Proving the TLS Handshake Secure (As It Is)

Karthikeyan Bhargavan[1], Cédric Fournet[2], Markulf Kohlweiss[2],
Alfredo Pironti[1], Pierre-Yves Strub[3], and Santiago Zanella-Béguelin[1,2]

[1] INRIA, Paris, France
{firstname.name}@inria.fr
[2] Microsoft Research, Cambridge, UK
{fournet,markulf}@microsoft.com
[3] IMDEA Software Institute, Madrid, Spain
pierre-yves@strub.nu

Abstract. The TLS Internet Standard features a mixed bag of cryptographic algorithms and constructions, letting clients and servers negotiate their use for each run of the handshake. Although many ciphersuites are now well-understood in isolation, their composition remains problematic, and yet it is critical to obtain practical security guarantees for TLS, as all mainstream implementations support multiple related runs of the handshake and share keys between algorithms.

We study the provable security of the TLS handshake, as it is implemented and deployed. To capture the details of the standard and its main extensions, we rely on miTLS, a verified reference implementation of the protocol. We propose new agile security definitions and assumptions for the signatures, key encapsulation mechanisms (KEM), and key derivation algorithms used by the TLS handshake. To validate our model of key encapsulation, we prove that both RSA and Diffie-Hellman ciphersuites satisfy our definition for the KEM. In particular, we formalize the use of PKCS#1v1.5 and build a 3,000-line EasyCrypt proof of the security of the resulting KEM against replayable chosen-ciphertext attacks under the assumption that ciphertexts are hard to re-randomize.

Based on our new agile definitions, we construct a modular proof of security for the miTLS reference implementation of the handshake, including ciphersuite negotiation, key exchange, renegotiation, and resumption, treated as a detailed 3,600-line executable model. We present our main definitions, constructions, and proofs for an abstract model of the protocol, featuring series of related runs of the handshake with different ciphersuites. We also describe its refinement to account for the whole reference implementation, based on automated verification tools.

1 Introduction

TLS is the most widely deployed protocol for securing communications and yet, after two decades of attacks, patches and extensions, its practical security remains unresolved. One of the most troublesome aspects of the protocol is its handling of a large number of cryptographic algorithms and constructions. New

J.A. Garay and R. Gennaro (Eds.): CRYPTO 2014, Part II, LNCS 8617, pp. 235–255, 2014.

extensions are added to the protocol and its implementations, while older features remain for backward compatibility. Thus, TLS clients and servers offer many choices, and each run of the handshake involves a negotiation of the best protocol version, ciphersuite, and extensions available at both ends. Such a tradeoff between flexibility and security creates several problems:

(1) It makes the security of TLS depend on its correct configuration, inasmuch as some versions (e.g. SSL2) and algorithms (e.g. MD5 and RC4) are much weaker than others, and may also suffer from different implementation flaws. In theory, only very restrictive configurations have been proved secure. In practice, dangerous mis-configurations of TLS are commonplace.

(2) It complicates the protocol logic, as the integrity of the negotiation itself relies on algorithms being negotiated; this is a persistent source of attacks, from protocol regression in SSL2 [27] to version fallback in current browsers [18].

(3) It demands stronger security assumptions, to reflect the fact that honest parties may use the same key materials with different algorithms. Intuitively, TLS *on its own* enables a range of chosen-protocol attacks whereby a weak algorithm (chosen by the attacker) may compromise the security of stronger algorithms (chosen by honest parties). We detail below several constructions of TLS that demand joint assumptions on collections of algorithms. Surprisingly, prior work on the provable security of TLS failed to make this observation or left it implicit.

Besides interference between multiple algorithms, TLS features dependencies between multiple runs of the handshake. For instance, a client connection may first run an RSA-based session to establish a master secret and keys for the record layer, then run a second session on the same connection, possibly with different algorithms and certificates. Using a parallel connection, the client may run a third *resumption* handshake, re-using the master secret of a prior session to derive new keys. At that point, the security of those keys depends on algorithms and constructions used in three runs of the handshake. (See for instance [5] for recent attacks involving three related handshakes.) This is in sharp contrast with prior work on the provable security of TLS [13, 16, 17], which focus on a fixed run of the protocol, for a fixed choice of algorithms.

1.1 Cryptographic Agility

Agile security considers families of schemes or protocols, all serving the same purpose, when the same keys are shared across members of the family. Acar et al. [1] propose agile definitions for pseudo-random functions (PRF) and encryption schemes, and advocate agility as a major practical concern for protocols like TLS. Instead, *combined,* or *joint security* [12] studies the sharing of keys between constructions serving different purposes, e.g. encryption and signing. TLS requires both agile and joint security; in the remainder we let the term *agility* encompass both concepts.

The agility mechanisms of TLS ares primarily driven by *ciphersuites* of the form TLS_e_s_WITH_r, which indicates a key encapsulation mechanism (KEM) e

and signature scheme s for the handshake, and an authenticated encryption scheme r for the record layer. For instance, the commonly-used ciphersuite `TLS_RSA_WITH_AES_256_CBC_SHA` indicates an RSA handshake: the client sends a fresh premaster secret encrypted under the server public key; both parties use it to extract a master secret, used in turn as the seed of a SHA1-based PRF to derive 4 keys for SHA1-based MACs and AES encryption in CBC mode. TLS 1.2 currently has 314 registered ciphersuites. More precisely, the choice of algorithms depends on additional data exchanged during the handshake (hence subject to active attacks), including protocol versions, certificate requests, certificate chains, and various extensions in the first two messages of the handshake (e.g. for choosing hash functions and elliptic curves). Still, because of key reuse across algorithms, we stress that the security of TLS does not reduce to the security of a few thousand fixed-algorithm variants of the handshake.

1.2 Empirical Study of Web Servers and Browsers

Using an online analyzer [24], we gathered extended information on server configurations for 215 of the top 500 domains,[1] including the TLS versions, ciphersuites, certificates, and extensions they offer. These servers accept 64 ciphersuites, with an average of 12 and standard deviation of 6. They still widely deploy weak algorithms: 70% accept at least one ciphersuite with MD5 and 90% at least one with RC4. All servers but one offer several versions; 37% offer only SSL3 and TLS 1.0; 56% offer all 4 versions from SSL3 to TLS 1.2. Although now forbidden by the standard, 3% still accept SSL2.

We also tested 12 TLS clients, including major web browsers (Chrome, Firefox, Internet Explorer, Safari) and libraries (NSS, OpenSSL, SChannel, Secure Transport). These clients similarly propose a large number of ciphersuites, ranging from 19 to 36; they all propose weak hash (MD5) or encryption methods (RC4, or even no encryption).

1.3 Cross-Ciphersuite Attacks

As a first example, most TLS servers are configured to use the same RSA certificate both for signing handshake messages and for decrypting premaster secrets. Experimentally, 69% of the servers we tested propose at least one ciphersuite using RSA for encryption and one using it for signing, and *all* 138 of those use the same key for both purposes.

As a second example, Mavrogiannopoulos et al. [20] report a cross-protocol attack between plain Diffie-Hellman (DH) and Elliptic-Curve Diffie-Hellman (ECDH) ciphersuites, due to a mis-interpretation of the signed group description sent by the server. Each family of ciphersuites is (a priori) secure in isolation, but configurations enabling a DH client and an ECDH server are subject to their attack.

[1] `http://www.alexa.com/topsites/global`, as of January 2014, excluding domains with no valid HTTPS certificate.

Our third example concerns the record algorithms (the r in TLS_e_s_WITH_r). Recall that both parties derive keys for r immediately after the KEM phase, and start using them before verifying the Finished messages that confirm the integrity of the handshake. As an optimization, the optional False Start TLS extension [19] lets clients send private application data before key confirmation. Depending on r, the *same* key materials are split into IVs, MAC keys, and encryption keys of various lengths. Hence, the client and the server may start using the same bits with different algorithms $r_{\mathcal{C}}$ and $r_{\mathcal{S}}$, for instance as an IV at the client and as a MAC key at the server. To our knowledge, we are the first to report this cross-algorithm attack against [19]. We do not have an exploit based on two standard record algorithms $(r_{\mathcal{C}}, r_{\mathcal{S}})$ but one can easily design a pair of schemes strong in isolation and subject to the attack, and key recovery attacks against any standard algorithm $r_{\mathcal{C}}$ could be used to attack strong $r_{\mathcal{S}}$ algorithms.

1.4 Multiple Sessions and Connections

Following the standard, we recall TLS terminology for multiple related handshakes; this differs from the key-exchange model of Bellare & Rogaway [3] with only one kind of sessions and no shared state between sessions. Local instances of the protocol provide a *connection* (concretely, taking ownership of a TCP connection), either as client or as server. Each connection goes through a sequence of *epochs*, each epoch running one *handshake*. For a given connection, we refer to additional handshakes in the sequence as *renegotiations*. We refer to epochs performing full handshakes as *sessions*, and to epochs performing abbreviated handshakes as *resumptions*. We have a transition from the current epoch to the next each time a handshake *completes* by successfully processing the last message of the handshake. Abstractly, the local instance never stops; it is then ready to send (or receive) the first message of the next handshake.

Sessions intend to establish a fresh *master secret*, associated with data extracted from the handshake messages that record its origin and purpose, and used to derive fresh keys for the record layer. *Resumptions* instead rely on a prior complete session to save the cost of public-key cryptography and directly derive fresh keys using the algorithms and master secret of the original session. For each epoch, the handshake consists of a series of messages exchanged using the current record-layer protection mechanisms, initially in the clear, then typically using authenticated encryption.

1.5 Proving the TLS Handshake Secure

The scope of this paper is the TLS handshake, as it is specified in the Internet Standard and (to a lesser extent) as it is commonly used. We model multiple, related sessions and connections, and the agility issues caused by multiple ciphersuites featuring RSA and DHE key exchanges. We also model unilateral and mutual authentication, based on RSA and (EC)DSA signatures. On the other hand, we do not cover static DH, PSK, and ECDHE key exchanges, and we do not investigate the joint usage of keys for signing and encryption. (See the full paper for their discussion.)

Our main result is provable security for a standard-compliant, reference implementation of the handshake, seen as a detailed cryptographic model of the protocol. Our provably-secure handshake code consists of 3,600 lines of F#. Its security relies on new agile assumptions, notably for its KEMs. We reduce them to lower-level assumptions on RSA encryption and Diffie-Hellman exchange, using a 3,000-line EasyCrypt [2] proof. Working with a reference implementation, and testing it against mainstream implementations, forces us to handle the details of multiple handshakes and algorithms. Proving it secure requires both modularity and automation.

A feature of TLS that traditionally resists abstraction is that the handshake releases algorithms and derived keys to the record layer *before* the handshake completes, so that its last messages can be exchanged as TLS fragments protected by the new keys. We revisit the cryptographic folklore that the handshake can only be proved secure by including these encrypted messages. The kernel of the lore is that it cannot be proved using a Bellare & Rogaway-style key-exchange definition. To achieve modularity, we separate record-key generation from handshake completion: our main definition releases the record keys in the middle of the handshake, before signaling its completion a few messages later. Since the handshake does *not* rely on record-layer protection, we can safely let the handshake adversary control both the network and the record layer. Completion is still necessary to confirm that the record keys are secure before encrypting any application data—but not for encrypting handshake Finished messages.

We stress that this paper establishes the security of the *handshake*, seen as a component of TLS, not the full communications protocol. Our main construction provides key indistinguishability, and ensures agreement on parameters for the record layer. Our results complement those of Bhargavan et al. [4], who describe miTLS, an implementation of TLS verified in the computational model of cryptography; they focus on the main TLS API and application security, but rely on stronger, ad hoc assumptions for RSA and Diffie-Hellman ciphersuites. Our handshake is integrated with miTLS, which provides additional definitions and verified code for the record layer and the protocol logic. (Their security model ensures in particular that the record keys are used for protecting application data only after handshake completion [4].) By composing our results with theirs, we obtain security for a reference implementation of the TLS standard and the sample applications built and verified on top of miTLS.

1.6 Overview of the Paper

We see the use of a verified reference implementation and automated tools as essential to precisely account for multiple related epochs and algorithms in TLS; §6 briefly describes our use of high-level programming, type systems, and provers to carry out modular cryptographic verification at this scale. To present our result and explain its proof structure, however, we rely on more succinct definitions and constructions, given in §2–5 and outlined below. This more abstract treatment suffices to convey the main ideas, but it necessarily omits many aspects of the handshake, such as its message formats. We refer to the standard [9] or the implementation for the details. Also, for simplicity, we do not model forward secrecy and state reveal e.g. for master secrets, and we consider only static compromise for long-term keys.

Signatures (§2) and Certificates. We begin with a relatively simple agile definition. TLS supports three core signature algorithms, $s \in \{RSA, DSA, ECDSA\}$, used with a range of algorithms h to hash the text before signing. The hash algorithm depends on protocol versions, ciphersuites and extensions. TLS does not enforce any key-based hash algorithm policy, so we need a notion of security that tolerates *some* weak algorithms in the standard. For instance, a verifier tricked into using MD5 may remain secure, provided the signer only uses SHA1, and vice-versa. For each core algorithm s, we define h^*-H-security against an adversary that must forge a valid signature for algorithms (s, h^*), given access to signing oracles for any algorithms (s, h) with $h \in H$. We show that a family of secure schemes may not be jointly secure, but we leave open its concrete analysis for the range of algorithms used in TLS.

Our model excludes any validation rules for certificates and their PKI, an important problem outside the scope of the TLS standard. Our constructions simply authenticate the exchanged certificate chains, and use a specification function to extract from them the public keys used in the handshake.

Master Secrets, Key Encapsulation, and Key Derivation (§3). Following Krawczyk et al. [17], we use KEMs [8] to model key-exchange; this allows us to unify RSA and Diffie-Hellman within the same formalism. Instead of treating the whole handshake as a KEM, however, following Morrissey et al. [21], we decompose it into *premaster secret*, *master secret*, and *record-key derivation* phases; this yields the modularity we need e.g. for modeling the re-use of master secrets between handshakes. We show how to securely construct a master secret KEM from a premaster secret KEM for RSA and Diffie-Hellman ciphersuites (Theorem 1) and, independently, how to derive record keys and Finished messages from master secrets (see the full paper). We formalize the proof of Theorem 1 in EasyCrypt. For RSA, this involves showing that countermeasures to Bleichenbacher's attacks [6, 15] provide enough protection against chosen-ciphertext attacks. We rely on the assumption that PKCS#1v1.5 ciphertexts are hard to re-randomize; we leave open the problem of further reducing this conjecture to standard RSA assumptions. Our result does not directly compare to the one of Krawczyk et al. as their KEM also includes key derivation and Finished messages, whereas we rely on this new assumption. To comply with the standard, we also support agility in the algorithm used to extract master secrets from a premaster secrets. As for agile signatures, we arrive at a definition parameterized by an algorithm for the encryptor and a set of algorithms for the decryptor.

Once established, the master secret is used to key a pseudo-random function (PRF) for multiple epochs for two purposes: (1) to derive the record-layer key materials for the epoch; and (2) to compute the MACs of all messages exchanged in an epoch to verify its integrity. Our corresponding security definition (in the full paper) requires that adversaries commit to a record-layer algorithm r before key derivation. This let us support the negotiation of r without having to make agile assumptions for the record layer, as discussed in §1.3.

Agile Security Model (§4) and Proof (§5) for Sequences of Handshakes. The main two goals of the handshake are to establish shared keys for the record layer, and to agree on many parameters, including those used in the handshake itself. To this end, we propose a new security definition that covers multiple epochs on different connections, related by resumptions and renegotiations. We equip our adversary (informally including the rest of TLS, the application, and the network) with oracles to create honest connections and long-term keys for clients and servers, to control their usage, and to exchange handshake messages. Each honest instance of the protocol represents a connection, and logs a sequence of *local assignments*, recording its view on the successive epochs of the connection. This enables us to capture TLS assignments in a generic manner. Our main integrity result is that, when a handshake completes, and under suitable conditions on algorithms and keys, honest clients and servers agree on all assignments for all epochs on the connection. More explicitly, for new sessions, both parties agree on a unique label; the negotiation algorithms, parameters, and key-exchange values; and the optional certificate chains for the client and the server. For resumptions, both parties agree on the label of the session being resumed, as well as a fresh unique label for key derivation.

We also provide secure key derivation, depending on distinguished exchange-value assignments for each ciphersuite. They are somewhat similar to session identifiers in Bellare-Rogaway models but are used to define both *safety*, akin to freshness, and partnering. A session is *safe* when honest client and server agree on these assignments, under suitable conditions on algorithms and long-term keys. As discussed above, our definition immediately releases all connection keys. We guarantee that the keys of safe sessions are indistinguishable from fresh random keys; this accounts for selective session key reveal and test queries in Bellare-Rogaway models. Additionally, we provide *verified safety*, that is, sufficient conditions on the recorded long-term keys that enable honest parties to infer that their session is safe.

Our main result (§5, Theorem 2) reduces the concrete security of the TLS handshake to agile assumptions on the constructions used for signatures, KEMs, and PRFs. Each epoch assigns a distinguished agility-parameter a, selecting all algorithms for the epoch. The theorem statement is parameterized by a predicate α on a that holds whenever all algorithms selected by a are (assumed to be) secure. Thus, it provides meaningful security only for epochs where $\alpha(a)$ holds, despite any other epochs. If α is always false, there is nothing to prove. If we care specifically about one ciphersuite, say `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA`, we may apply our theorem with α set to true only when a selects that ciphersuite. This already improves on non-agile results for TLS that assume all honest parties agree *in advance* on a ciphersuite and reject any others.

Our model accounts for agility with respect to record algorithms, and yields channel security for miTLS without agile assumptions on the algorithms r used in the record layer. We thus validate the use of stateful LHAE [23] for clients and servers that negotiate r. We require, however, that no application data be sent before the Finished messages are verified. For implementations that violate this requirement [19], stronger agile assumptions seem unavoidable.

Code-Based Verified Implementation (§6). We finally present the reference implementation of the handshake we integrated into miTLS, and its verification against our security definition, based on the same modular proof structure but at a greater level of detail, relying on type-based verification for scalability. Our code supports the standard and commonly-used extensions; we tested it against various mainstream TLS clients and servers, using 4 versions ranging from SSL3 to TLS 1.2, 12 ciphersuites, and various subsets of extensions. It improves on the original miTLS code [4], which supported less features, and whose security relied on monolithic, TLS-specific assumptions for RSA and DH ciphersuites. The full paper reports experimental results showing that our code runs handshakes with reasonable performance. To enable its automated verification, our code is structured into small, independent modules (that is, program libraries) parameterized by algorithm descriptors. For instance, our library code for the HMAC-based PRF used in TLS implements agility before calling selected core algorithms, e.g. SHA1. In contrast, the code that implements SHA1 is outside the scope of our verification effort—we document our agile cryptographic assumption on it, and call a standard library. Each cryptographic construction used in the handshake corresponds to a separate library in the code. We define the security of libraries for multiple keys and multiple algorithms; the corresponding definitions and reductions to single-key security of individual algorithms appear in the full paper.

In summary, our work sheds light on important design and implementation issues of TLS. To our knowledge, we provide the first provable-security results for TLS that account for algorithm agility. We are also the first to give an abstract security model for handshakes related by resumption and renegotiation.

Further Reading. Our website `http://www.mitls.org` provides additional materials: the miTLS source code; the EasyCrypt proof of Theorem 1; and a companion paper with empirical data on TLS handshakes, auxiliary definitions, constructions, and proofs, and extended discussions of attacks and related work.

2 Agile Signatures

An *agile signature scheme* consists of three algorithms: KeyGen is a standard key generation algorithm, while Sign and Verify take an extra agility parameter. For instance, given a core signature scheme $s = (\mathsf{keygen}, \mathsf{sign}, \mathsf{verify})$, the hash-then-sign scheme $S_s = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ of TLS is defined as follows: $\mathsf{KeyGen} \triangleq \mathsf{keygen}$ generates a key pair for algorithm s; $\mathsf{Sign}(h, sk, m) \triangleq \mathsf{sign}(sk, h(m))$ computes a signature using the core scheme s and hash algorithm h; and $\mathsf{Verify}(h, pk, m, \sigma) \triangleq \mathsf{verify}(pk, h(m), \sigma)$ verifies a purported signature σ for message m

hashed with algorithm h. We define existential unforgeability under chosen-message attacks (EUF-CMA) for agile signatures.

Definition 1 (EUF-CMA). *Let* $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *be an agile signature scheme,* $p^\star$ *a parameter, and* P *a set of parameters, and consider the following forgery game:*

```
Game $\mathsf{EUF} \triangleq$
    $pk, sk \leftarrow \mathsf{KeyGen}();\; M := \varnothing$
    $m', \sigma \leftarrow \mathcal{A}^{\mathsf{SIGN}}(pk)$
    return $m' \notin M \wedge \mathsf{Verify}(p^\star, pk, m', \sigma)$

Oracle $\mathsf{SIGN}(p, m) \triangleq$
    if $p \notin P$ then return $\perp$
    $M := M \cup \{m\}$
    return $\mathsf{Sign}(p, sk, m)$
```

The scheme is $(\epsilon, t, p^\star, P)$*-secure against EUF-CMA if, for any* $\mathcal{A}$ *that runs in time* t*, the* EUF *game returns* true *with probability at most* ϵ*.*

This definition generalizes plain EUF-CMA security; the two coincide for a scheme with fixed hash algorithm h, i.e. $(p^\star, P) = (h, \{h\})$. We do not require $p^\star \in P$; for instance, one may pragmatically assume that forging an MD5-based signature is hard when given only SHA1-based signatures. Indeed, the attacks of Stevens et al. [26] rule out (MD5, {MD5, ...})-security, but (MD5, {SHA1})-security may still hold. On the other hand, non-agile security does not imply agile security. Consider for instance the scenario where the pre-image security of MD5 is broken. Then the attacks described by Naccache and Shparlinski [22] are likely to break (SHA256, {MD5, SHA256})-security, even though (SHA256, {SHA256})-security would still hold.

The TLS standard features the following hash-then-sign schemes: prior to version 1.2, RSA PKCS#1v1.5 signatures use the concatenation of MD5 and SHA1 hashes and (EC)DSA signatures use SHA1. TLS 1.2 introduces additional agility to facilitate migration from MD5 and SHA1 to stronger algorithms. Designers are aware of agility problems, and prescribe ad hoc countermeasures [9, §7.4.3]. The standard still requires that (EC)DSA use SHA1, delaying the migration to stronger algorithms. It also adds an encoding of the hash algorithm identifier to guarantee that all hash algorithms have disjoint range.

Given algorithms h and h' with disjoint ranges, if the core signature scheme itself is (ϵ, t)-EUF-CMA secure on their joint range, then we have $(\epsilon', t', h, \{h, h'\})$-security for the corresponding agile hash-then-sign signature scheme, where the difference between ϵ, t and ϵ', t' depends on the reduction to the collision resistance of h. Sadly, the core signature schemes used in TLS are not EUF-CMA secure. The best we can do, for now, is thus to assume that the hash-then-sign signature scheme that uses them meets Definition 1.

3 Master Secrets and Key Encapsulation

Following [14, 17], we model the basic key-exchange functionality of TLS as different variations on KEMs. However, we separate the derivation of the master secret from the derivation of keys for the record-layer. We model the premaster secret phase for RSA and Diffie-Hellman exchanges as agile KEMs (keygen, !enc,dec) parameterized by a 2-byte protocol version string.

RSA. keygen generates a fresh RSA key pair $(pk,\ sk)$; $\mathsf{enc}(pv, pk)$ appends a randomly chosen 46-byte string to pv to obtain the premaster secret pms, and returns it with the ciphertext c resulting from its PKCS#1v1.5 encryption under pk; $\mathsf{dec}(pv, sk, c)$ decrypts c with sk. If the padding is correct and the decrypted pms is exactly 48 bytes long, it returns pms with the first 2 bytes replaced by pv, otherwise it returns $\bot$; such errors are handled in our ms-KEM below.

Diffie-Hellman. keygen selects group parameters pp, generates a fresh pair of DH values $(g^x,\ x)$, and returns $pk = (pp, g^x)$ and $sk = (pk, x)$ as public and private KEM keys; $\mathsf{enc}(pv, (pp, g^x))$ samples y and returns $pms = g^{xy}$ and $c = g^y$; $\mathsf{dec}(pv, (pk, x), c)$ returns $c^x = g^{xy}$. The ciphertext space guarantees that c is in a large prime-order subgroup specified by pk. In contrast to the RSA pms-KEM, neither enc nor dec depend on pv.

On their own, these two premaster secret KEMs are *not* secure under any indistinguishability notion, even under relatively weak active attacks such as, for instance, plaintext-checking attacks (PCA): recall the Bleichenbacher attack, and the lack of active security for basic Diffie-Hellman (e.g., querying a plaintext-checking oracle on c^r and pms^r for any $r \neq 1$, suffices to distinguish a random pms from the one encapsulated in c). Rather than using pms as a key, TLS feeds it through an agile *key extraction function* (KEF) parameterized by a hash algorithm, to compute the master secret ms.

We model this phase of the handshake as an *agile labeled KEM*, extending the labeled KEMs of [14, 17] with an agility parameter. Given an agile (unlabeled) pms-KEM $e = (\mathsf{keygen}, \mathsf{enc}, \mathsf{dec})$ and an agile key extraction function family KEF, the master secret KEM $E_e = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ of TLS is defined as follows:

- $\mathsf{KeyGen}() \triangleq \mathsf{keygen}()$;
- $\mathsf{Enc}(pv, h, pk, \ell) \triangleq pms, c \leftarrow \mathsf{enc}(pv, pk);\ ms \leftarrow \mathsf{KEF}(pv, h, pms, \ell);$ $\textbf{return}\ ms, c$

  generates a premaster secret pms and a ciphertext c using e, then derives a master secret ms for ℓ using KEF.
- $\mathsf{Dec}(pv, h, sk, \ell, c) \triangleq pms \leftarrow \mathsf{dec}(pv, sk, c);\ \textbf{if}\ pms = \bot\ \textbf{then}\ pms \leftarrow pv \,\|\, \$;$ $\textbf{return}\ \mathsf{KEF}(pv, h, pms, \ell)$

  decrypts the ciphertext c to obtain pms. If decryption fails, it computes a fake pms by appending a random 46-byte string to pv (this is never the case for DH). It returns the value obtained from pms and ℓ using the agile KEF.

We assume sufficient checks to ensure that all arguments are well-formed before calling the master secret KEM algorithms; e.g., for Diffie-Hellman, our code validates group parameters and checks that pk and c belong to a large prime-order subgroup before calling Dec.

We define security for agile labeled KEMs as indistinguishability under replayable chosen-ciphertext attacks (IND-RCCA), a relaxation of CCA security, first introduced for public-key encryption by Canetti et al. [7].

Definition 2 (IND-RCCA). *Let* (KeyGen, Enc, Dec) *be an agile labeled KEM, $p^\star$ a parameter, P a set of parameters; and consider the following game:*

```
Game RCCA ≜
  $pk, sk \leftarrow \mathsf{KeyGen}()$
  $K, L := \varnothing$
  $b \leftarrow \{0,1\}$
  $b' \leftarrow \mathcal{A}^{\mathsf{ENC},\mathsf{DEC}}(pk)$
  return $(b' = b)$
```

```
Oracle ENC($\ell$) ≜
  if $\ell \in L$ then return $\bot$
  $k_0, c \leftarrow \mathsf{Enc}(p^\star, pk, \ell)$
  $k_1 \leftarrow \$$
  $K(\ell) := K(\ell) \cup \{k_0, k_1\}$
  return $k_b, c$
```

```
Oracle DEC($p, \ell, c$) ≜
  if $\ell \in L \vee p \notin P$ then return $\bot$
  $L := L \cup \{\ell\}$
  $k \leftarrow \mathsf{Dec}(p, sk, \ell, c)$
  if $k \in K(\ell)$ then return $\bot$
  return $k$
```

The RCCA advantage of $\mathcal{A}$, $\mathbf{Adv}^{\mathsf{RCCA}}_{p^\star,P}(\mathcal{A})$ is defined as $2\Pr[\mathsf{RCCA} : b' = b] - 1$. The scheme is $(\epsilon, t, p^\star, P)$-secure against IND-RCCA-n when the advantage of any adversary $\mathcal{A}$ running in time t and making at most n queries to ENC *is at most ϵ. We write IND-RCCA instead of IND-RCCA-*1.

The check $\ell \in L$ in the decryption oracle reflects a property of TLS: honest servers decrypt at most once for each nonce. The check $\ell \in L$ in the encryption oracle is analogous to the restriction of Krawczyk et al. [17] to define IND-CCCA security for non-agile KEMs.

The lemma below (proved by a standard hybrid argument in the full paper) enables us to prove security for a single query, then use the multi-query variant for reasoning about TLS in our main theorem.

Lemma 1. *If a KEM* (KeyGen, Enc, Dec) *is $({}^{\epsilon}/{}_{n}, t', p^\star, P)$-secure against IND-RCCA, then it is $(\epsilon, t, p^\star, P)$-secure against IND-RCCA-n, where $t' = t + O(n \cdot t_{\mathsf{Enc}})$ and t_{Enc} is the worst-case cost of algorithm* Enc*.*

Next, we define the assumptions for our main theorem on the TLS master secret KEM: *non-randomizability under plaintext-checking attacks* (NR-PCA) and *one-wayness under plaintext-checking attacks* (OW-PCA).

Definition 3 (NR-PCA, OW-PCA). *Let* (keygen, enc, dec) *be an agile (unlabeled) KEM, $p^\star$ a parameter, and P a set of parameters. Consider the two games:*

```
Game OW-PCA ≜
  $pk, sk \leftarrow \mathsf{keygen}()$
  $k^\star, c^\star \leftarrow \mathsf{enc}(p^\star, pk)$
  $k \leftarrow \mathcal{A}^{\mathsf{PCO}}(pk, c^\star)$
  return $(k = k^\star)$
```

```
Game NR-PCA ≜
  $pk, sk \leftarrow \mathsf{keygen}()$
  $k^\star, c^\star \leftarrow \mathsf{enc}(p^\star, pk)$
  $c \leftarrow \mathcal{A}^{\mathsf{PCO}}(pk, c^\star)$
  return $(c \neq c^\star \wedge k^\star = \mathsf{dec}(p^\star, sk, c))$
```

```
Oracle PCO($p, k, c$) ≜
  if $p \notin P \vee k = \bot$ then
      return $\bot$
  $k' \leftarrow \mathsf{Dec}(p, sk, c)$
  return $(k' = k)$
```

The NR-PCA advantage of $\mathcal{A}$, $\mathbf{Adv}^{\mathsf{NR\text{-}PCA}}_{p^\star,P}(\mathcal{A})$ is the probability that the NR-PCA *game returns* true*. The KEM is $(\epsilon, t, p^\star, P)$-secure against NR-PCA if the advantage of any adversary $\mathcal{A}$ running in time t is at most ϵ. OW-PCA advantage and security are defined analogously.*

The full paper gives preliminary theorems and conjectures on these assumptions, and relates our agile IND-RCCA KEMs to prior work and more standard assumptions. We hope this will stimulate further cryptanalytic work on TLS.

Our main result on KEMs is that the generic *ms*-KEM E_e of TLS is IND-RCCA secure if the underlying *pms*-KEM e is both NR-PCA and OW-PCA secure. The proof (in the full paper) has been formalized using EasyCrypt. The proof is in the random oracle model for the agile KEF. As explained above, we consider the single challenge case.

Theorem 1 (RCCA from NR-PCA and OW-PCA). *Let $\mathcal{A}$ be a $(p^\star, P)$-RCCA adversary for E_e running in time $t_{\mathcal{A}}$ and making at most q_{KEF} and q_{DEC} queries to the random and decryption oracle, respectively. Let $p^\star = (pv^\star, h^\star)$ and $P' \triangleq \{pv \mid (pv, h) \in P\}$. There exist an OW-PCA adversary $\mathcal{B}$ and an NR-PCA adversary $\mathcal{C}$ against e, both running in time $t_{\mathcal{A}} + O(q_{\mathsf{DEC}} \cdot q_{\mathsf{KEF}})$, such that*

$$\mathbf{Adv}^{\mathsf{RCCA}}_{p^\star, P}(\mathcal{A}) \leq 2\Big(\mathbf{Adv}^{\mathsf{NR\text{-}PCA}}_{pv^\star, P'}(\mathcal{B}) + \mathbf{Adv}^{\mathsf{OW\text{-}PCA}}_{pv^\star, P'}(\mathcal{C}) + 2^{|pv|-|pms|}\left(q_{\mathsf{KEF}} + q_{\mathsf{DEC}}\right)\Big).$$

The factor $2^{|pv|-|pms|}$ is the entropy of the value $pv \,\|\, \$$ used to derive the master secret when RSA decryption fails, as recommended by TLS 1.2 to mitigate Bleichenbacher attacks. With the DH *pms*-KEM, decryption never fails (as the ciphertext validation is done beforehand) so the last term above can be omitted.

4 Defining Agile Security for Sequences of Handshakes

Our security definition for handshakes is general enough to apply to TLS, as specified in the standard and coded in miTLS, while hiding implementation details like message formats and specific cryptographic constructions. The adversary creates and interacts with multiple instances i of a handshake protocol Π by calling Π's oracles, detailed below. Each instance has a fixed role $\mathcal{R}$, either $\mathcal{C}$ for Client or $\mathcal{S}$ for Server, and models a connection endpoint.

- $\mathsf{KeyGen}(v)$ creates and stores a new honest keypair for the long-term public-key algorithm v (in TLS, ranging over s for signing and e for key encapsulation) and returns the associated public key. Similarly, $\mathsf{KeyInject}(v, pk, sk)$ stores a dishonest keypair (assuming pk is not yet in the store).
- $\mathsf{Init}(\mathcal{R}, cfg_{\mathcal{R}})$ creates an instance with role $\mathcal{R}$ and local configuration $cfg_{\mathcal{R}}$; it returns a fresh handle i.
- $\mathsf{Send}_i(frag)$ lets an existing instance i process a fragment, depending on its current state. As a result, the instance may update its state, assign local variables, and return a response. (In TLS, responses range over sequences of handshake and CCS message fragments, intended to be sent to the peer, as well as error messages.)
- $\mathsf{Control}_i(env)$ changes the global, internal state of the handshake, e.g., enabling the adversary to control access to stored sessions and private keys by the protocol the next time Send will be called, or to trigger a renegotiation request. This single oracle accounts for many control functions in the miTLS handshake implementation. For example, $\mathsf{Control}$ provides the environment with means to reject certificates that it deems invalid.

Each instance maintains its private local state (e.g. using local variables). Each instance can go through a sequence of epochs (e.g. recording the number of cycles in the state machine). For each epoch, it records a sequence of *variable assignments*, extended as the result of calls to Send and Control. Each variable is assigned at most once in every epoch. The selection and ordering of assignments within an epoch depends on the protocol; for instance, a client epoch may assign its client-certificate variable, then send a message to the server, causing the server epoch to record the same assignment later in the protocol.

Our definition is based on local variable assignments, which summarize the view of clients and servers so far about each epoch. This is adequate to model the handshake as a component within TLS, but this differs from models based on *matching conversations* [3] that compare the (unparsed) messages they have sent and received so far. We use assignments to express the main goals of the protocol, for instance assigning a fresh random value to the record key variable k; and agreeing on all assignments as a session completes. We list below the main variables used in our presentation, but our definition can account for a more detailed model of the TLS handshake.

ℓ epoch identifier; in TLS, the concatenation of the client and server random values.

$\ell_{session}$ resumption identifier; in TLS, the identifier of the epoch that completed the session being resumed. (The miTLS code also assigns the TLS *sessionId*, chosen by the server, but we do *not* use it as an identifier as it is not necessarily unique.)

$a_{\mathcal{C}}$, $a_{\mathcal{S}}$ client and server negotiation parameters; in TLS, they consist of protocol versions, ciphersuites, and extension messages.

a agility parameter; in TLS, the protocol version, the negotiated ciphersuite, and data extracted from the first flight of messages sent by the server.

$cert_{\mathcal{C}}$,$cert_{\mathcal{S}}$ client and server certificate chains. In TLS, these certificates are optional; e.g. the assignment $cert_{\mathcal{C}} := \bot$ denotes the absence of client certificate.

$ex_{\mathcal{C}}$, $ex_{\mathcal{S}}$ client and server exchange variables, possibly secret, used to specify safety.

k record key for the epoch; in TLS, depending on a, this key is usually split into 4 keys for MAC & encrypt.

complete successful completion flag, marking the end of the handshake for this epoch.

Unless explicitly mentioned for key-exchange materials, these variables are public: the adversary can read them, but not change them; the protocol can write them once in every epoch, but not read them. (This restriction matters only for the record key, as we replace it with a random value.) The agility-parameter variable a determines the algorithms and constructions used by the handshake. Our security properties are conditioned by a strength predicate $\alpha(a)$ that indicates whether those algorithms are strong enough to secure the epoch. When the role of an epoch is clear from the context, the *peer* refers to the opposite role, and the *peer-exchange variable* refers to the exchange variable of the opposite role (e.g. $ex_{\mathcal{C}}$ when $\mathcal{R}$ is $\mathcal{S}$).

We deliberately avoid modeling certificate validation. For the handshake, certificate chains are authenticated, uninterpreted bitstrings. We leave as future work supplementing our model with an application-level certificate infrastructure above the miTLS API. We assume given a public specification function $\mathsf{pk}(\mathit{cert})$ that returns either the public key associated with a certificate chain, or $\bot$. The session state does not need to explicitly mention public keys, but public keys can appear in exchange variables.

A security model for a protocol describes how queries are answered and how session variables are assigned. Next, we define properties of these models as they interact with an adversary.

Definition 4 (Honesty, Safety, Matching Algorithms and Completion). *For a handshake protocol Π and a strength predicate $\alpha(\cdot)$, an adversary that calls Π's oracles any number of times produces a trace of interleaved variable assignments for a series of epochs for each instance. In this trace:*

- *As determined by its assigned agility parameter a: an epoch is either a* session*, with distinguished* client- *and* server-exchange *variables, or a* resumption*, with an $\ell_{session}$ variable; sessions (and their exchange variables) are either* static *or* ephemeral*; a* static *session has at least one* static *exchange variable; an* ephemeral *session has only* ephemeral *exchange variables.*
- *A (long-term) public key is* honest *for algorithm v if it was returned by a call to $\mathsf{KeyGen}(v)$. A session's ephemeral server-exchange variable assignment is* honest *if there is a server session with the same assignment to its server-exchange variable—and conversely for ephemeral client-exchange variables.*
- *A client session is* safe *if (i) $\alpha(a)$ holds; (ii) honest public keys for a's algorithms are assigned to all static exchange variables; and (iii) there is a server session with the same assignment to the ephemeral server-exchange variable. A server session is* safe *if the converse holds.*
- *A resumption is* safe *if $\alpha(a)$ holds and $\ell_{session}$ is the identifier of a safe and complete session.*
- *An epoch has matching algorithm $r = \mathsf{record}(a)$ when there is a peer epoch with the same identifier ℓ and algorithm r.*
- *An epoch is* complete *when it includes the assignment $\mathit{complete} := 1$.*

Anticipating on §5, for TLS we define the client exchange value $ex_{\mathcal{C}}$ to be the master secret ms together with the KEM public key pk, and the server-exchange variable $ex_{\mathcal{S}}$ to be the public key pk of the KEM. The latter is static for TLS-RSA, but ephemeral for TLS-DHE. Here ms is explicitly secret and ephemeral.

Definition 5 (Handshake Security). *Let Π be a handshake protocol, $\alpha(\cdot)$ a strength predicate, and $\mathcal{A}$ an adversary that calls Π's oracles any number of times. Consider the following properties:*

(1) **Uniqueness:** *epoch identifiers are used at most once in each role.*
Let $\mathbf{Adv}^{\mathrm{U}}(\mathcal{A})$ be the probability that two different epochs with the same role assign the same value to ℓ when $\mathcal{A}$ terminates.

(2) **Verified Safety:** *if the peer of a session uses a strong signature algorithm to authenticate and the public-key for the peer signature is honest, then the peer-exchange variable assignment is honest.*

Let $\mathbf{Adv}^{\mathrm{S}}(\mathcal{A})$ *be the probability that, when* $\mathcal{A}$ *terminates, there is an epoch such that* $\alpha(a)$ *holds; the public key of the peer is honest; and the assignment to the peer exchange value is not honest (i.e. not assigned by any peer);*

(3) **Agile Key Derivation:** *depending on a random bit* b*, replace the record key assigned in safe epochs with matching algorithm* r *with a fresh* $k \leftarrow \mathsf{KeyGen}(r)$*, assigning the same value to epochs that have the same identifier* ℓ*, algorithms* $\mathsf{kdf}(a)$ *and exchange variables or resumption identifier.*
Let $\mathbf{Adv}^{\mathrm{K}}(\mathcal{A}) = 2p - 1$ *where* p *is the probability that* $\mathcal{A}$ *returns* b*.*

(4) **Agreement:** *for every safe and complete epoch, there is a safe epoch in the other role such that their two instances agree on all prior assignments.*
Let $\mathbf{Adv}^{\mathrm{I}}(\mathcal{A})$ *be the probability that, when* $\mathcal{A}$ *terminates: an instance created by* $\mathsf{Init}(\mathcal{R}, \mathit{cfg})$ *assigns* $\mathit{complete} := 1$ *in a safe epoch; and no instance created by* $\mathsf{Init}(\overline{\mathcal{R}}, \mathit{cfg}')$ *begins with a series of epochs with the same assignments to all variables (up to, but possibly excluding* $\mathit{complete} := 1$*).*

The handshake is (ϵ, t, α)*-secure when for any adversary* $\mathcal{A}$ *running in time* t*, we have* $\mathbf{Adv}^{\mathrm{G}}(\mathcal{A}) \leq \epsilon$*, for* $\mathrm{G} = \mathrm{U}, \mathrm{S}, \mathrm{K}, \mathrm{I}$*.*

Discussion. The properties above are given in chronological order: in TLS in particular, protocol instances first exchange fresh random values, then derive keys, and finally confirm the integrity of the session negotiation.

Property (1) simply ensures that ℓ provides a unique identifier, later authenticated using (4); we use these identifiers for matching client and server sessions.

Property (2) enables, for instance, a client that trusts both the negotiated algorithm and the server certificates to deduce that its server-exchange variable is honest, and conclude that its session is safe.

Property (3) idealizes the derived key; this is key indistinguishability. Recall that TLS uses the key before the two parties actually agree on the record algorithms. Conservatively, (3) idealizes the key only when the record algorithms match. As Krawczyk et al. [17], our model does not consider forward secrecy.

Property (4) guarantees agreement on all variable assignments at the client and server instances since their creation, not just the assignments of the current epoch. Hence, as soon as one epoch safely completes, the peers agree also on all prior epochs on that connection—even those that were not safe, or not verifiably safe. For TLS, this property holds only thanks to the (mandatory) secure renegotiation extension, which links each epoch to its predecessor. This property is closely related to the TLS renegotiation results of Giesen et al. [11]. They additionally propose an extension of TLS that would guarantee agreement on the full stream of application data, not just the handshake epochs. On the other hand, our model and security definition also cover resumptions and RSA ciphersuites, which are not covered by their results. Unlike previous analyses of TLS, our definition accounts for session resumptions. Property (4) guarantees agreement on the new epoch identifier ℓ and the identifier $\ell_{session}$ of the resumed session (and hence on the new record keys), as long as the original session is safe. The epochs of the original session may be on a different connection, between a different pair of instances; for those instances, safety for the original session independently guarantees agreement on all its original variable assignments.

TLS applications often group connections that use the same session or the same long-term key, allowing them to share resources and access rights. For example, web browsers allow all connections to the same server to share resources via the Same Origin Policy. It may seem desirable to guarantee a strong relationship between such connections, but our Property (4) guarantees agreement only for the sequence of epochs over a single connection. Indeed, the natural extension of this property to multiple connections does not hold for TLS, as shown by the triple handshake attack of Bhargavan et al. [5]. In this attack, an unsafe server-authenticated session is resumed on a new connection and then renegotiated with a new safe mutually-authenticated session. For the new safe epoch, Property (4) retroactively guarantees agreement on the prior resumption, but *not* on the original unsafe session that was resumed. Consequently, it is possible for a client and server instance to have a safe epoch but inconsistent variable assignments for the session associated with a prior resumed epoch; this leads to a variety of attacks, similar to the renegotiation attacks of Ray [25]. A stronger agreement can be achieved either at the application level, by checking agreement on prior connections, or by a protocol extension that includes a hash of the log of the original session in resumption handshakes [5]; we leave the modeling of this extension and its security for future work.

Compared with classic key exchange models [3] and the key exchange part of ACCE [13], our definition yields useful additional properties. Property (4) guarantees agreement on the negotiation parameters $a_{\mathcal{C}}$ and $a_{\mathcal{S}}$ for safe and complete epochs, thereby preventing version and ciphersuite rollback attacks.

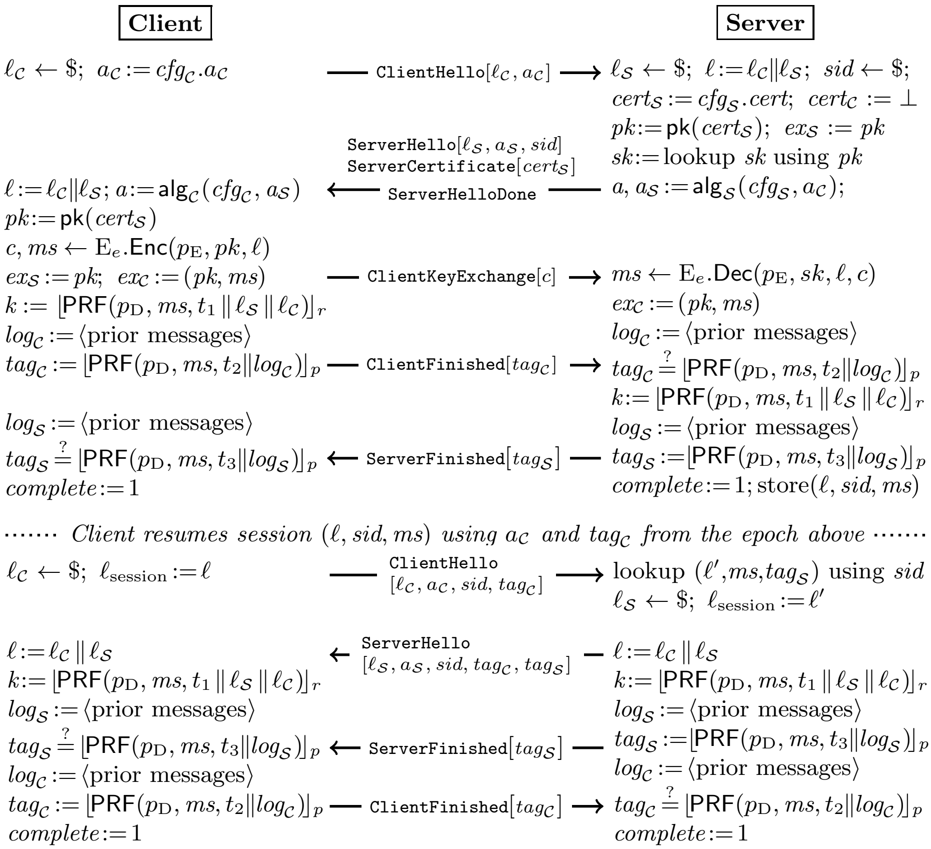
Our definition also provides (some) security for anonymous connections, which can be composed with other authentication mechanisms to achieve application security. For example, renegotiation with client and server certificates may provide mutual authentication on top of an initial, safe, but anonymous handshake. Late application-level, client password authentication may also yield mutual authentication, as illustrated by miTLS [4].

5 Proving Agile Security for TLS Handshakes

We are now ready to reduce the security of TLS handshakes to the security of agile signatures, KEMs and PRFs. We structure the proof to apply simultaneously to the protocol, illustrated in Figures 1 and 2, and to its miTLS implementation.

Figure 1 shows the assignments performed by a client instance and a server instance that run two successive, matching handshakes on the same connection: for both instances, a static session, followed by a (renegotiated) resumption. Figure 2 similarly shows the assignments for an ephemeral session. The agility parameter a of the handshake indicates which algorithm to use for each underlying functionality. We write for instance $a := \mathsf{alg}_{\mathcal{C}}(cfg_{\mathcal{C}}, a_{\mathcal{S}})$ to retrieve a from the client configuration and the negotiation parameter of the server; $e, p := \mathsf{kem}(a)$ to retrieve the core algorithm e and public parameter of the master secret KEM from a; and $E_e.\mathsf{Enc}$ for encryption using the master-secret KEM for e.

Our second main theorem reduces the security of TLS handshakes to their underlying algorithms, depending on a *strength predicate* on their agility parameters.



Client		**Server**
$\ell_C \leftarrow \$;\ a_C := cfg_C.a_C$	— ClientHello[ℓ_C, a_C] ⟶	$\ell_S \leftarrow \$;\ \ell := \ell_C \| \ell_S;\ sid \leftarrow \$;$
		$certs_S := cfg_S.cert;\ cert_C := \bot$
		$pk := \mathsf{pk}(certs_S);\ ex_S := pk$
	ServerHello[ℓ_S, a_S, sid]	$sk :=$ lookup sk using pk
	ServerCertificate[$certs_S$]	
$\ell := \ell_C \| \ell_S;\ a := \mathsf{alg}_C(cfg_C, a_S)$	⟵ ServerHelloDone —	$a, a_S := \mathsf{alg}_S(cfg_S, a_C);$
$pk := \mathsf{pk}(certs_S)$		
$c, ms \leftarrow \mathrm{E}_e.\mathsf{Enc}(p_\mathrm{E}, pk, \ell)$		
$ex_S := pk;\ \ ex_C := (pk, ms)$	— ClientKeyExchange[c] ⟶	$ms \leftarrow \mathrm{E}_e.\mathsf{Dec}(p_\mathrm{E}, sk, \ell, c)$
$k := \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_1 \| \ell_S \| \ell_C) \rfloor_r$		$ex_C := (pk, ms)$
$log_C := \langle$prior messages$\rangle$		$log_C := \langle$prior messages$\rangle$
$tag_C := \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_2 \| log_C) \rfloor_p$	— ClientFinished[tag_C] ⟶	$tag_C \overset{?}{=} \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_2 \| log_C) \rfloor_p$
		$k := \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_1 \| \ell_S \| \ell_C) \rfloor_r$
$log_S := \langle$prior messages$\rangle$		$log_S := \langle$prior messages$\rangle$
$tag_S \overset{?}{=} \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_3 \| log_S) \rfloor_p$	⟵ ServerFinished[tag_S] —	$tag_S := \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_3 \| log_S) \rfloor_p$
$complete := 1$		$complete := 1;$ store(ℓ, sid, ms)
······ *Client resumes session* (ℓ, sid, ms) *using* a_C *and* tag_C *from the epoch above* ······		
$\ell_C \leftarrow \$;\ \ell_\mathrm{session} := \ell$	— ClientHello [ℓ_C, a_C, sid, tag_C] ⟶	lookup (ℓ',ms,tag_S) using sid
		$\ell_S \leftarrow \$;\ \ell_\mathrm{session} := \ell'$
$\ell := \ell_C \| \ell_S$	⟵ ServerHello [$\ell_S, a_S, sid, tag_C, tag_S$] —	$\ell := \ell_C \| \ell_S$
$k := \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_1 \| \ell_S \| \ell_C) \rfloor_r$		$k := \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_1 \| \ell_S \| \ell_C) \rfloor_r$
$log_S := \langle$prior messages$\rangle$		$log_S := \langle$prior messages$\rangle$
$tag_S \overset{?}{=} \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_3 \| log_S) \rfloor_p$	⟵ ServerFinished[tag_S] —	$tag_S := \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_3 \| log_S) \rfloor_p$
$log_C := \langle$prior messages$\rangle$		$log_C := \langle$prior messages$\rangle$
$tag_C := \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_2 \| log_C) \rfloor_p$	— ClientFinished[tag_C] ⟶	$tag_C \overset{?}{=} \lfloor \mathsf{PRF}(p_\mathrm{D}, ms, t_2 \| log_C) \rfloor_p$
$complete := 1$		$complete := 1$

Two epochs on the same connection: the first handshake establishes a session without client authentication using static keys; the second one resumes the session.
Conventions in the figure:
(1) We use $\overset{?}{=}$ for checks; a failed check stops the instance.
(2) We use := for assigning epoch variables; variables exchanged in messages are implicitly assigned, e.g. the server assigns ℓ_C and a_C after parsing the first message.
(3) We omit the extraction of the negotiated key exchange algorithm e and the parameters $p_\mathrm{E}, p_\mathrm{D}$ from a; for instance, we write p_D for $\mathsf{prf}(a)$.
(4) We omit ChangeCipherSpec messages: they are not part of the handshake protocol.
(5) We write $\langle$prior messages$\rangle$ for the concatenation of all messages sent and received so far in the epoch, starting from the latest ClientHello. (6) We let $\lfloor . \rfloor_r$ and $\lfloor . \rfloor_p$ be functions that truncate to record-key and MAC sizes.
(7) We let t_1, t_2, t_3 abbreviate the constant strings "derive key", "client finished", "server finished"; we write $\|$ for bytestring concatenation.

Fig. 1. Abstract model of TLS handshake protocol (static handshake; resumption)

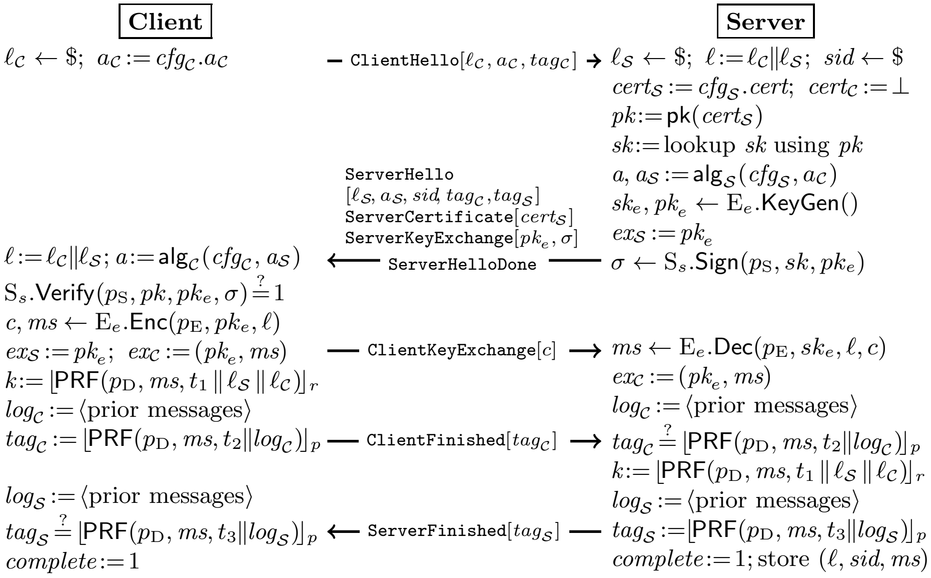

Client
Server
$\ell_{\mathcal{C}} \leftarrow \$;\ a_{\mathcal{C}} := cfg_{\mathcal{C}}.a_{\mathcal{C}}$
ClientHello$[\ell_{\mathcal{C}}, a_{\mathcal{C}}, tag_{\mathcal{C}}]$
$\ell_{\mathcal{S}} \leftarrow \$;\ \ell := \ell_{\mathcal{C}} \| \ell_{\mathcal{S}};\ sid \leftarrow \$$
$certs_{\mathcal{S}} := cfg_{\mathcal{S}}.cert;\ cert_{\mathcal{C}} := \bot$
$pk := \mathsf{pk}(certs_{\mathcal{S}})$
$sk :=$ lookup sk using pk
$a, a_{\mathcal{S}} := \mathsf{alg}_{\mathcal{S}}(cfg_{\mathcal{S}}, a_{\mathcal{C}})$
$sk_e, pk_e \leftarrow \mathrm{E}_e.\mathsf{KeyGen}()$
$exs_{\mathcal{S}} := pk_e$
$\sigma \leftarrow \mathrm{S}_s.\mathsf{Sign}(p_{\mathrm{S}}, sk, pk_e)$
ServerHello
$[\ell_{\mathcal{S}}, a_{\mathcal{S}}, sid, tag_{\mathcal{C}}, tag_{\mathcal{S}}]$
ServerCertificate$[certs_{\mathcal{S}}]$
ServerKeyExchange$[pk_e, \sigma]$
ServerHelloDone
$\ell := \ell_{\mathcal{C}} \| \ell_{\mathcal{S}};\ a := \mathsf{alg}_{\mathcal{C}}(cfg_{\mathcal{C}}, a_{\mathcal{S}})$
$\mathrm{S}_s.\mathsf{Verify}(p_{\mathrm{S}}, pk, pk_e, \sigma) \overset{?}{=} 1$
$c, ms \leftarrow \mathrm{E}_e.\mathsf{Enc}(p_{\mathrm{E}}, pk_e, \ell)$
$exs_{\mathcal{S}} := pk_e;\ exc_{\mathcal{C}} := (pk_e, ms)$
$k := \lfloor \mathsf{PRF}(p_{\mathrm{D}}, ms, t_1 \| \ell_{\mathcal{S}} \| \ell_{\mathcal{C}}) \rfloor_r$
$log_{\mathcal{C}} := \langle$prior messages$\rangle$
$tag_{\mathcal{C}} := \lfloor \mathsf{PRF}(p_{\mathrm{D}}, ms, t_2 \| log_{\mathcal{C}}) \rfloor_p$
ClientKeyExchange$[c]$
$ms \leftarrow \mathrm{E}_e.\mathsf{Dec}(p_{\mathrm{E}}, sk_e, \ell, c)$
$exc_{\mathcal{C}} := (pk_e, ms)$
$log_{\mathcal{C}} := \langle$prior messages$\rangle$
ClientFinished$[tag_{\mathcal{C}}]$
$tag_{\mathcal{C}} \overset{?}{=} \lfloor \mathsf{PRF}(p_{\mathrm{D}}, ms, t_2 \| log_{\mathcal{C}}) \rfloor_p$
$k := \lfloor \mathsf{PRF}(p_{\mathrm{D}}, ms, t_1 \| \ell_{\mathcal{S}} \| \ell_{\mathcal{C}}) \rfloor_r$
$log_{\mathcal{S}} := \langle$prior messages$\rangle$
$log_{\mathcal{S}} := \langle$prior messages$\rangle$
$tag_{\mathcal{S}} \overset{?}{=} \lfloor \mathsf{PRF}(p_{\mathrm{D}}, ms, t_3 \| log_{\mathcal{S}}) \rfloor_p$
ServerFinished$[tag_{\mathcal{S}}]$
$tag_{\mathcal{S}} := \lfloor \mathsf{PRF}(p_{\mathrm{D}}, ms, t_3 \| log_{\mathcal{S}}) \rfloor_p$
$complete := 1$
$complete := 1;$ store (ℓ, sid, ms)

Fig. 2. Abstract model of TLS handshake protocol (ephemeral renegotiation)

Its proof (in the full paper) relies on intermediate definitions for multi-key libraries and, as a first step, uses hybrid arguments to lift security from our agile definitions to the multi-key setting.

Theorem 2 (TLS Handshake). *Let a, $a^\star$ range over the agility parameters supported by TLS. Let $P_s = \{p^\star \mid s, p^\star := \mathsf{sig}(a^\star)\}$, $P_e = \{p^\star \mid e, p^\star := \mathsf{kem}(a^\star)\}$, and $P = \{p^\star \mid p^\star := \mathsf{prf}(a^\star)\}$. Let α be a strength predicate (Definition 4) such that the following assumptions hold:*

(1) *If $\alpha(a)$ and $s, p := \mathsf{sig}(a)$ then S_s is EUF-CMA $(\epsilon_{s,p}, t_{s,p}, p, P_s)$-secure.*
(2) *If $\alpha(a)$ and $e, p := \mathsf{kem}(a)$ then E_e is IND-RCCA-n_{ms} $(\epsilon_{e,p}, t_{e,p}, p, P_e)$-secure.*
(3) *If $\alpha(a)$ and $p := \mathsf{prf}(a)$ then PRF is an (ϵ_p, t_p, p, P)-secure PRF.*

Let n_s bound the number of calls to $S_s.\mathsf{KeyGen}$. Let n and n_{ms} bound the number of epochs and sessions. Let n_e bound the number of calls to $E_e.\mathsf{KeyGen}$, both for ephemeral and static KEMs. The TLS handshake is (ϵ, t, α)-secure, where

$$\epsilon = \sum_s \sum_p n_s \epsilon_{s,p} + \sum_e \sum_p n_e \epsilon_{e,p} + n_{ms} \sum_p \epsilon_p + n^2 (2^{-225} + 2^{-\mathsf{min}_p |\lfloor \cdot \rfloor_p|})$$

and where each t_* in the assumptions is at most t plus the cost of simulating Π in the reduction.

Discussion. In the theorem, the sets P_s, P_e, and P represent the worst case. Indeed, signers may, for those keys that they consider honest, stop using signature algorithm s together with weak hash functions, like MD5, while TLS may still support verification using such hash algorithms for backward compatibility. To model such scenarios, one could instead add P_s, P_e, and P to the state of the experiment to record which hash algorithms have been used so far for signing, decrypting and deriving keys to obtain a more precise statement.

6 Verified Reference Implementation

We jointly programmed the TLS handshake and developed its proof. We finally outline our code, and explain how its structure and automated verification relate to the cryptographic models of §2–5; we provide additional details and performance results in the full paper. Our handshake implementation for miTLS consists of 3,600 lines of F# code plus 2,050 lines of F7 specifications; it supports four protocol versions, three key exchange mechanisms, two signature algorithms, and four hash functions. It deals mostly with the protocol aspects; indeed, our cryptographic proof for Theorem 1, conducted with EasyCrypt, concerns less than 200 lines of F#. Conversely, Theorem 2 involves the full codebase and proving it requires a modular design and automated program verification techniques.

We adopt the type-based cryptographic verification method of Fournet et al. [10], previously applied to miTLS by Bhargavan et al. [4, §2]. The miTLS library consists of 45 modules, not counting application code or platform libraries. Each module implements a single cryptographic functionality or protocol component and represents an abstraction boundary through its interface. A module is either trusted to be implemented correctly (e.g. the session database), or idealized under a cryptographic assumption (e.g. signatures) then verified, or perfectly verified (e.g. the protocol state machine). Each module interface specifies preconditions, postconditions, and type abstractions that govern the conditions under which secrets (keys, plaintexts, etc.) may be read or written by other modules.

We discuss the design of three important components that we modified during the course of this paper. *TLSInfo* defines agility parameters and logical predicates (corresponding to α in Definition 4) that specify algorithmic strength, honesty for both long-term-keys and ephemeral secrets, matching record algorithms, and handshake completion events. This new logical model is more detailed than the original one [4]; furthermore, we extended the session structure and logical model to provide a general treatment of protocol extensions. *HandshakeMessages* implements message formatting and parsing; agreement (Definition 5(4)) depends on its details, since only formatted data is cryptographically authenticated. This code is complicated but not especially deep, and best handled using automated verification. *Handshake* implements the handshake state machine (*Send* in §5). Its code is not as simple as suggested by the KEMs of §3, since the TLS standard employs different sequences of messages for (say) RSA and DHE handshakes. Hence, we have similar but separate code for them, each of their interfaces complying with the KEM abstraction of §3. Also, our code handles errors and warnings, omitted in this presentation but also verified.

Our new results on the handshake, composed with prior results on miTLS [4] (the record layer, the top-level API, and various applications) yield agile, verified application security for TLS as it is.

References

1. Acar, T., Belenkiy, M., Bellare, M., Cash, D.: Cryptographic agility and its reation to circular encryption. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 403–422. Springer, Heidelberg (2010)
2. Barthe, G., Grégoire, B., Heraud, S., Zanella-Béguelin, S.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
3. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 232–249. Springer, Heidelberg (1994)
4. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y.: Implementing TLS with verified cryptograhic security. In: IEEE Symposium on Security and Privacy (2013)
5. Bhargavan, K., Delignat-Lavaut, A., Fournet, C., Pironti, A., Strub, P.-Y.: Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In: IEEE Symposium on Security and Privacy (2014)
6. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998)
7. Canetti, R., Krawczyk, H., Nielsen, J.B.: Relaxing chosen-ciphertext security. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 565–582. Springer, Heidelberg (2003)
8. Cramer, R., Shoup, V.: Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. SIAM J. Computing 33(1), 167–226 (2003)
9. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2 (2008)
10. Fournet, C., Kohlweiss, M., Strub, P.-Y.: Modular code-based cryptographic verification. In: ACM CCS 2011 (2011)
11. Giesen, F., Kohlar, F., Stebila, D.: On the security of TLS renegotiation. In: ACM CCS 2013 (2013)
12. Haber, S., Pinkas, B.: Securely combining public-key cryptosystems. In: ACM CCS 2001 (2001)
13. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 273–293. Springer, Heidelberg (2012)
14. Jonsson, J., Kaliski Jr., B.S.: On the security of RSA encryption in TLS. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 127–142. Springer, Heidelberg (2002)
15. Klíma, V., Pokorný, O., Rosa, T.: Attacking RSA-based sessions in SSL/TLS. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 426–440. Springer, Heidelberg (2003)
16. Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DH and TLS-RSA in the standard model. Cryptology ePrint Archive, Report 2013/367 (2013)

17. Krawczyk, H., Paterson, K.G., Wee, H.: On the security of the TLS protocol: A systematic analysis. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 429–448. Springer, Heidelberg (2013)
18. Langley, A.: Unfortunate current practices for HTTP over TLS (2011), http://www.ietf.org/mail-archive/web/tls/current/msg07281.html
19. Modadugu, N., Langley, A., Moeller, B.: Transport Layer Security (TLS) False Start. Internet Draft (2010)
20. Mavrogiannopoulos, N., Vercauteren, F., Velichkov, V., Preneel, B.: A cross-protocol attack on the TLS protocol. In: ACM CCS 2012 (2012)
21. Morrissey, P., Smart, N.P., Warinschi, B.: A modular security analysis of the TLS handshake protocol. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 55–73. Springer, Heidelberg (2008)
22. Naccache, D., Shparlinski, I.E.: Divisibility, Smoothness and Cryptographic Applications. ArXiv e-prints (October 2008)
23. Paterson, K.G., Ristenpart, T., Shrimpton, T.: Tag size does matter: Attacks and proofs for the TLS record protocol. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 372–389. Springer, Heidelberg (2011)
24. Qualys SSL labs. SSL server test, https://www.ssllabs.com/ssltest/analyze.html
25. Ray, M.: Authentication gap in TLS renegotiation (2009), http://extendedsubset.com/Renegotiating_TLS.pdf
26. Stevens, M., Sotirov, A., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D.A., de Weger, B.: Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. Cryptology ePrint Archive, Report 2009/111 (2009)
27. Wagner, D., Schneier, B.: Analysis of the SSL 3.0 protocol. In: 2nd USENIX Workshop on Electronic Commerce, WOEC 1996 (1996)