our predicate-based notions in the previous sections in a natural way) we write for example $\mathcal{L}(\ell) = [\ell.\mathsf{id} \notin \mathrm{Corr}]$ to identify an honest party's session ℓ. Analogously, we conveniently reuse the identity sets $\mathcal{C}$ for client and $\mathcal{S}$ for server session labels by defining $\mathcal{C}(\ell) = [\ell.\mathsf{role} = \mathtt{client}]$ resp. $\mathcal{S}(\ell) = [\ell.\mathsf{role} = \mathtt{server}]$.

In the definition below of the full key confirmation predicate we abstractly speak of subsets $\mathcal{L}$ and $\mathcal{L}'$ of labels. The predicate stipulates that for each accepting session with a label ℓ from $\mathcal{L}$, where the partner is neither corrupt nor unauthenticated (in which case the adversary could impersonate the partner), there exists another session with a label ℓ' from $\mathcal{L}'$ such that this session also accepts the same key. Note that we do not demand that the session ℓ' is actually held by the intended partner specified by $\ell.\mathsf{pid}$ (which is captured as a distinct modular property within Match security), but only that it is partnered according to the session identifiers. This conveniently allows combining (full) key confirmation with other sid-based security notions, e.g., to achieve authentication and partnering properties when coupled with (implicit) authentication resp. Match security or to achieve explicit key authentication when linked with key secrecy.[6]

As a final technical remark, key confirmation can only be expected for sessions that communicate with a distinct, uncorrupted party, as we cannot reason about an adversarially controlled session deriving certain values or holding the same key. This is reflected in both the definitions of full and almost-full key confirmation by demanding that $\ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}$.

Definition 3.1 (Full key confirmation predicate): The predicate $\mathsf{FullConf}(\mathcal{L}, \mathcal{L}')$ that defines full key confirmation is the following:

$$\begin{aligned}\forall \ell \in \mathcal{L} :: \ & [\ell.\mathsf{status} = \mathtt{accept} \wedge \ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}] \\ \Longrightarrow \ & [\exists \ell' \in \mathcal{L}' :: \ (\ell'.\mathsf{status} = \mathtt{accept} \\ & \qquad \wedge \mathsf{Partners}(\ell, \ell') \wedge \mathsf{samekey}(\ell, \ell'))].\end{aligned}$$

Note that $\mathsf{Partners}(\ell, \ell')$ ensures that $l \neq l'$.

A protocol offers full key confirmation if no efficient adversary can make the predicate false, except with negligible probability.

Definition 3.2 (Full key confirmation): A key exchange protocol Π provides full $(\mathcal{L}, \mathcal{L}')$-key confirmation if for any PPT adversary $\mathcal{A}$ there exists a negligible function $\mathrm{negl}(n)$ such that

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}}_{\Pi,\mathcal{A}}(n) = 1\right] \leq \mathrm{negl}(n)$$

for the generic security experiment in Figure 2 with the predicate set to $\mathsf{Pred} = \mathsf{FullConf}(\mathcal{L}, \mathcal{L}')$.

Note that the notion above implicitly captures the time-critical aspect that some other session already holds the same key at the point in time when a party accepts. Whereas the predicate FullConf is evaluated on the final execution state, thus not allowing to distinguish between sessions ℓ for which the paired session ℓ' existed *before* or only *after* session ℓ accepted, the quantification over all adversaries $\mathcal{A}$ rules out the case that there has not been such a session ℓ' before. That is, assume that (any of the possibly multiple paired sessions)

[6]Consulting once more the "Handbook of Applied Cryptography" [27], the authors there also treat the properties separately, and define explicit key authentication as the combination of key confirmation with (implicit) key authentication (where the latter comprises authenticity and key secrecy).

ℓ' only accepted after ℓ in an execution of some adversary $\mathcal{A}$. Then one can imagine a pruned version of the adversary which stops immediately after ℓ has accepted, say, simply by picking a random stop point in the execution. If $\mathcal{A}$ triggers the event that any paired session ℓ' existed only afterwards with non-negligible probability, then the pruned version of $\mathcal{A}$ would break full key confirmation as above.

B. Almost-Full Key Confirmation

We now turn to the guarantees that key confirmation can offer to the party that sends the last message in a protocol, and which therefore has no guarantee that its intended partner accepts (since the adversary may simply drop the message).

A FALSE START. To understand the subtleties involved in designing a definition for this case, we first explore a possible notion which, although intuitively appealing, has important shortcomings.

Intuitively, the best guarantee for the party that sends the last message (and accepts) is that there is some other session which, if it eventually accepts, will have accepted the same key. This intuition is captured by the following formula:

$$\forall \ell \in \mathcal{L} :: \ [\ell.\mathsf{status} = \mathtt{accept} \wedge \ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}] \implies \\ [\exists \ell' \in \mathcal{L}' :: \ (\ell'.\mathsf{status} = \mathtt{accept} \implies \mathsf{samekey}(\ell, \ell'))] \, .$$

It turns out that this notion is too weak. The problem is that the predicate is satisfied whenever there is some session ℓ' that has not accepted. To understand why this is the case, consider the negation of the above predicate (which an adversary that attempts to break the property must ensure it evaluates to true).

$$\exists \ell \in \mathcal{L} :: \ \ell.\mathsf{status} = \mathtt{accept} \wedge \ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\} \\ \wedge \left[\forall \ell' \in \mathcal{L}' :: \ \left(\ell'.\mathsf{status} = \mathtt{accept} \wedge \overline{\mathsf{samekey}(\ell, \ell')}\right)\right] .$$

Note that to make the predicate true, the adversary has to ensure that all sessions accept and as soon as a single session ℓ' rejects, the formula cannot be satisfied anymore and the adversary loses. This is clearly too restrictive since at least for sessions unrelated to ℓ, the adversary should not be required to make them accept. To fix the definition, we have to take additional information into account to characterize sessions that will compute the same key as ℓ.

THE RIGHT DEFINITION. We define the notion of *almost-full key confirmation* based on sessions which are waiting to receive the final message. Note that these are sessions which still lack some information to express the full session identifiers and thus revert to *key-confirmation identifiers* for a weaker type of partnering. Almost-full key confirmation then ensures that the identified session holding the same key-confirmation identifier indeed accepts with the same key (if it eventually accepts at all). This essentially captures the previous, fallen short intuition of having another session that, if it eventually accepts, derives the same key, but restricts this requirement to sessions agreeing on the same key-confirmation identifier.

Definition 3.3 (Almost-full key confirmation predicate): The predicate $\mathsf{AlmostConf}(\mathcal{L}, \mathcal{L}')$ that defines almost-full key confirmation is the following:

$$\forall \ell \in \mathcal{L} :: \ [\ell.\mathsf{status} = \mathtt{accept} \wedge \ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}] \implies \\ [\exists \ell' \in \mathcal{L}' :: \ (\ell.\mathsf{kcid} = \ell'.\mathsf{kcid} \\ \wedge (\ell'.\mathsf{status} = \mathtt{accept} \implies \mathsf{samekey}(\ell, \ell')))].$$

KEY-CONFIRMATION IDENTIFIER BINDING. So far, key-confirmation identifiers, on which the definition of the almost-full key confirmation predicate are based upon, are not bound to the actual session identifiers or to keys. In order to give them practical meaning, we need to establish links to the notion of partnering as well as the derived keys.

First, it is natural to require that whenever two sessions are partnered, they in particular agree on the key-confirmation identifier.[7] More importantly, key-confirmation identifiers are supposed to capture the idea that, whenever two sessions accept and hold the same key-confirmation identifier, they also derive the same key. We formalize these concepts by defining the predicate KCIDbind which returns true if and only if all of the following conditions holds.

1) For all sessions ℓ, ℓ' with $\mathsf{Partners}(\ell, \ell') = \mathsf{true}$, it holds that $\ell.\mathsf{kcid} = \ell'.\mathsf{kcid}$, i.e., partnered sessions agree on the same key-confirmation identifier.
2) For all sessions ℓ, ℓ' with $\ell.\mathsf{kcid} = \ell'.\mathsf{kcid}$ and $\ell.\mathsf{status} = \ell'.\mathsf{status} = \mathtt{accept}$, it holds that $\mathsf{samekey}(\ell, \ell') = \mathsf{true}$, i.e., sessions with the same key-confirmation identifier, upon acceptance, will derive the same key.

Definition 3.4 (Key-confirmation identifier binding): A key exchange protocol Π provides key-confirmation identifier binding if for any PPT adversary $\mathcal{A}$ there exists a negligible function $\mathrm{negl}(n)$ such that

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}}_{\Pi,\mathcal{A}}(n) = 1\right] \leq \mathrm{negl}(n)$$

for the generic security experiment in Figure 2 with the predicate set to $\mathsf{Pred} = \mathsf{KCIDbind}$.

DEFINING ALMOST-FULL KEY CONFIRMATION. We are now ready to define almost-full key confirmation.

Definition 3.5 (Almost-full key confirmation): A key exchange protocol Π provides almost-full $(\mathcal{L}, \mathcal{L}')$-key confirmation if it satisfies key-confirmation identifier binding and for any PPT adversary $\mathcal{A}$ there exists a negligible function $\mathrm{negl}(n)$ such that

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}}_{\Pi,\mathcal{A}}(n) = 1\right] \leq \mathrm{negl}(n)$$

for the generic security experiment in Figure 2 with the predicate set to $\mathsf{Pred} = \mathsf{AlmostConf}(\mathcal{L}, \mathcal{L}')$.

To elaborate why this formalization of almost-full key confirmation captures the right property, let us first again

[7] We note that, beyond this connection, we do not require any particular properties (e.g., concerning authentication) from session identifiers in the context of key confirmation. These aspects are modularly captured within Match security and the definition of (implicit) authentication.

consider the negation of the predicate AlmostConf (i.e., the formula the adversary needs to make evaluate to true):

$$\exists \ell \in \mathcal{L} :: \ \ell.\mathsf{status} = \mathtt{accept} \wedge \ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}$$
$$\wedge\, [\forall \ell' \in \mathcal{L}' :: \ \ell.\mathsf{kcid} \neq \ell'.\mathsf{kcid}$$
$$\vee\, (\ell'.\mathsf{status} = \mathtt{accept} \wedge \overline{\mathsf{samekey}(\ell, \ell')})].$$

First of all note that the part $\ell.\mathsf{kcid} \neq \ell'.\mathsf{kcid}$ formalizes that a protocol cannot choose to have unique key-confirmation identifiers per session, e.g., by setting the identifier to some local random value. This is so as this would trivially mean that for any session $\ell \in \mathcal{L}$ that an adversary initiates, all other sessions $\ell' \in \mathcal{L}'$ have non-matching key-confirmation identifiers, so the adversary immediately wins.

When a protocol instead lets every session $\ell \in \mathcal{L}$ accept with a kcid that matches the one of some session $\ell' \in \mathcal{L}'$, this allows the adversary to focus on such matching sessions. In contrast to the initial false-start formalization, the adversary can in particular let sessions reject that do not hold the same key-confirmation identifier as ℓ.

Finally, a trivial way to achieve almost-full key confirmation is for a protocol to set kcid to the same (e.g., empty) value for every session. Note that key-confirmation identifier binding then in turn requires that every session accepts with the same key. Although this rightly appears to be unreasonable (as it contradicts key secrecy), it is consistent from the perspective of key confirmation: If every session derives the same key, every session is trivially assured that, if there is another accepting session, it will hold the same key.

CHOOSING A KEY-CONFIRMATION IDENTIFIER. A natural question arising from the definition of almost-full key confirmation is how to set the key-confirmation identifiers for a specific protocol. As for the regular session identifiers and their use within the freshness condition for defining key secrecy, there is an interplay between the security notion (key secrecy resp. almost-full key confirmation) and the soundness requirements for the identifiers (Match security resp. key-confirmation identifier binding).

On the one hand, to achieve almost-full key confirmation, a protocol has to couple up any accepting session (in $\mathcal{L}$) with a session (in $\mathcal{L}'$) holding the same key-confirmation identifier kcid. As already discussed, this in particular prevents using a unique kcid value per session. On the other hand, choosing the same key-confirmation identifier (e.g., an empty kcid) for every session, by key-confirmation identifier binding, implies that every session must derive the same key. As this in particular contradicts key secrecy, it is also not a viable option for any reasonable key exchange protocol.

Therefore a protocol needs to balance out the choice for setting key-confirmation identifiers between these two extremes. From a practical point of view, the key-confirmation identifier would intuitively comprise as much of the session identifier such that, together with the last protocol message, it fully determines the derived key. In some cases, even the actual key might already be computable (and hence serve as a "trivial" key-confirmation identifier) before the last message is received. The generic transformation based on an additional exchange of MACs given in Section V is such an example. In many practical protocols (that intuitively achieve almost-full key confirmation), however, the last message(s) will substantially contribute to the key and, hence, only partial information is available when setting (and hence captured in) the key-confirmation identifier. This is, for example, the case in TLS 1.3 (cf. Section IV for our detailed analysis), where the key is derived from a hash over all messages, including the client's last messages CCRT and CCV. Therefore, when the client in TLS 1.3 accepts, the server does not know these messages yet and cannot have set kcid based on the key. Instead, we need to leverage the already exchanged part of the session identifier as key-confirmation identifier, which then fixes a unique key together with the client's last messages. This motivates why we chose to capture "agreement on the same key up to receipt of the last message" using a generic identifier string rather than relying on a particular protocol value or a partial communication transcript.

C. Relationship

We now take a look at the relationship between full and almost-full key confirmation and see why the former implies the latter (given key-confirmation identifier binding).

Theorem 3.6: Let Π be key exchange protocol that provides full $(\mathcal{L}, \mathcal{L}')$-key confirmation as well as key-confirmation identifier binding. Then Π also provides almost-full $(\mathcal{L}, \mathcal{L}')$-key confirmation.

Proof: We need to show that, for any session $\ell \in \mathcal{L}$ that accepts ($\ell.\mathsf{status} = \mathtt{accept}$) with a distinct, uncorrupted partner ($\ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}$), there exists a session $\ell' \in \mathcal{L}'$ which

(a) shares the same key-confirmation identifier ($\ell.\mathsf{kcid} = \ell'.\mathsf{kcid}$) and
(b) on acceptance derives the same key ($\ell'.\mathsf{status} = \mathtt{accept} \implies \mathsf{samekey}(\ell, \ell')$).

First, observe that by full $(\mathcal{L}, \mathcal{L}')$-key confirmation, for any such session $\ell \in \mathcal{L}$ there exists a session $\ell' \in \mathcal{L}'$ that accepted ($\ell'.\mathsf{status} = \mathtt{accept}$), is partnered with ℓ ($\mathsf{Partners}(\ell, \ell')$), and holds the same key ($\mathsf{samekey}(\ell, \ell')$). Due to key-confirmation identifier binding, ℓ' then also shares the same key-confirmation identifier (i.e., $\ell.\mathsf{kcid} = \ell'.\mathsf{kcid}$), satisfying (a). Furthermore (again by key-confirmation identifier binding), when ℓ' accepts, we have additionally that $\ell.\mathsf{status} = \ell'.\mathsf{status} = \mathtt{accept}$, and so $\mathsf{samekey}(\ell, \ell')$ holds, satisfying (b). ∎

MATCH SECURITY VS. KEY CONFIRMATION. On a more distant relation, let us note that Match security and key confirmation (both full and almost-full) are independent notions (i.e., a protocol can provide either one without providing the other one). On the one hand, setting sid = kcid to be a unique string per session trivially lets the protocol satisfy Match security, but renders full and almost-full key confirmation unachievable. On the other hand, having all sessions use the same (arbitrary) identifiers sid and kcid and derive the same key key trivially

satisfies full and almost-full key confirmation, but violates Match security (due to more than two sessions being partnered with each other).

Match security thus rather constitutes a counterpart to the freshness conditions used for key secrecy than being related to key confirmation.

D. Confirmation guarantees for unauthenticated peers

Informal definitions of key confirmation usually demand that another, even *possibly unidentified* party holds the same key (e.g., the "Handbook of Applied Cryptography" [27, Definition 12.7]). Note that our notions of full and almost-full key confirmation originally guarantee that for sessions which communicate with an *identified* (and uncorrupted) partner ($\ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}$) there is another session which (eventually) holds the same key.

The extension of our notions to unauthenticated peers turn out to hold trivially by correctness(-like) properties. For this, first note that key confirmation guarantees for sessions with unauthenticated peers can only be provided if the actual communication partner is indeed honest. This is so since no confirmation model can ensure that an adversarially-controlled session derives a session key at some point.

In the case of full key confirmation, any extension to unauthenticated peers (i.e., dropping the prerequisite $\ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}$ for the session ℓ in question) would hence need to condition on the existence of a partnered (honest) session to the session ℓ, e.g., adding the requirement $\exists \ell'' \in \mathcal{L}' :: \mathsf{Partners}(\ell, \ell'')$ to the full key confirmation predicate. But then any (correct) key exchange protocol provides this kind of "unauthenticated" full key-confirmation anyway: Correctness demands that partnered sessions (i.e., with the adversary only passively relaying messages) derive the same key. This means that such a session ℓ'' already serves as a partnered session holding the same key according to our original definition.

In the case of almost-full key confirmation, guarantees for unauthenticated peers would again need to be conditioned on an existing session sharing the same key confirmation identifier (e.g., swapping in the requirement $\exists \ell'' \in \mathcal{L}' :: \ell.\mathsf{kcid} = \ell''.\mathsf{kcid}$ for $\ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}$). But then again, the natural requirement of key-confirmation identifier binding (being part of the almost-full key confirmation property) already satisfies almost-full key-confirmation for such cases trivially: It ensures that two sessions sharing the same key-confirmation identifier, upon acceptance, will derive the same key. Thus, session ℓ'' again serves as the desired partnered session according to out current notion.

In conclusion, obtaining no advantage from definitions encompassing unauthenticated peers, we focus on the key confirmation guarantees attainable when communicating with an authenticated partner.

IV. Key Confirmation in TLS 1.3 draft-10

In this section, we investigate the key confirmation properties provided by the currently developed next version of the Transport Layer Security protocol (TLS 1.3). More precisely, we consider the recent TLS 1.3 draft `draft-ietf-tls-tls13-10` [29] (short: `draft-10`) and its full (EC)DHE handshake. We remark that we do not analyze key secrecy or Match security for `draft-10`, which is beyond the scope of this work, but focus on key confirmation. Key secrecy for TLS 1.3 `draft-10` and earlier draft versions has been recently studied by Dowling et al. [18], [19]. We furthermore do not treat the other handshake modes specified in `draft-10` for resumption (PSK/PSK-(EC)DHE) and zero round-trip time (0-RTT) [29, Section 6.2] or extensions added in later drafts like post-handshake client authentication (`draft-11` [30]).

Figure 3 depicts the TLS 1.3 `draft-10` full (EC)DHE handshake including the essential steps of the Diffie–Hellman-based key derivation.[8] We sometimes use abbreviated message names (`ClientHello` $=$ `CH`, `ClientKeyShare` $=$ `CKS`, and so on).

As in previous versions, TLS 1.3 in `draft-10` employs `Finished` messages (sent both by the client and the server), which are essentially MAC values computed over the (hashed) transcript (excluding the `Finished` messages), the so-called "session hash", in order to "provide[] key confirmation" [29, p. 31]. Importantly, in contrast to previous TLS versions, the MAC key employed is *not* the derived session key, but a separate finished_secret derived through a key derivation function from one of the secret Diffie–Hellman values established within the key exchange. In that sense, `draft-10` essentially follows the popular paradigm we discuss in Section V to exchange MACs over the transcript (or the session identifiers) in order to achieve key confirmation.

A. Key Confirmation without Finished Messages

Interestingly, we can however actually show that already a shortened variant of the `draft-10` handshake *without* the `ClientFinished` and `ServerFinished` messages (which we denote as `draft-10-nf`) provides the same strongest form of key confirmation one can expect. That is, in the mutually authenticating handshake the server is assured that the client already accepted with the same key at the time the server accepts while the client is assured that, if the server later accepts, it will do so with the same key. In contrast, in the unilaterally authenticating handshake—as expected—only the client is guaranteed (full) key confirmation.[9] We will first elaborate in detail how to prove key confirmation for the shortened `draft-10-nf` handshake and then demonstrate in Section IV-B how this result can easily be adapted to the actual `draft-10` handshake.

More formally, we show that the mutually authenticating `draft-10-nf` handshake (short: `draft-10-nf-m`) achieves full

[8] In TLS 1.3 `draft-10`, handshake messages from `EncryptedExtensions` on are sent encrypted under a separately derived handshake traffic key (independent of the final session key). We disregard this handshake encryption in our analysis as it does not affect the key confirmation guarantees for the final session key we are interested in here.

[9] Observe that, for unilateral authentication, in contrast to `draft-10` the server sends the last protocol message in the shortened `draft-10-nf` variant due to the omitted finished messages.

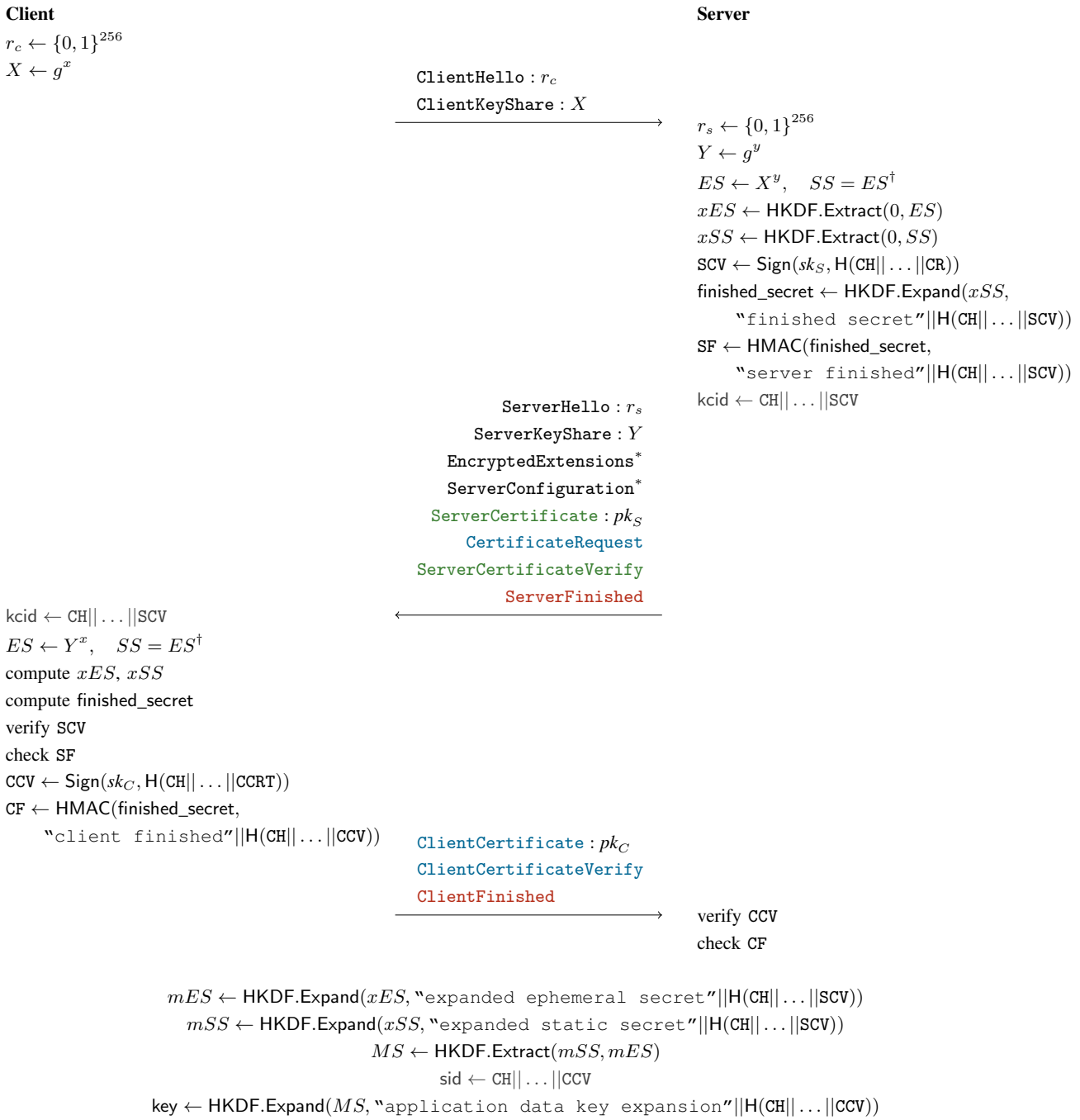

Figure 3: The TLS 1.3 `draft-10` full (EC)DHE handshake protocol with Diffie–Hellman-based key derivation. Messages `ServerCertificate` and `ServerCertificateVerify` are sent both for unilateral and mutual authentication, messages `CertificateRequest`, `ClientCertificate`, and `ClientCertificateVerify` are only sent for mutual authentication. Starred messages are situation-dependent and not always sent. In the computations, we abbreviate message names as `ClientHello` = `CH`, `ClientKeyShare` = `CKS`, etc. Enumerations for session-hash values of messages exchanged (like H(`CH`$||\ldots||$`CCV`)) always exclude the `Finished` messages. Without client authentication, the server may derive the session key already after sending `ServerFinished`.
In the shortened `draft-10-nf` we define and analyze, the `ServerFinished` and `ClientFinished` messages are omitted.

† TLS 1.3 `draft-10` allows handshake variants where the client announces knowledge of a cached server configuration in which case xSS is derived differently. We do not capture this variant in our analysis and hence omit it here for the ease of presentation.

$(\mathcal{S},\mathcal{C})$-key confirmation and almost-full $(\mathcal{C},\mathcal{S})$-key confirmation and that the unilaterally authenticating `draft-10-nf` handshake (`draft-10-nf-u`) provides full $(\mathcal{C},\mathcal{S})$-key confirmation. While we analyze both authentication variants of the handshake separately, we remark that both results also hold when handshakes are allowed to run with mutual and with unilateral concurrently in our model, as the message flow is unique for each authentication variant which allows to tell the according sessions apart.

For both handshake variants our proofs rely on the `ClientCertificateVerify` resp. `ServerCertificateVerify` message exchanged, which essentially is a signature over the complete resp. almost-complete transcript and can, hence, intuitively be seen as a signature-based analogue of the MAC-based transform discussed in Section V. We define the key-confirmation identifiers in `draft-10-nf` to be set to $\mathsf{kcid} = \texttt{ClientHello}||\ldots||\texttt{ServerCertificateVerify}$ by the server on sending its `ServerCertificateVerify` message and by the client on receiving that message.

Theorem 4.1: The TLS 1.3 `draft-10-nf-m` handshake for mutual authentication without finished messages satisfies full $(\mathcal{S},\mathcal{C})$-key confirmation and almost-full $(\mathcal{C},\mathcal{S})$-key confirmation. Formally, for any efficient adversary $\mathcal{A}$ against full $(\mathcal{S},\mathcal{C})$-key confirmation resp. almost-full $(\mathcal{C},\mathcal{S})$-key confirmation there exist efficient algorithms $\mathcal{B}_1$, $\mathcal{B}_2$ such that

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)=1\right] \leq n_s^2\cdot 2^{-|\text{nonce}|}+\mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1}+n_u\cdot\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{B}_2},$$

for $\mathsf{Pred}=\mathsf{FullConf}(\mathcal{S},\mathcal{C})$ resp. $\mathsf{Pred}=\mathsf{AlmostConf}(\mathcal{C},\mathcal{S})$, where n_u is the maximum number of participating parties, n_s is the maximum number of sessions, and $|\text{nonce}|=256$ is the bit-length of the nonces.

Proof: We show that, for a server session, the `ClientCertificateVerify` message of an honest client suffices as assurance that this client has accepted with the same key when the server session accepts. In turn, for a client session, the `ServerCertificateVerify` message of an honest server ensures that this server agrees on the key-confirmation identifier kcid and will, if it accepts, derive the same key. To this extend we modify the original $\mathbf{Exp}^{\mathsf{Pred}}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)$ experiment in three steps, showing that the advantage difference of adversary $\mathcal{A}$ can each time be bounded by the advantage of breaking the security of TLS 1.3 components, finally reaching an experiment where the advantage of $\mathcal{A}$ is 0.

First, we consider the modified experiment $\mathbf{Exp}^{\mathsf{Pred}'}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)$ which is as before, except it aborts (and outputs 0) if, during the execution, any two honest sessions choose the same nonce (r_c or r_s). The probability that the experiment aborts can be bounded from above by $n_s^2\cdot 2^{-|\text{nonce}|}$ where n_s is the maximum number of sessions and $|\text{nonce}|$ is the nonces' bit-length. Therefore,

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)=1\right] \leq \Pr\left[\mathbf{Exp}^{\mathsf{Pred}'}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)=1\right]+n_s^2\cdot 2^{-|\text{nonce}|}.$$

Second, we switch to $\mathbf{Exp}^{\mathsf{Pred}''}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)$ in which we also abort if, during the execution, any two honest sessions compute the same hash value for two different inputs to the hash function H. We can bound the probability that this experiment aborts for this reason by the advantage $\mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1}$ of an adversary $\mathcal{B}_1$ against the collision resistance of H. For this purpose, $\mathcal{B}_1$ simply simulates $\mathbf{Exp}^{\mathsf{Pred}''}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)$ for $\mathcal{A}$ on its own and outputs the two colliding inputs when they occur during the simulation, perfectly simulating the experiment up to this point and always winning when the modified experiment aborts. Hence we have that

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}'}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)=1\right] \leq \Pr\left[\mathbf{Exp}^{\mathsf{Pred}''}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)=1\right]+\mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1}.$$

Third, we consider $\mathbf{Exp}^{\mathsf{Pred}'''}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)$, which, in addition, aborts whenever a simulated server resp. client session obtains, within the `ClientCertificateVerify` resp. `ServerCertificateVerify` message, a valid signature (under the public key of some non-corrupted client resp. server $i\in\mathcal{I}$) which was not output by any honest client resp. server session. Again, we can bound the probability of this abort by the advantage $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{B}_2}$ of an adversary $\mathcal{B}_2$ against the existential unforgeability of the deployed signature scheme Sig, simulating the experiment for $\mathcal{A}$ as follows. Initially, $\mathcal{B}_2$ randomly chooses a party i among the at most n_u parties, associating the challenge public key pk^* with it, and generates the long-term key for all other parties $i\in\mathcal{I}\setminus\{i\}$. During the simulation, $\mathcal{B}_2$ then uses it signing oracle whenever a signature needs to be computed under the secret key of i. When a simulated session obtains a valid signature that no other honest session has output, $\mathcal{B}_2$ aborts the experiment and outputs that signature as its forgery. If $\mathcal{B}_2$ correctly guessed the (non-corrupted) identity i under whose public key pk^* the obtained signature verifies, this is a valid forgery and $\mathcal{B}_2$ wins, thus

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}''}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)=1\right] \leq \Pr\left[\mathbf{Exp}^{\mathsf{Pred}'''}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)=1\right]+\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{B}_2}.$$

Finally, we can now separately argue along the lines of the predicates $\mathsf{Pred}=\mathsf{FullConf}(\mathcal{S},\mathcal{C})$ resp. $\mathsf{Pred}=\mathsf{AlmostConf}(\mathcal{C},\mathcal{S})$ as well as $\mathsf{Pred}=\mathsf{KCIDbind}$ (which is required as part of almost-full key confirmation):

- For $\mathsf{Pred}=\mathsf{FullConf}(\mathcal{S},\mathcal{C})$, observe that each accepting server session with a non-corrupted partner obtains, within the `ClientCertificateVerify` message, a valid signature generated by an honest client over the hash of the transcript $\texttt{ClientHello}||\ldots||\texttt{ClientCertificate}$,

which exactly relates to that transcript as no hash collisions occurred. Hence, in particular, for each such server session that accepts there exists a client session that already accepted, shares the same transcript, and, therefore, in particular also holds the same session identifier and derives the same session key.

- For $\mathsf{Pred} = \mathsf{AlmostConf}(\mathcal{C},\mathcal{S})$, note that the signature sent within `ServerCertificateVerify` is computed over (the non-colliding hash of) the transcript up to the `ServerCertificate` message which, along with the `ServerCertificateVerify` message itself forms the key-confirmation identifier kcid. Therefore, for any accepting client session ℓ, there exists an honest server session ℓ' sharing the same kcid.
  Furthermore, no second honest client will send a `ClientCertificateVerify` message with a signature over a (hash of a) transcript containing the same kcid contents due to unique nonces. Also, no server obtains a forged signature and no hash collision occurs. Thus the server session ℓ' will only accept when receiving this client session's `ClientCertificateVerify` message. As the values signed within `ClientCertificateVerify` together with `ClientCertificateVerify` uniquely determine the key derivation, session ℓ' will, if at all, accept with the same session key.
- For $\mathsf{Pred} = \mathsf{KCIDbind}$, the first condition that equal session identifiers imply equal key-confirmation identifiers follows immediately from defining kcid to be a prefix of sid.
  The second condition is satisfied for the same reasons that make sessions holding the same kcid in the AlmostConf predicate derive the same key. Again, kcid contains the transcript up to the `ServerCertificateVerify` message, which is signed by the client within its `ClientCertificateVerify` message that no second honest client will generate (and which can, at this point, neither be forged nor be derived due to colliding nonces or hashes). Therefore, if a client accepts (as the first of two sessions sharing the same kcid), it outputs the only `ClientCertificateVerify` message server session will accept. As `ClientCertificateVerify` fixes the complete transcript which fully determines the derived keys, both sessions will necessarily derive the same key as required.

In other words, Pred (being $\mathsf{FullConf}(\mathcal{S},\mathcal{C})$, $\mathsf{AlmostConf}(\mathcal{C},\mathcal{S})$, resp. KCIDbind) is always satisfied in $\mathbf{Exp}^{\mathsf{Pred}'''}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)$ and thus

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}'''}_{\texttt{draft-10-nf-m},\mathcal{A}}(n)=1\right]=0.$$

■

Theorem 4.2: The TLS 1.3 `draft-10-nf-u` handshake for unilateral authentication without finished messages satisfies full $(\mathcal{C},\mathcal{S})$-key confirmation. Formally, for any efficient adversary $\mathcal{A}$ against full $(\mathcal{C},\mathcal{S})$-key confirmation there exist efficient algorithms $\mathcal{B}_1$, $\mathcal{B}_2$ such that

$$\Pr\left[\mathbf{Exp}^{\mathsf{FullConf}(\mathcal{C},\mathcal{S})}_{\texttt{draft-10-nf-u},\mathcal{A}}(n)=1\right] \leq \mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1}+n_u\cdot\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{B}_2},$$

where n_u is the maximum number of participating parties.

Proof: We show that, for unilateral authentication, the `ServerCertificateVerify` message obtained by a client session ensures that the sending server session has already accepted with the same session key.

Again, we first modify the original $\mathbf{Exp}^{\mathsf{FullConf}(\mathcal{C},\mathcal{S})}_{\texttt{draft-10-nf-u},\mathcal{A}}(n)$ experiment in two steps similar to those in the proof of Theorem 4.1. First, we identically bound the probability that two honest sessions compute a colliding hash value for two different inputs by $\mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1}$ for an efficient reduction $\mathcal{B}_1$. We then ensure that no client session obtains a `ServerCertificateVerify` message containing a valid signature under an honest server's public key which was not generated by any honest session (recall that there is no `ClientCertificateVerify` message sent as the client does not authenticate). Similarly to the proof of Theorem 4.1 we can bound the probability that this happens by the advantage $\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{B}_2}$ of an adversary $\mathcal{B}_2$, again using its signing oracle for the challenged client identity.

Due to these modifications, we are now assured of unconditional full $(\mathcal{C},\mathcal{S})$-key confirmation: the signature sent within `ServerCertificateVerify` together with that message itself fully determines the key derivation and, hence, whenever a client accepts, the (honest) server session sending sending the `ServerCertificateVerify` message already accepted with the same session identifier and session key. ■

B. Key Confirmation with Finished Messages

Coming back to the original TLS 1.3 `draft-10` handshake, it is easy to see that the proof in Theorem 4.1 for key confirmation under mutual authentication in `draft-10-nf` immediately carries over to the `draft-10` handshake with mutual authentication (short: `draft-10-m`). For this, recall that the `ServerFinished` and `ClientFinished` message do not enter the key derivation, which means they can essentially be treated as an "arbitrary bitstring" attached to the `ServerCertificateVerify` resp. `ClientCertificateVerify` messages. Hence, we can apply the identical proof to the `draft-10` mutually authenticating handshake to establish the same key confirmation properties.

Theorem 4.3: The TLS 1.3 `draft-10-m` handshake for mutual authentication with finished messages satisfies full $(\mathcal{S},\mathcal{C})$-key confirmation and almost-full $(\mathcal{C},\mathcal{S})$-key confirmation. Formally, for any efficient adversary $\mathcal{A}$ against full $(\mathcal{S},\mathcal{C})$-key confirmation resp. almost-full $(\mathcal{C},\mathcal{S})$-key confirmation there exist efficient algorithms $\mathcal{B}_1$, $\mathcal{B}_2$ such that

$$\Pr\left[\mathbf{Exp}^{\mathsf{Pred}}_{\texttt{draft-10-m},\mathcal{A}}(n)=1\right] \leq n_s^2\cdot 2^{-|\text{nonce}|}+\mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1}+n_u\cdot\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{B}_2},$$

for $\mathsf{Pred} = \mathsf{FullConf}(\mathcal{S},\mathcal{C})$ resp. $\mathsf{Pred} = \mathsf{AlmostConf}(\mathcal{C},\mathcal{S})$, where n_u is the maximum number of participating parties, n_s is the maximum number of sessions, and $|\text{nonce}| = 256$ is the bit-length of the nonces.

Interestingly, when it comes to unilateral authentication, the `ClientFinished` messages sent as the single additional message from the client to the server changes the order in which the session key is accepted (the client accepting first here) and, hence, necessarily renders full $(\mathcal{C},\mathcal{S})$-key confirmation (as for `draft-10-nf-u`) unachievable. We can however show that clients indeed still enjoy almost-full $(\mathcal{C},\mathcal{S})$-key confirmation for the unilaterally authenticating `draft-10` handshake (`draft-10-u`).

Theorem 4.4: The TLS 1.3 `draft-10-u` handshake for unilateral authentication with finished messages satisfies almost-full $(\mathcal{C},\mathcal{S})$-key confirmation. Formally, for any efficient adversary $\mathcal{A}$ against almost-full $(\mathcal{C},\mathcal{S})$-key confirmation there exist efficient algorithms $\mathcal{B}_1$, $\mathcal{B}_2$ such that

$$\Pr\left[\mathbf{Exp}^{\mathsf{AlmostConf}(\mathcal{C},\mathcal{S})}_{\texttt{draft-10-u},\mathcal{A}}(n) = 1\right] \leq \mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1} + n_u \cdot \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig},\mathcal{B}_2},$$

where n_u is the maximum number of participating parties.

Proof: We show that the `ServerCertificateVerify` message obtained from an honest server ensures this server agrees on kcid and will, on acceptance, derive the same key.

After excluding hash collisions and signature forgeries within `ServerCertificateVerify` again as in the proof of Theorem 4.2, the (unforged) `ServerCertificateVerify` message, computed over (the non-colliding hash of) the transcript, ensures the sending server agrees on the same key-confirmation identifier kcid.

Furthermore, the kcid value uniquely determines the derived key[10]: observe that kcid = `ClientHello`$||\ldots||$`ServerCertificateVerify` contains all messages affecting the key derivation as messages `ClientCertificate` and `ClientCertificateVerify` are not sent and `ServerFinished` as well as `ClientFinished` are not included in the session hashes. Therefore, any two (accepting) sessions agreeing on the same kcid will derive the same session key and hence $\mathsf{AlmostConf}(\mathcal{C},\mathcal{S})$ as well KCIDbind are satisfied (note that still kcid is a prefix of sid). ■

To summarize, our analysis shows that key confirmation in the full handshake of the latest `draft-10` of TLS 1.3 is already established through the exchanged `CertificateVerify` messages and, hence, the `Finished` messages included for that purpose are somewhat redundant. This unveils a potential misconception that MACs over the transcript (in form of the `Finished` messages) are always necessary to achieve key confirmation.

[10]This in particular renders requiring unique nonces unnecessary in this case.

Admittedly, there are however further (non-full) handshake modes provisioned in `draft-10`, particularly for key exchanges based on a cached server configuration or achieving zero round-trip time [29, Sections 6.2.2 and 6.2.3]. These variants do not necessarily exchange `CertificateVerify` messages and, hence, rely on the `Finished` messages to provide both authentication as well as key confirmation. For the sake of a uniform protocol design, it therefore seems advisable to keep `Finished` messages also in the full handshake, even though their main purpose, key confirmation, can as well be achieved through the `CertificateVerify` messages only.

V. Generic Transform

We next show that the popular transformation of exchanging MACs over the session identifiers at the end of a key exchange protocol, as for instance advised in [2], [1], actually provides the strongest form of key confirmation one can expect. Here, a key-refreshing step should be carried out for generating fresh MAC keys (one for the client and one for the server) as well as a separate, new session key via a key derivation function KDF. In the remainder of the section we assume that the client always goes last in a protocol Π and the server therefore sends the first MAC in the conclusive transfers.

Below we assume that KDF acts as a secure pseudorandom generator, i.e., that its output for a random input key key is indistinguishable from random, and that its output can be formatted to be used as a session key and MAC keys.

The MAC scheme $\mathcal{M} = (\mathsf{KGen},\mathsf{MAC},\mathsf{Vf})$ should be key-only unforgeable in the sense that the adversary cannot output a message m^* with a valid MAC, without even being allowed to see a MAC for another message. This in particular implies that for key confirmation, as performed in our transformation through independent keys for client and for server, it suffices to output parts of the refreshed key to confirm that one holds the (refreshed) key part. If both parties use the same key for message authentication then one needs a one-time unforgeable MAC scheme, though. We touch this issue at the end of the section when discussing alternative approaches.

Theorem 5.1: Let KDF be a secure pseudorandom generator and $(\mathsf{KGen},\mathsf{MAC},\mathsf{Vf})$ be a key-only unforgeable MAC scheme. If a key exchange protocol Π has Match security and key secrecy, then the transformation Π_{MAC} in Figure 4 preserves these properties, and moreover satisfies full $(\mathcal{S},\mathcal{C})$-key confirmation and almost-full $(\mathcal{C},\mathcal{S})$-key confirmation. The protocol Π_{MAC} also provides the same authentication property as Π.

Proof: We first note that any security property of Π_{MAC} defined via the generic experiment in Figure 2 (for one of our predicates Pred in question), involving some adversary $\mathcal{A}$, can be perfectly simulated by an adversary $\mathcal{A}'$ playing the generic experiment (for Pred) against Π. Adversary $\mathcal{A}'$ runs a black-box simulation of $\mathcal{A}$. For this, it always makes a RevealKey query for any session that accepts (in Π) and simulates the final steps of Π_{MAC} with the derived key, and otherwise relays all oracle queries of $\mathcal{A}$ in Π_{MAC} to its oracles.

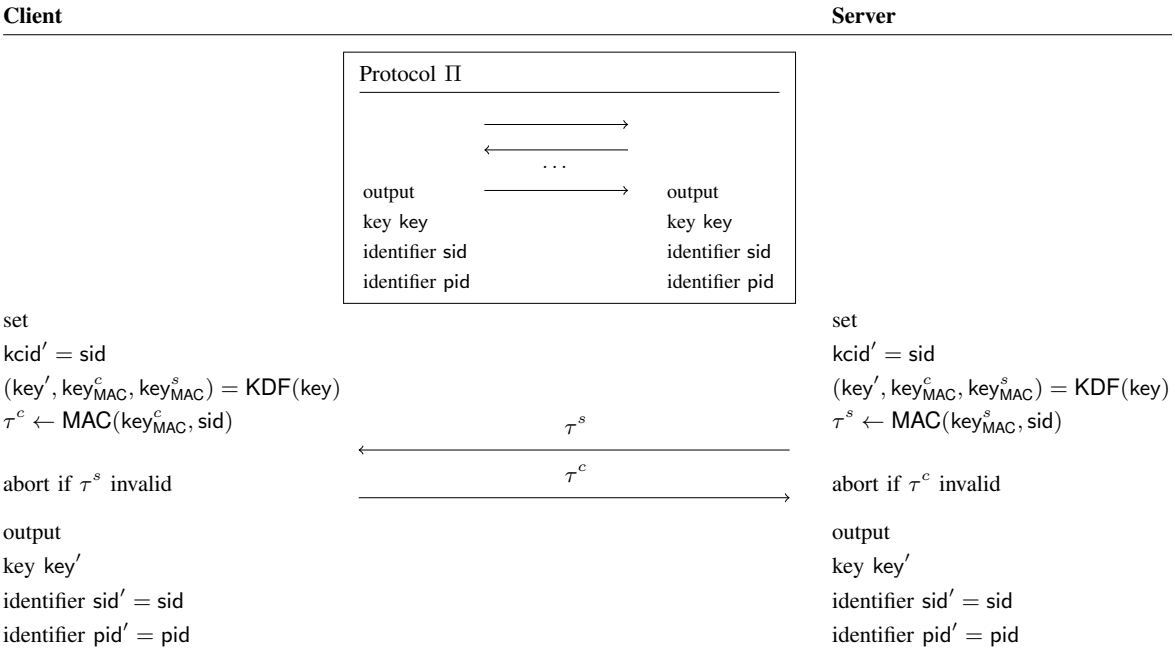

Figure 4: Transformed protocol Π_{MAC}.

Since the simulation is perfect we can view any simulated run of $\mathcal{A}$ as a concrete execution of its experiment against Π_{MAC}. Furthermore, since for the predicates we consider here the RevealKey queries of $\mathcal{A}'$ are irrelevant to the outcome, it follows that any concrete simulated execution of $\mathcal{A}$ satisfying the predicate in the corresponding experiment against Π_{MAC} also yields the result true in the experiment of $\mathcal{A}'$ against Π.

Match SECURITY. To show Match security of Π_{MAC} consider an arbitrary fixed execution of $\mathcal{A}$, simulated through $\mathcal{A}'$. For that particular execution, note that partnering in Π_{MAC} implies partnering in Π, because if two sessions in Π_{MAC} have accepted and hold the same session identifier $\mathsf{sid}' = \mathsf{sid}$, then this is also true for Π. Then violation of any of the four Match-security properties in Π_{MAC} entails a corresponding violation for Π:

1) Partnered sessions cannot hold distinct keys: As explained above, if two sessions are partnered for Π_{MAC} then they are also partnered for Π. If they hold distinct keys in Π_{MAC}, then they also must so in Π, because key derivation is deterministic and they would otherwise also obtain the same key in Π_{MAC}.
2) Three or more sessions share the same session identifier: Since identifiers are identical in both settings, this follows straightforwardly.
3) Two partnered sessions do not adopt the client-server relationship: Again, since this does not change in Π_{MAC} this again follows easily.
4) A partnered session assumes to talk to a different party: Since partnering carries over and ℓ.id and ℓ.pid coincide in Π and Π_{MAC} for all sessions ℓ, this cannot happen.

Hence, Match security of Π_{MAC} follows from the Match security of Π.

AUTHENTICATION. Violation of the authentication property in Π_{MAC} means that an honest session ℓ outputs $\ell.\mathsf{pid}' \neq *$ but such that the partnered session ℓ' has a different identifier $\ell'.\mathsf{id}'$. Such a violation must occur for both unilateral and mutual authentication. Since the value of $\mathsf{pid}' = \mathsf{pid}$ is identical in Π and Π_{MAC} it follows that the mismatch must already appear in Π. Hence, we can construct an adversary $\mathcal{A}'$ for Π using an adversary $\mathcal{A}$ against Π_{MAC} like in the previous case.

KEY SECRECY. Next we show key secrecy of the derived protocol Π_{MAC}. To this end we first "normalize" any attacker $\mathcal{A}_{\mathsf{MAC}}$ against Π_{MAC}. The first modification refers to the fact that we let $\mathcal{A}_{\mathsf{MAC}}$ lose if there are sessions with identical session identifiers but different keys. It follows from the Match-security that we only decrease $\mathcal{A}_{\mathsf{MAC}}$'s success probability negligibly, but from now on we can assume consistent keys in the sense that, if the public session identifiers match, then so do the session keys.

The next step is to assume that $\mathcal{A}_{\mathsf{MAC}}$ "is fresh" in the sense that it never violates the freshness predicate Fresh. This can be accomplished by letting $\mathcal{A}_{\mathsf{MAC}}$ stop and return 0 immediately before it would violate the freshness condition, e.g., revealing a session key of a partner of the tested session. Such cases can be easily checked with the help of the public session identifiers. Since the adversary would lose when refuting freshness this modification cannot decrease the adversary's success probability. The freshness condition ensures that we can conveniently re-write the secrecy condition that $\Pr\left[\mathbf{Exp}^{\mathsf{scrcy}}_{\Pi_{\mathsf{MAC}},\mathcal{A}_{\mathsf{MAC}}}(n) = 1\right]$

is negligibly close to $\frac{1}{2}$ as demanding that

$$\left|\Pr\left[\mathbf{Exp}^{\mathsf{scrcy\text{-}lr},0}_{\Pi_{\mathsf{MAC}},\mathcal{A}_{\mathsf{MAC}}}(n)=1\right]-\Pr\left[\mathbf{Exp}^{\mathsf{scrcy\text{-}lr},1}_{\Pi_{\mathsf{MAC}},\mathcal{A}_{\mathsf{MAC}}}(n)=1\right]\right|$$

is negligible, where $\mathbf{Exp}^{\mathsf{scrcy\text{-}lr},b}_{\Pi_{\mathsf{MAC}},\mathcal{A}_{\mathsf{MAC}}}(n)$ is the experiment $\mathbf{Exp}^{\mathsf{scrcy}}_{\Pi_{\mathsf{MAC}},\mathcal{A}_{\mathsf{MAC}}}(n)$ with the Test oracle's bit b fixed.

The third change is to assume that $\mathcal{A}_{\mathsf{MAC}}$, before the actual experiment begins, outputs an index t such that it will test the t-th completed session with label ℓ_t, and that there will be no partnered session accepting before the tested session. If ℓ_t would not satisfy this we can assume that $\mathcal{A}_{\mathsf{MAC}}$ loses the game. This modification can only decrease the adversary's advantage (over $\frac{1}{2}$) by a factor $\frac{1}{n_s}$ for the maximal number n_s of sessions. To see this define another adversary which initially picks t at random from $\{1,2,\ldots,n_s\}$ and the runs a black-box simulation of $\mathcal{A}_{\mathsf{MAC}}$, checking via the public session identifiers that the choice satisfies the required properties. Note that this new adversary still "is fresh" if $\mathcal{A}_{\mathsf{MAC}}$ is.

Now assume that there exists a normalized PPT adversary $\mathcal{A}_{\mathsf{MAC}}$ for the protocol Π_{MAC} such that the distinguishing probability above is non-negligible. Then we can define the following adversary $\mathcal{A}$ against the original protocol Π. Initially, $\mathcal{A}_{\mathsf{MAC}}$ outputs its test session number t. Then $\mathcal{A}$ runs a black-box simulation of $\mathcal{A}_{\mathsf{MAC}}$, using its external oracles to simulate the Send interaction of $\mathcal{A}_{\mathsf{MAC}}$ with the core part Π of the protocol Π_{MAC}. Once this core part is completed, adversary $\mathcal{A}$ first fetches the session key key of Π. There are three cases for that:

- For sessions accepting before the t-th session, as well as for sessions accepting later, but which are not partnered with session ℓ_t adversary $\mathcal{A}$ makes Reveal request to get the session key.
- For the t-th session adversary $\mathcal{A}$ calls the Test oracle to get the key.
- For any session partnered with ℓ_t, which completes after the test session, adversary $\mathcal{A}$ uses the key it has obtained through the Test query.

Given such a key key, adversary $\mathcal{A}$ computes $(\mathsf{key}',\mathsf{key}^c_{\mathsf{MAC}},\mathsf{key}^s_{\mathsf{MAC}}) \leftarrow \mathsf{KDF}(\mathsf{key})$ and continues the execution internally in the simulation, with the help of these keys. For the other queries, Test and Reveal of $\mathcal{A}_{\mathsf{MAC}}$, adversary $\mathcal{A}$ proceeds correspondingly, fetching the key key with the corresponding query (if $\mathcal{A}$ does not hold it already), computing $(\mathsf{key}',\mathsf{key}^c_{\mathsf{MAC}},\mathsf{key}^s_{\mathsf{MAC}}) \leftarrow \mathsf{KDF}(\mathsf{key})$ and returning key'. Any Corrupt query is simply relayed. Eventually, $\mathcal{A}$ outputs $\mathcal{A}_{\mathsf{MAC}}$'s guess b'.

Consider for the moment the case that the Test oracle's bit in $\mathcal{A}$'s attack is $b=0$, such that the oracle returns the actual session key key to $\mathcal{A}$ in the Test query. In this case the above simulation for the normalized adversary $\mathcal{A}_{\mathsf{MAC}}$ perfectly mimics an attack on Π_{MAC} of this adversary. The reason is that identical session identifiers guarantee identical keys, such that $\mathcal{A}$ consistently uses the right keys when identifying partners of the tested session via session identifiers. Furthermore, $\mathcal{A}$ only makes additional Reveal queries (on top of the ones that $\mathcal{A}_{\mathsf{MAC}}$ makes) to sessions unpartnered to the test session, since by assumption about $\mathcal{A}_{\mathsf{MAC}}$ there are no partnered sessions to the t-th one before that session has completed. It follows that $\mathcal{A}$ never violates freshness with the extra queries, and thus "is fresh", too.

The more challenging case is that $\mathcal{A}$'s Test oracle uses $b=1$ and returns an independent key key. Then $\mathcal{A}$ as above returns a key key' derived via the key derivation function KDF from key, whereas the original experiment would return a random key key' instead. We next show that, given $b=1$, the difference is negligible by the pseudorandomness of KDF. Suppose that, instead of using $\mathsf{KDF}(\mathsf{key})$ for independent and random key key—recall that we fix $b=1$—our adversary $\mathcal{A}$ would use a random tuple $(\mathsf{key}',\mathsf{key}^c_{\mathsf{MAC}},\mathsf{key}^s_{\mathsf{MAC}})$ and run the identical experiment as above, only using this tuple now. Then the output of $\mathcal{A}$ cannot change significantly, or else we straightforwardly obtain a distinguisher against KDF which receives either $\mathsf{KDF}(\mathsf{key})$ for random key, or a random $(\mathsf{key}',\mathsf{key}^c_{\mathsf{MAC}},\mathsf{key}^s_{\mathsf{MAC}})$ as input.

Next, change $\mathcal{A}$'s simulation by letting it pick an independent key'' instead of the random key'. This does not change the output distribution, but it enables us to revert to the pseudorandom values $(\mathsf{key}',\mathsf{key}^c_{\mathsf{MAC}},\mathsf{key}^s_{\mathsf{MAC}}) \leftarrow \mathsf{KDF}(\mathsf{key})$ again, by having $\mathcal{A}$ replace key' by a random value key'' in the first component. If the pseudorandomness of $(\mathsf{key}',\mathsf{key}^c_{\mathsf{MAC}},\mathsf{key}^s_{\mathsf{MAC}}) \leftarrow \mathsf{KDF}(\mathsf{key})$ would change the output distribution significantly, then we would again obtain a successful distinguisher against the pseudorandomness of KDF. The final hop is to use the actual session key key of protocol Π, instead of a random key key to derive $(\mathsf{key}',\mathsf{key}^c_{\mathsf{MAC}},\mathsf{key}^s_{\mathsf{MAC}}) \leftarrow \mathsf{KDF}(\mathsf{key})$. The closeness here follows from the security of the original protocol Π, implying that this cannot alter $\mathcal{A}$'s output behavior significantly, or else we easily obtain a successful adversary against the key secrecy of Π. In this final game, however, the simulation (for fixed bit $b=1$) is perfectly indistinguishable from $\mathcal{A}_{\mathsf{MAC}}$'s attack on Π_{MAC} for $b=1$, since $\mathcal{A}_{\mathsf{MAC}}$ now obtains an independent random key key'' for the Test query session, and genuine keys are used elsewhere, also in the additional confirmation steps of Π_{MAC} in the tested session.

In summary, we obtain that

$$\Pr\left[\mathbf{Exp}^{\mathsf{scrcy\text{-}lr},0}_{\Pi,\mathcal{A}}(n)=1\right]=\Pr\left[\mathbf{Exp}^{\mathsf{scrcy\text{-}lr},0}_{\Pi_{\mathsf{MAC}},\mathcal{A}_{\mathsf{MAC}}}(n)=1\right]$$

for the case $b=0$, and that

$$\left|\Pr\left[\mathbf{Exp}^{\mathsf{scrcy\text{-}lr},1}_{\Pi,\mathcal{A}}(n)=1\right]-\Pr\left[\mathbf{Exp}^{\mathsf{scrcy\text{-}lr},1}_{\Pi_{\mathsf{MAC}},\mathcal{A}_{\mathsf{MAC}}}(n)=1\right]\right|$$

is negligible for $b=1$, by the key secrecy of Π and the pseudorandomness of KDF. Using the freshness of our adversaries we conclude that $\mathcal{A}_{\mathsf{MAC}}$'s advantage (over $\frac{1}{2}$) must be negligible by the key secrecy of Π.

KEY CONFIRMATION. We first show that the transformed protocol achieves full $(\mathcal{S},\mathcal{C})$-key confirmation. Note that this time we cannot assume that the original protocol Π has this property.

Assume $\Pr\left[\mathbf{Exp}^{\mathsf{Pred}}_{\Pi,\mathcal{A}}(n)=1\right]$ has non-negligible probability for Pred equal to the full $(\mathcal{S},\mathcal{C})$-key confirmation predicate, i.e., the predicate evaluates to false in the final state.

Then there exists a server session ℓ that has accepted and $\ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}$, but there is no (partnered) client session that has accepted with the same key as ℓ. Since we can always abort the adversary against key confirmation if some server session without partner accepts, we can assume that there is no further oracle query after ℓ accepts, and that the key of ℓ is therefore not revealed and that both $\ell.\mathsf{pid}$ and $\ell.\mathsf{id}$ are not corrupted (note, if $\ell.\mathsf{id}$ would have been a corrupt server session before, it would have not accepted anymore). Furthermore, ℓ has no partnered session since such a session would have the same key because of Match security (except with negligible probability).

To proceed, we guess the server session ℓ for which full key confirmation is violated by picking its index randomly among the at most n_s many sessions. In the first game hop, we use key secrecy of the original protocol Π to replace the key key for ℓ by a random key. Since there is no partnered session, key is not required by any other session. The simulation is analogous to the proof of key secrecy for Π_{MAC}. In the second game hop, we apply KDF security to make $(\mathsf{key}', \mathsf{key}^c_{\mathsf{MAC}}, \mathsf{key}^s_{\mathsf{MAC}})$ random for ℓ. Finally, we construct a MAC forger $\mathcal{B}$ as follows. The forger $\mathcal{B}$ black-box simulates $\mathcal{A}$, guesses ℓ, and returns the MAC τ^c received in session ℓ together with the "message" sid as a forgery. The winning probability is non-negligible since the MAC must be valid for ℓ in order to accept, and the random key $\mathsf{key}^c_{\mathsf{MAC}}$ is used for the first time to verify the validity of the MAC.

Next, we show that Π_{MAC} provides almost-full $(\mathcal{C}, \mathcal{S})$-key confirmation. For this we first note that key-confirmation identifier binding holds trivially, since the extended protocol sets $\mathsf{kcid}' = \mathsf{sid}' = \mathsf{sid}$ such that partnered sessions with the same session identifier sid' also have identical key-confirmation identifiers. Furthermore, Match security of the underlying protocol implies that sessions with the same $\mathsf{kcid}' = \mathsf{sid}$ also hold the same key if they eventually accept. Hence, predicate KCIDbind is satisfied, except with negligible probability.

Assume next that $\Pr\left[\mathbf{Exp}^{\mathsf{Pred}}_{\Pi,\mathcal{A}}(n) = 1\right]$ has non-negligible probability for Pred equal to the almost-full $(\mathcal{C}, \mathcal{S})$-key confirmation predicate, i.e., the predicate evaluates to false in the final state. Then there exists a client session ℓ that has accepted with session identifier sid' and $\ell.\mathsf{pid} \notin \mathrm{Corr} \cup \{*\}$, but there is no server session that has set its key-confirmation identifier to sid', or if it has, then it has accepted a different key.

To proceed, we guess the client session ℓ for which almost-full key confirmation as above is violated. In the first game hop, we use key secrecy of the original protocol Π to replace the key key for ℓ by a random key. Since there is no partnered (completed) session in Π_{MAC} with the same $\mathsf{kcid}' = \mathsf{sid}$ and the same key key', key is not required by any other session. The simulation is now analogous to the proof of key secrecy for Π_{MAC}. In the second game hop, we once more apply KDF security to make $(\mathsf{key}', \mathsf{key}^c_{\mathsf{MAC}}, \mathsf{key}^s_{\mathsf{MAC}})$ random for ℓ. Finally, we construct a MAC forger $\mathcal{B}$ as follows. The forger $\mathcal{B}$ black-box simulates $\mathcal{A}$, guesses ℓ, and returns the MAC τ^s and sid as a forgery. The winning probability is non-negligible since the MAC must be valid for ℓ to accept, and the random key $\mathsf{key}^s_{\mathsf{MAC}}$ is used for the first time to check the validity of the the MAC. ■

We note that there are several variations to the above instantiation which, if done properly, can all be shown to be secure:

- One variation, followed, e.g., by NIST [2], is to generate only a single MAC key and let the client and server prepend a distinct key word in front of the "message" sid for the computation, e.g., "client"||sid and "server"||sid. Here the MAC would require to be one-time unforgeable, though.
- Another variant—which can be combined with the previous approach—is to use strongly unforgeable MACs (where it is also infeasible to create a new MAC for the same message), e.g., deterministic MACs where verification is done via re-computation, and define $\mathsf{sid}' = (\mathsf{sid}, \tau^s, \tau^c)$.
- A third variant is to include the session identifier sid in the key derivation, $\mathsf{KDF}(\mathsf{key}, \mathsf{sid})$. For public session identifiers and strong key derivation functions this cannot harm the security. On the other hand, the basic security properties of the underlying protocol Π already tie session identifiers to session keys strongly, such that further linking through KDF seems unnecessary.
- If the session identifier is already strongly tied to the session keys, it is also possible to remove sid from the input of the MACs. This variant together with a single MAC key and "client"/"server" labels is used in HMQV-C [23], the 3-message variant of the HMQV protocol.

VI. Conclusion

Key confirmation is a widely targeted functional property in cryptographic key exchange protocols, but has received only very limited formal treatment so far. Our work provides the first extensive formalization of key confirmation through according security definitions in the game-based modeling tradition. A particular strength of our notions for key confirmation is their definitional modularity: they can easily be plugged together with more traditional notions like key secrecy to obtain combined security guarantees.

On the practical side, our analysis of the current TLS 1.3 `draft-10` full handshake validates that it achieves the design goal of providing key confirmation. As a potentially surprising technical result, key confirmation in the full handshake however does not require the exchange `Finished` dedicated to this purpose in the specification, but can readily be established from the exchanged `CertificateVerify` messages. While this result cannot serve as an argument for dropping the `Finished` messages due to their use in further handshake variants, it casts light on the cryptographic strengths of various handshake messages in general and the session hash and online signatures concepts integrated into TLS 1.3 in particular.

Our generic transform on the theoretical side validates the wide-spread concept of achieving key confirmation through exchanged MACs over the protocol transcript, which not

only formalizes a folklore approach but more importantly can serve as a verified blueprint for future key exchange protocol designs.

In future work we plan to investigate the extension of our approach to the (extended) Canetti–Krawczyk model for key exchange and the interplay between key confirmation and forward secrecy. For example, it might be possible to generalize the results obtained for the security strengthening compilers introduced in [16] and [11] to other methods providing key confirmation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for valuable comments. Marc Fischlin is supported by the Heisenberg grant Fi 940/3-2 of the German Research Foundation (DFG). This work was done in part while Marc Fischlin was visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467. Benedikt Schmidt is supported by ONR grant N00014-15-1-2750 and the European Commission's Seventh Framework Programme Marie Curie Cofund Action AMAROUT II (grant no. 291803). This work has also been co-funded by the DFG as part of project S4 within the CRC 1119 CROSSING, the European Union Seventh Framework Programme (FP7/2007-2013) grant agreement 609611 (PRACTICE), and ERC Advanced Grant ERC-2010AdG-267188-CRIPTO.

REFERENCES

[1] E. Barker, L. Chen, A. Regenscheid, and M. Smid. SP 800-56B. Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography. NIST Special Publication, National Institute of Standards & Technology, Aug. 2009.

[2] E. Barker, L. Chen, A. Roginsky, and M. Smid. SP 800-56A r2. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography. NIST Special Publication, National Institute of Standards & Technology, May 2013.

[3] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In B. Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 139–155. Springer, Heidelberg, May 2000.

[4] M. Bellare and P. Rogaway. Entity authentication and key distribution. In D. R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, Heidelberg, Aug. 1994.

[5] F. Bergsma, B. Dowling, F. Kohlar, J. Schwenk, and D. Stebila. Multi-ciphersuite security of the secure shell (SSH) protocol. In G.-J. Ahn, M. Yung, and N. Li, editors, *ACM CCS 14: 21st Conference on Computer and Communications Security*, pages 369–381. ACM Press, Nov. 2014.

[6] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy*, pages 98–113. IEEE Computer Society Press, May 2014.

[7] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459. IEEE Computer Society Press, May 2013.

[8] S. Blake-Wilson, D. Johnson, and A. Menezes. Key agreement protocols and their security analysis. In M. Darnell, editor, *Crytography and Coding*, volume 1355 of *Lecture Notes in Computer Science*, pages 30–45. Springer Berlin Heidelberg, 1997.

[9] S. Blake-Wilson and A. Menezes. Authenticated Diffie-Hellman key agreement protocols (invited talk). In S. E. Tavares and H. Meijer, editors, *SAC 1998: 5th Annual International Workshop on Selected Areas in Cryptography*, volume 1556 of *Lecture Notes in Computer Science*, pages 339–361. Springer, Heidelberg, Aug. 1999.

[10] S. Blake-Wilson and A. Menezes. Unknown key-share attacks on the station-to-station (STS) protocol. In H. Imai and Y. Zheng, editors, *PKC'99: 2nd International Workshop on Theory and Practice in Public Key Cryptography*, volume 1560 of *Lecture Notes in Computer Science*, pages 154–170. Springer, Heidelberg, Mar. 1999.

[11] C. Boyd and J. Nieto. On forward secrecy in one-round key exchange. In L. Chen, editor, *Cryptography and Coding*, volume 7089 of *Lecture Notes in Computer Science*, pages 451–468. Springer Berlin Heidelberg, 2011.

[12] C. Brzuska. *On the Foundations of Key Exchange*. PhD thesis, Technische Universität Darmstadt, Darmstadt, Germany, 2013. http://tuprints.ulb.tu-darmstadt.de/3414/.

[13] C. Brzuska, M. Fischlin, B. Warinschi, and S. C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM CCS 11: 18th Conference on Computer and Communications Security*, pages 51–62. ACM Press, Oct. 2011.

[14] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In B. Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 453–474. Springer, Heidelberg, May 2001.

[15] L. Chen and C. Kudla. Identity based authenticated key agreement protocols from pairings. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003)*, pages 219–233. IEEE Computer Society, June 2003.

[16] C. J. F. Cremers and M. Feltz. Beyond eCK: Perfect forward secrecy under actor compromise and ephemeral-key reveal. In S. Foresti, M. Yung, and F. Martinelli, editors, *ESORICS 2012: 17th European Symposium on Research in Computer Security*, volume 7459 of *Lecture Notes in Computer Science*, pages 734–751. Springer, Heidelberg, Sept. 2012.

[17] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[18] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In I. Ray, N. Li, and C. Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer and Communications Security*, pages 1197–1210. ACM Press, Oct. 2015.

[19] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081, 2016. http://eprint.iacr.org/2016/081.

[20] EMVCo LLC. EMV ECC key establishment protocols. http://www.emvco.com/specifications.aspx?id=243, 2012.

[21] K. Hoeper and L. Chen. SP 800-120. Recommendation for EAP Methods Used in Wireless Network Access Authentication. NIST Special Publication, National Institute of Standards & Technology, Sept. 2009.

[22] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 273–293. Springer, Heidelberg, Aug. 2012.

[23] H. Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 546–566. Springer, Heidelberg, Aug. 2005.

[24] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448. Springer, Heidelberg, Aug. 2013.

[25] B. A. LaMacchia, K. E. Lauter, and A. Mityagin. Stronger security of authenticated key exchange. In *Provable Security, First International Conference, ProvSec 2007*, volume 4784 of *Lecture Notes in Computer Science*, pages 1–16. Springer, Nov. 2007.

[26] A. Langley. Comment at the Real World Crypto (RWC) Workshop, New York, Jan. 2014.

[27] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, 1996.

[28] P. Morrissey, N. P. Smart, and B. Warinschi. The TLS handshake protocol: A modular analysis. *Journal of Cryptology*, 23(2):187–223, Apr. 2010.
[29] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 – draft-ietf-tls-tls13-10. https://tools.ietf.org/html/draft-ietf-tls-tls13-10, Oct. 2015.
[30] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 – draft-ietf-tls-tls13-11. https://tools.ietf.org/html/draft-ietf-tls-tls13-11, Dec. 2015.
[31] TLS Mailing List. Subject: Kill Finished (and other tricks for hardware). https://www.ietf.org/mail-archive/web/tls/current/msg12162.html, Apr. 2014.
[32] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006.