Proving the TLS Handshake Secure (as it is)

Karthikeyan Bhargavan [*] Cédric Fournet [†] Markulf Kohlweiss [‡]
Alfredo Pironti [§] Pierre-Yves Strub [¶] Santiago Zanella-Béguelin [‖]

March 6, 2014

Abstract

The TLS protocol 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. We experimentally confirm that all mainstream implementations of TLS share key materials between many different algorithms, some of them of dubious strength. We outline new attacks we found in their handling of session resumption and renegotiation, stressing the need to model multiple related instances of the handshake.

We systematically 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. miTLS inter-operates with mainstream browsers and servers for many protocol versions, configurations, and ciphersuites; and it provides application-level, provable security for some.

We propose new agile security definitions and assumptions for the signatures, key encapsulation mechanisms, and key derivation algorithms used by the TLS handshake. By necessity, our definitions are stronger than those expected with simple modern protocols.

To validate our model of key encapsulation, we prove that RSA ciphersuites satisfy the security assumption needed for our proof of the handshake. Specifically, we formalize the use of PKCS#1v1.5 encryption in TLS, including recommended countermeasures against Bleichenbacher attacks, and build a 3,000-line EasyCrypt proof of its security 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.

Keywords: TLS protocol, handshake, key exchange, cryptographic agility, provable security, reference implementation, PKCS, RSA, KEM

[*] INRIA. E-mail: karthikeyan.bhargavan@inria.fr
[†] Microsoft Research. E-mail: fournet@microsoft.com
[‡] Microsoft Research. E-mail: markulf@microsoft.com
[§] INRIA. E-mail: alfredo.pironti@inria.fr
[¶] IMDEA Software Institute. E-mail: pierre-yves@strub.nu
[‖] Microsoft Research. E-mail: santiago@microsoft.com

Contents

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 controversial. One of the most troublesome aspects of the protocol is its handling of a large number of cryptographic algorithms and constructions. This diversity shows no sign of abating, as new extensions are added to the protocol and its implementations, while older features are maintained 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 trade-off 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 [see e.g. 11]. In theory, only very restrictive configurations have been proved secure. In practice, dangerous mis-configurations of TLS and its underlying certificates are commonplace [see e.g. 25, 20].

(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 SSL2's protocol regression [60] to current browsers' version fallback [43].

(3) It demands stronger security assumptions, to reflect the fact that honest parties may use the same keys with different algorithms. Intuitively, TLS *on its own* enables a range of chosen-protocol attacks [34, 31] 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. The situation is aggravated by the common practice of buying a single certificate for multiple purposes.

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 connection 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 connection keys. At that point, the security of those keys depends on algorithms and constructions used in three runs of the handshake. This is in sharp contrast with prior work on the provable security of TLS [30, 37, 39], which focus on a fixed run of the protocol, for a fixed choice of algorithms. (See §7.1 for a detailed discussion of related work on provable security for TLS, and §7.2 for new attacks involving triple handshakes.)

1.1 Cryptographic Agility in TLS

Agile security considers families of schemes or protocols, all serving the same purpose, when the same key materials are shared across members of the family. Acar et al. [2] propose agile definitions for pseudorandom functions and encryption schemes, and advocate agility as a major practical concern for protocols like TLS. Instead, *combined,* or *joint security* [28] studies the sharing of key materials 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. Prior works look at the idiosyncratic use of cryptographic primitives in TLS such as hash functions and randomness extractors [22, 21], but do not consider agile security.

The agility mechanisms of TLS is primarily driven by ciphersuites of the form $\texttt{TLS\_}e\texttt{\_}s\texttt{\_WITH\_}r$. This ciphersuite roughly indicates a key encapsulation mechanism (KEM) e and a 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 pre-master secret encrypted under the server public key, used as the seed of a SHA1-based PRF for deriving 4 keys for SHA1-based MACs and AES encryption in CBC mode. TLS 1.2 currently has 314 registered ciphersuites [29]. More precisely, the choice of algorithms depends on additional data exchanged during the handshake (and subject to active attacks), including protocol versions, certificate requests, certificate chains, Diffie-Hellman group descriptions, and the contents of 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 thousands fixed-algorithm variants of the handshake.

1.2 Empirical Study of Major Web Servers and Browsers

Using an online analyzer [54], we gathered in January 2014 extended information on server configurations for 215 out of the top 500 domains,[1] including the TLS versions, ciphersuites, certificates, and extensions they offer. The full results are reported in §A.

The servers tested accept a total of 64 ciphersuites, with an average of 12 and standard deviation of 6. They accept on average more than 5 encryption algorithms and 2 hash methods. They still widely deploy weak algorithms: 70% accept at least one ciphersuite based on MD5 and 90% at least one based on RC4.

All tested websites but one offer at least two TLS 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 with compatible ciphersuites. They all disable TLS-level compression. 86% support the (mandatory) secure renegotiation extension, leaving the others vulnerable to renegotiation attacks [55]. 60% support session tickets for resumption.

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). On the other hand, clients tend to support more recent ciphersuites than servers, notably those based on elliptic curves.

1.3 Cross-ciphersuite attacks

As a first, well-known example of key reuse, most web servers are still configured to use the same RSA certificate both for signing handshake messages and for decrypting pre-master 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. This practice is discouraged and our presentation could treat those keys as compromised; for the sake of modularity we first cover single-purpose keys and show how to extend our results to dual-purpose keys under stronger assumptions obtained by extending our definitions for signatures (§2) and KEMs (§3), with oracles for decrypting and signing, respectively. Klíma and Rosa [35] developed attacks in this model, and Degabriele et al. [17] recently demonstrated their applicability to the context of the EMV protocol. We further discuss these concerns in §B.2.

As a second example, Mavrogiannopoulos et al. [48] report an interesting cross-protocol attack between plain Diffie-Hellman (DH) and Elliptic-Curve Diffie-Hellman (ECDH) ciphersuites, due to a mis-

[1] `http://www.alexa.com/topsites/global` excluding domains that do not support HTTPS with a valid certificate.

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.

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 to confirm the integrity of the handshake. As an optimization, Langley and Moeller [45] even let 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 in the client and as a MAC key in the server. To our knowledge, we are the first to report this cross-algorithm attack against [45]. 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 we suspect that key recovery attacks against e.g. RC4 in $r_{\mathcal{C}}$ could be used to attack strong $r_{\mathcal{S}}$ schemes.

1.4 Multiple Sessions and Connections

We set up some TLS terminology for multiple related handshakes. 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 Precise, Modular, Code-Based Security for the TLS Handshake

Our main result is provable security for a standard-compliant, reference implementation of the TLS handshake, seen as a very 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 and Diffie-Hellman encryption, using a 3,000-line EasyCrypt [5] proof. Working with a reference implementation, and testing it against others, forces us to handle all the details needed to achieve multi-handshake, multi-algorithm security for TLS as it is used—sometimes resolving ambiguities in the standard. Proving it secure requires both modularity and automation. Conversely, the attacks in §7.2 illustrate the need to jointly model agility, resumption, and renegotiation.

A feature of TLS that traditionally resists abstraction is that the handshake delivers 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 TLS can only be proved secure by including the encrypted finished messages. The kernel of the lore is that it cannot be proved under a Bellare and Rogaway [7] style key-exchange definition. To achieve modularity, we change the definition and separate record-key generation from handshake completion. Our main definition delivers the record keys in the middle of the handshake, before signalling 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 a necessary condition for the secure use of the record keys for encrypting application data—bot not for encrypting handshake finished messages. This resolves the *finished message controversy* of Jager et al. [30] in a novel and surprisingly elegant way.

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 keys, and ensures agreement on parameters for the record layer. In addition, we have integrated our implementation of the handshake into miTLS [9], the first implementation of TLS verified in the computational model of cryptography. miTLS also has verified code for the record layer and the protocol logic, showing the usability of the keys established by our handshake; its security model ensures by typing that the record keys are used for protecting application data only after completion. By composing our results with theirs, we establish security of a whole, standard-compliant reference implementation of the TLS standard, as well as the security of sample web applications built on top of the resulting TLS API.

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 algorithms and epochs in the TLS handshake. §6 briefly describes our use of high-level programming, type systems, and provers to carry out a modular cryptographic proof 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. This more abstract treatment suffices to convey the main ideas, and deliberately omits many important aspects of the handshake, such as its message formats. We refer to the standard [18] or the implementation for the details.

Agile 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, and describe the hash-and-sign construction of TLS, but 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 but separate problem. Our constructions simply authenticate the exchanged certificates chains, and use a specification function to extract from them the public keys used in the handshake.

Master secrets, key encapsulation, and key derivation Similarly to Krawczyk et al. [39], we use key encapsulation mechanisms [16] to model key-exchange; this allows us to unify RSA and Diffie-Hellman ciphersuites within the same formalism. As Morrissey et al. [50], to improve modularity, we decompose the handshake into *pre-master secret*, *master secret*, and *record-key derivation* phases—this decomposition matters e.g. for modelling the re-use of master secrets for resumption.

We show how to securely construct a master-secret KEM from a pre-master-secret KEM in the random oracle model (§3, Theorem 3) and, independently, how to derive record keys and finished-message MACs from master secrets (§B.3). We formalize the proof of Theorem 3 in EasyCrypt. The proof is particularly complex for RSA: it involves showing that PKCS#1v1.5 with countermeasures to Bleichenbacher's [10] padding oracle attack and its follow-ups [36] provides enough protection against chosen-ciphertext attacks.[2] 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 additional require that PKCS#1v1.5 ciphertexts be hard to re-randomize. During the EasyCrypt development, we discovered minor flaws in our first informal proof, as well as in the proof of Krawczyk et al.; the authors acknowledged these flaws, which fortunately do not affect the overall bound, and fixed them in a long version of their paper [40]. Besides, complying with the TLS standard, we support agility in the hash algorithm used to extract a master secret from a pre-master secret. As for agile signatures in §2, we arrive at a security 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. The corresponding definition is given in §B.3; it involves a novel commit oracle to support algorithmic agility in the record-layer algorithm r without having to make strong agile assumptions on record algorithm families, as discussed in §1.3.

Agile security model (§4) and TLS proof (§5) for multiple sequences of handshakes The main two goals of the handshake are to establish shared symmetric keys for the record layer, and to agree on many parameters, notably 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 application, the rest of TLS, and the network) with oracles to create honest connections and long-term keys for clients and servers, to control their usage, and to send network 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 diverse TLS-specific 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 TLS sessions, both parties agree on a unique label (obtained by concatenating client and server random values), the negotiation parameters and key-exchange values of the client and server, an algorithm description (primarily the protocol version and the negotiated ciphersuite) and optional certificate chains for the client and the server. For TLS resumptions, both parties agree on the label of the session being resumed, as well as a fresh unique label (obtained by concatenating new client and server random values) for key derivation.

We also provide secure key derivation, depending on distinguished exchange-value assignments for each ciphersuite. A session is *safe* when honest client and server agree on these assignments (this is similar to matching conversations), under suitable conditions on algorithms and long-term keys. As discussed above, our definition immediately releases all connection keys. For safe sessions, we guarantee that these keys are indistinguishable from fresh random keys. (In TLS, but not within the handshake, these keys will be used e.g. to encrypt the finished messages, but record encryption plays no role in our definition.) Additionally, depending on the signing keys, 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 4) reduces the concrete security of the TLS handshake to agile assumptions on the constructions used for signatures, for KEMs, and for PRFs. Each epoch assigns a

[2] Building on our work, Kohlweiss et al. [38] use the same assumption and general proof idea to extend their work to support TLS-RSA.

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` [30], we may apply our theorem with α set to true only when a selects that ciphersuite. This is already much stronger than prior non-agile results for TLS that assume all honest parties agree *in advance* on a ciphersuite and reject any others.

Figure 1 gives a model of the TLS handshake with enough detail to follow our proof, but it covers only two epochs (a static handshake with an anonymous client and a resumption) and still elides many details and requires familiarity with the TLS standard. We recall, however, that our main result also applies to our standard-compliant implementation of the handshake for miTLS. Remarkably, our model accounts for agility with respect to record algorithms and allows us to prove channel security *without* the need for agile assumptions on the algorithms r used in the record layer. We thus validate the use of stateful LHAE [52] in the TLS standard, even for clients and servers that negotiate r. We require, however, that no application data be sent before the finished messages are verified. Some implementations violate this requirement, e.g. all Google servers and various browsers [45]; stronger agile assumptions are then unavoidable.

Code-Based Verified Implementation (§6) We finally present our reference implementation of the handshake, integrated in miTLS, and explain how we verify it 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. We provide experimental results, showing that our 'executable model' within miTLS runs sample client and server applications with comparable performance. Our code improves on the original one for miTLS [9], which supported less features, and whose security relied on monolithic, TLS-specific assumptions for RSA and DH ciphersuites.

To handle agile security in TLS, and to enable its automated verification using the proof given in this paper, our code is structured into small, independent modules (that is, program libraries), many of them parameterized by algorithm descriptors. Thus, our code, e.g. for the HMAC based PRF of 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 of our constructions for the handshake corresponds to a separate module in the code. As we treat cryptographic constructions as program libraries, we express their security for multiple keys and multiple algorithms. (§B.4 describes them, and provides the corresponding hybrid arguments.) To further align the code and the cryptographic proof, we express their security as indistinguishability between a concrete and an idealized variant of the code, under usage restrictions formally captured (and enforced) using a precise type system, as described by Fournet et al. [23] and Bhargavan et al. [9].

Our work sheds light on important design and implementation issues of TLS as it is used today; it also suggests simple improvements to strengthen its security. To our knowledge, ours are the first provable-security results for TLS that account for algorithm agility. We are also the first to fully model the security of interdependent handshakes related by (session) resumption and (connection) renegotiation.

Further reading Appendixes provide the raw data for our empirical analysis and additional discussions, definitions, constructions, and proofs. An attack paper and video describing triple-handshake attacks over

TLS, can be found at https://www.secure-resumption.com/. Further material is available at the miTLS webpage at http://www.mitls.org/.

1.7 Notation

We use sans-serif font for algorithm names, e.g. Alg. If such an algorithm uses a more primitive algorithm, we denote it by alg. In security experiments, we denote ALG the oracle giving access to algorithm Alg.

We use $:=$ for deterministic assignments, and $\leftarrow$ to denote a random assignment either uniformly from a finite set or according to a distribution determined by a probabilistic algorithm. When describing generic key exchange or key derivation primitives we use \$ to denote the key space.

We use identifiers of cryptographic primitives, like h for a hash algorithm, s for a signature scheme, or e for a KEM, as both the name of the scheme and the scheme itself when there is no confusion. We denote signature and KEM schemes constructed from, and thus parameterized by these schemes, by S_s and E_e respectively.

We sometimes abuse notation and write, e.g., (a, b, c) instead of $((a, b), c)$ to improve readability.

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 that determines their hash algorithms. For instance, given a core signature scheme $s = (\mathsf{keygen}, \mathsf{sign}, \mathsf{verify})$, the hash-then-sign scheme $S_s = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ for TLS is defined as follows (we use h both as the name of the hash algorithm and the algorithm itself since there is no confusion): $\mathsf{KeyGen} \triangleq \mathsf{keygen}$ generates a key pair for algorithm s; $\mathsf{Sign}(h, sk, m) \triangleq \mathsf{sign}(sk, h(m))$ computes a signature using the base signature scheme 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 $\mathsf{EUF} \triangleq$	Oracle $\mathsf{SIGN}(p, m) \triangleq$
$pk, sk \leftarrow \mathsf{KeyGen}();\ M := \varnothing$	**if** $p \notin P$ **then return** $\perp$
$m', \sigma \leftarrow \mathcal{A}^{\mathsf{SIGN}}(pk)$	$M := M \cup \{m\}$
return $m' \notin M \wedge \mathsf{Verify}(p^\star, pk, m', \sigma)$	**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, which coincides with agile EUF-CMA security 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. [58] rule out only (MD5, {MD5, ...})-security, while (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 [51] are likely to break (SHA256, {MD5, SHA256})-security, even though (SHA256, {SHA256})-security would still hold.

The TLS standard features the following 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 introduced additional agility to facilitate migration from MD5 and SHA1 to stronger hash functions. The designers seem to be aware of potential agility problems, and prescribe ad hoc countermeasures [18, §7.4.3]. The standard still restricts (EC)DSA to use SHA1, delaying the migration to stronger algorithms. On the other hand, it adds an encoding of the hash algorithm identifier as defined in [32] for RSA to guarantee that all hash algorithms have disjoint range.

If the base signature scheme itself were (ϵ, t)-EUF-CMA secure on the range of h and h'; then we would 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 base 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 9. (As evidenced by Bleichenbacher at the Crypto'06 rump session and elaborated by Kühn et al. [41], implementations need to be careful.)

3 Master Secrets & Key Encapsulation

As explained in §1.5, we handle KEMs in two steps. We first model RSA and Diffie-Hellman pre-master secret phase as agile KEMs, written $(\mathsf{keygen}, \mathsf{enc}, \mathsf{dec})$ and parameterized by a 2-byte protocol version string. (Thankfully, TLS never mixes secret long-term key materials between RSA and Diffie-Hellman.)

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 pre-master secret pms, and returns the ciphertext c resulting from encrypting it under pk using PKCS#1v1.5; $\mathsf{dec}(pv, sk, c)$ decrypts c with sk using PKCS#1v1.5. If the padding is correct and the result pms is exactly 48 bytes long, it returns pms with the first 2 bytes replaced by pv, otherwise returns $\perp$. The latter case is handled generically in the construction of the master secret KEM given below.

Diffie-Hellman: keygen selects group parameters pp, generates a fresh pair of DH values $(g^x,\ x)$, and returns $pk = (pp, g^x)$ as the public and $sk = (pk, x)$ as the 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 $c^x = g^{xy}$. In contrast to the RSA pms-KEM, neither enc or dec depend on pv.

On their own, these two pre-master secret KEMs are *not* secure under any indistinguishability notion, even under relatively weak active attacks, e.g. 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. This *encapsulate-then-hash* approach for KEMs in analogous to the *hash-then-sign* approach for signatures.

Generic *ms*-KEM construction We model the master secret KEM of TLS as an *agile labeled KEM* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ whose agility parameters are pairs composed of a valid protocol version and a hash algorithm name, and where labels are the concatenation of the client and server nonces. 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 pre-master secret pms and a ciphertext c using the enc algorithm of e, then derives a master secret ms from ℓ using the agile 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. It returns the value obtained from pms and ℓ using the agile KEF.

We define security for agile labeled KEMs as indistinguishability under replayable chosen-ciphertext attacks (IND-RCCA). This is a relaxation of CCA security, introduced for public-key encryption by Canetti et al. [15], that suffices for our proof of the handshake.

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 $\triangleq$	Oracle ENC(ℓ) $\triangleq$	Oracle DEC(p,ℓ,c) $\triangleq$
$pk, sk \leftarrow \mathsf{KeyGen}()$	**if** $\ell \in L$ **then return** $\bot$	**if** $\ell \in L \lor p \notin P$ **then return** $\bot$
$K, L := \varnothing$	$k_0, c \leftarrow \mathsf{Enc}(p^\star, pk, \ell)$	$L := L \cup \{\ell\}$
$b \leftarrow \{0,1\}$	$k_1 \leftarrow \$$	$k \leftarrow \mathsf{Dec}(p, sk, \ell, c)$
$b' \leftarrow \mathcal{A}^{\mathsf{ENC},\mathsf{DEC}}(pk)$	$K(\ell) := K(\ell) \cup \{k_0, k_1\}$	**if** $k \in K(\ell)$ **then return** $\bot$
return $(b' = b)$	**return** k_b, c	**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 TLS invariant: 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. [39] in the definition of IND-CCCA security for non-agile labeled KEMs. In §3.3 we remove this usage restriction, and replace it with the requirement that the adversary (the reduction in the proof) calls a commit oracle before calling the DEC oracle. This is natural for TLS, where the server is committed to a label when it generates its nonce.

The lemma below enables us to prove IND-RCCA security for a single query, and to 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*.*

Proof. Let $\mathcal{A}$ be an adversary against IND-RCCA-n and consider the hybrid game RCCA_i run with $\mathcal{A}$ whose encryption oracle returns k_1 (a random key) for the first i queries and k_0 (a real key) for the rest. The RCCA advantage of $\mathcal{A}$ can be written as

$$\mathbf{Adv}^{\mathsf{RCCA}}_{p^\star,P}(\mathcal{A}) = \Pr[\mathsf{RCCA}_0 : b' = 1] - \Pr[\mathsf{RCCA}_n : b' = 1]$$

If $\mathcal{A}$ can distinguish between RCCA_0 and RCCA_n with advantage ϵ, then using $\mathcal{A}$ one can construct an adversary $\mathcal{B}$ that queries ENC only once and has advantage ϵ/n. Adversary $\mathcal{B}$ chooses uniformly an index $i \in \{1,\ldots,n\}$, answers to $\mathcal{A}$'s first $i-1$ queries with a random key and a ciphertext computed using the Enc algorithm, to the i-th query using its own ENC oracle, and to the rest with real keys as the game

RCCA would do if $b = 0$. $\mathcal{B}$ answers decryption queries forwarding them to its own DEC oracle, returning $\perp$ if the answer is a key computed during the simulation of an encryption query with the same label, and eventually returns the same response as $\mathcal{A}$. When $b = 0$, for a chosen i the output of the RCCA game for $\mathcal{B}$ is the same as the output of RCCA_{i-1}, and when $b = 1$ it is the same as the output of RCCA_i. We write RCCA($\mathcal{B}$) to denote the RCCA game for $\mathcal{B}$. Summing over all i,

$$\begin{aligned}\mathbf{Adv}^{\mathsf{RCCA}}_{p^\star,P}(\mathcal{B}) &= \Pr[\mathsf{RCCA}(\mathcal{B}) : b' = 1 \mid b = 0] - \Pr[\mathsf{RCCA}(\mathcal{B}) : b' = 1 \mid b = 1] \\ &= \frac{1}{n} \sum_{i=1}^{n} \Pr[\mathsf{RCCA}_{i-1} : b' = 1] - \Pr[\mathsf{RCCA}_i : b' = 1] \\ &= \frac{1}{n} \left(\Pr[\mathsf{RCCA}_0 : b' = 1] - \Pr[\mathsf{RCCA}_n : b' = 1]\right) = \frac{1}{n} \mathbf{Adv}^{\mathsf{RCCA}}_{p^\star,P}(\mathcal{A})\end{aligned}$$

The running time of $\mathcal{B}$ is simply that of $\mathcal{A}$ plus the cost of choosing the index i and simulating the encryption oracle of $\mathcal{A}$, which is essentially $n \cdot t_{\mathsf{Enc}}$. □

We define the assumptions used by our main theorem on KEMs: *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 following games:*

Game OW-PCA $\triangleq$	Game NR-PCA $\triangleq$	Oracle PCO$(p,k,c) \triangleq$
$pk, sk \leftarrow \mathsf{keygen}()$	$pk, sk \leftarrow \mathsf{keygen}()$	**if** $p \notin P \vee k = \perp$ **then return** $\perp$
$k^\star, c^\star \leftarrow \mathsf{enc}(p^\star, pk)$	$k^\star, c^\star \leftarrow \mathsf{enc}(p^\star, pk)$	$k' \leftarrow \mathsf{Dec}(p, sk, c)$
$k \leftarrow \mathcal{A}^{\mathsf{PCO}}(pk, c^\star)$	$c \leftarrow \mathcal{A}^{\mathsf{PCO}}(pk, c^\star)$	**return** $(k' = k)$
return $(k = k^\star)$	**return** $c \neq c^\star \wedge k^\star = \mathsf{dec}(p^\star, sk, c)$	

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.*

3.1 Security of TLS pre-master secret KEMs

We give some preliminary theorems and conjectures about the NR-PCA and OW-PCA security of TLS *pms*-KEMs, and relate our agile IND-RCCA KEMs to prior work and more standard assumptions. We hope that this will stimulate further cryptanalytic work on TLS.

Conjecture 1 (Non-randomizability of RSA *pms*-KEM)**.** *Due to the random self-reducibility of RSA encryption, we conjecture that re-randomizing an RSA pms-KEM ciphertext is as hard as solving the RSA problem (with a considerable reduction loss). In fact, NR-PCA follows from OW-PCA and the* common-input extractability *assumption of [6] (swapping the role of randomness and plaintexts). This latter assumption holds unconditionally for small exponent RSA and certain parameters—not those of TLS—of the PKCS#1v1.5 encoding.*

Note that the DH *pms*-KEM is trivially non-randomizable, as it has unique ciphertexts, and that security against NR-PCA implies security against OW-PCA as long as it is easy to find more than one ciphertext of a given plaintext.

Conjecture 2 (OW-PCA security of RSA *pms*-KEM)**.** *[33] gives us reason to believe that the RSA pms-KEM is (ϵ, t)-OW-PCA secure under the (ϵ', t')-partial-RSA decision oracle assisted RSA assumption where ϵ', t' are, however, not tight.*

Theorem 1 (OW-PCA security of DH *pms*-KEM)**.** *The DH pms-KEM is (ϵ, t)-OW-PCA secure under the (ϵ, t')-Strong Diffie-Hellman assumption [1], where t' is essentially t. This is the assumption that it is hard to compute g^{xy} given g^x, g^y and access to a DDH oracle with the first argument fixed to g^x.*

Proof. The reduction $\mathcal{B}$ receives pp, g^x, g^y as input and has access to a restricted DDH oracle $\mathsf{DDH}(g^x, \cdot, \cdot)$. $\mathcal{B}$ calls the OW-PCA adversary with parameters (pp, g^x) as pk and g^y as c, and answers a plaintext-checking query $\mathsf{PCO}(pv, pms, c)$ using $\mathsf{DDH}(g^x, c, pms)$. $\mathcal{B}$ returns to its challenger the key output by the OW-PCA adversary. If the OW-PCA adversary succeeds, then this key equals g^{xy} and $\mathcal{B}$ wins its game. □

Theorem 2 (Security under PRF-ODH)**.** *The ms-KEM $E_{DH} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is $(\epsilon, t, p, \{p\})$-IND-RCCA under the (ϵ, t)-PRF-ODH assumption for the group parameters pp generated by* KeyGen *and the pseudo-random function $f_{g^{uv}}(\cdot)$ defined as $\mathsf{KEF}(p, g^{uv}, \cdot)$.*

In fact under the PRF-ODH formulation of Krawczyk et al. [39], E_{DH} is $(\epsilon, t, h, \{h\})$-IND-CCA secure, even if TLS would allow the reuse of nonces.

3.2 Security of TLS master secret KEM

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 has been formalized using EasyCrypt. The proof is in the random oracle model for the agile KEF. We discuss later how this assumption can be relaxed. As weaker hash algorithms like MD5 are still widely supported by TLS, a proof in the random oracle is particularly problematic for TLS as it is used today. We investigate ways to avoid the random oracle assumption for all hash algorithms except the one being attacked in §B.1, but it is instructive to consider the setting where all KEF functions are modeled as random oracles first.

We prove security in the single-challenge case and rely on Theorem 1 to extend it to the multi-challenge setting.

Theorem 3 (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 oracle and decryption oracle. 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*

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

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 Diffie-Hellman KEMs, decryption never fails (as the public-key and ciphertext validation is done beforehand) so the last term in the bound above can be omitted.

Proof. In the single-challenge setting, we can represent the adversary $\mathcal{A}$ as a pair of procedures $(\mathcal{A}_1, \mathcal{A}_2)$ sharing state, the procedure $\mathcal{A}_1$ chooses the label for the single query to the encryption oracle, while $\mathcal{A}_2$

tries to guess the challenge bit b. The initial game in the ROM is thus:

Game $\mathsf{RCCA} \triangleq$
$pk, sk \leftarrow \mathsf{KeyGen}()$
$Q, K, L := \varnothing;\ b \leftarrow \{0,1\}$
$\ell^\star \leftarrow \mathcal{A}_1^{\mathsf{KEF},\mathsf{DEC}}(pk)$
if $\ell^\star \in L$ **then return** false
$ms_0, c^\star \leftarrow \mathsf{Enc}(p^\star, pk, \ell^\star)$
$ms_1 \leftarrow \$;\ K(\ell^\star) := \{ms_0, ms_1\}$
$b' \leftarrow \mathcal{A}_2^{\mathsf{KEF},\mathsf{DEC}}(ms_b, c^\star)$
return $(b' = b)$

Oracle $\mathsf{KEF}(p, pms, \ell) \triangleq$
if $(p, pms, \ell) \notin \mathsf{dom}(Q)$ **then**
  $Q(p, pms, \ell) \leftarrow \$$
return $Q(p, pms, \ell)$

Oracle $\mathsf{DEC}(p, \ell, c) \triangleq$
if $\ell \in L \vee p \notin P$ **then return** $\perp$
$L := L \cup \{\ell\}$
$ms \leftarrow \mathsf{Dec}(p, sk, \ell, c)$
if $ms \in K(\ell)$ **then return** $\perp$
return ms

The proof proceeds by a sequence of games; we describe them below.

- RCCA_0. We inline the definition of Dec in the initial game, and move the call to the enc algorithm of the pms-KEM used to compute $pms^\star$ before the first call to the adversary. This game is perfectly equivalent to the initial game because the label chosen by $\mathcal{A}_1$ is not needed to compute $pms^\star$.

- RCCA_1. At the beginning of the game, for each pair (pv, ℓ), sample a random string $F(pv, \ell)$. When decryption of the pms fails during a decryption query, use $pv \,\|\, F(pv, \ell)$ in place of $pv \,\|\, \$$ to compute the master secret. Since each label ℓ appears at most once in a decryption query, each of the used values is random and independent as in RCCA_0 and the two games are equivalent.

- RCCA_2. When decryption of the pms fails during a decryption query, simply use a random ms rather than $\mathsf{KEF}((pv, h), pv \,\|\, F(pv, \ell), \ell)$. This only makes a difference if the adversary makes this query directly and hence

$$\Pr[\mathsf{RCCA}_1 : b = b'] \leq \Pr[\mathsf{RCCA}_2 : b = b'] + \Pr[\mathsf{RCCA}_2 : \exists pv\ h\ \ell, ((pv, h), pv \,\|\, F(pv, \ell), \ell) \in \mathsf{dom}(Q)]$$

  Moreover, since $\mathsf{dom}(Q)$ contains at most $q_{\mathsf{KEF}} + q_{\mathsf{DEC}}$ values, and each one determines a unique pair (pv, ℓ), the latter probability is at most $2^{|pv|-|pms|}(q_{\mathsf{KEF}} + q_{\mathsf{DEC}})$. Note that in game RCCA_2 the values $F(pv, \ell)$ are independent of $\mathsf{dom}(Q)$ because they are never used to answer decryption queries.

- RCCA_3. Same as RCCA_2, but using a random ms_0. The game aborts when either $\mathcal{A}_1$ or $\mathcal{A}_2$ query directly $\mathsf{KEF}(p^\star, pms^\star, \cdot)$, or $\mathcal{A}_2$ queries the decryption oracle with $p^\star, \ell^\star$ and a valid ciphertext $c \neq c^\star$ that decrypts to $pms^\star$. Note that the first abort condition would allow one to compute $pms^\star$ from $\mathcal{A}$'s queries using a plaintext-checking oracle, while the second condition yields a re-randomization of the challenge ciphertext. Moreover, since both ms_0 and ms_1 are random, the view of the adversary in this game is independent of the challenge bit b, which means that $\Pr[\mathsf{RCCA}_3 : b = b'] = 1/2$. Thus,

$$\Pr[\mathsf{RCCA}_2 : b = b'] - \Pr[\mathsf{RCCA}_3 : b = b'] = \Pr[\mathsf{RCCA}_2 : b = b'] - 1/2 \leq \Pr[\mathsf{RCCA}_3 : abort]$$

- RCCA_4. Since the view of the adversary is independent of the bit b and we only care about the probability of the simulation aborting, we drop b and give the adversary a random challenge ms_0 (unrelated to pms^*).

We reformulate the simulation of KEF and decryption queries using two maps Q_1 and Q_2 as follows:

```
Game RCCA_4 ≜
  pk, sk ← KeyGen()
  Q_1, Q_2, K, L := ∅
  (pms*, c*) ← enc(pv*, pk)
  ℓ* ← A_1^{KEF,DEC}(pk)
  ms_0, ms_1 ← $
  K(ℓ*) := {ms_0, ms_1}
  b' ← A_2^{KEF,DEC}(ms_0, c*)
```

```
Oracle KEF(p, pms, ℓ) ≜
  if (p, pms, ℓ) ∉ dom(Q_1) then
  Q_1(p, pms, ℓ) ← $
  if ℓ ∈ dom(Q_2) then
    (pv, h, ms, c) := Q_2(ℓ)
    if (pv, h) = p ∧ pms = dec(pv, sk, c)
    then Q_1(p, pms, ℓ) ← ms
  return Q_1(p, pms, ℓ)
```

```
Oracle DEC(p, ℓ, c) ≜
  if ℓ ∈ L ∨ p ∉ P then return ⊥
  L := L ∪ {ℓ}
  if (p, ℓ, c) = (p*, ℓ*, c*) then return ⊥
  (pv, h) := p; pms ← dec(pv, sk, c)
  if (p, pms, ℓ) ∈ dom(Q_1)
  then ms := Q_1(p, pms, ℓ) else ms ← $
  Q_2(ℓ) := (p, ms, c)
  if ms ∈ K(ℓ) then return ⊥
  return ms
```

The simulation is such that if (pv, h, pms, ℓ) is an entry in Q in RCCA_3, then either it is also in Q_1 and the associated ms values coincide, or else Q_2 maps ℓ to (pv, h, ms, c) where $ms = Q(pv, h, pms, \ell)$ and $\mathsf{dec}(pv, sk, c) = pms$. This allows the simulator to answer KEF and decryption queries consistently. Moreover, we have

$$
\begin{aligned}
&\Pr[\mathsf{RCCA}_3 : abort] \leq \\
&\Pr[\mathsf{RCCA}_4 : \exists \ell, (p^\star, pms^\star, \ell) \in \mathsf{dom}(Q_1)] + \\
&\Pr[\mathsf{RCCA}_4 : \ell^\star \in \mathsf{dom}(Q_2) \wedge \textbf{let } (pv, h, ms, c) = Q_2(\ell^\star) \textbf{ in } (pv, h) = p^\star \wedge c \neq c^\star \wedge \mathsf{dec}(pv, sk, c) = pms^\star]
\end{aligned}
$$

We bound each of the terms on the right-hand-side of this inequality independently using reductions to OW-PCA and NR-PCA.

- We use the following adversaries against OW-PCA and NR-PCA:

```
Adversary B^PCO(pk, c*) ≜
  Q_1, Q_2, K, L := ∅
  ℓ* ← A_1^{KEF,DEC}(pk)
  ms_0, ms_1 ← $
  K(ℓ*) := {ms_0, ms_1}
  b' ← A_2^{KEF,DEC}(ms_0, c*)
  foreach (pv, h, pms, ℓ) ∈ dom(Q_1) do
    if PCO(pv, pms, c*) then return pms
  return $
```

```
Adversary C^PCO(pk, c*) ≜
  Q_1, Q_2, K, L := ∅
  ℓ* ← A_1^{KEF,DEC}(pk)
  ms_0, ms_1 ← $
  K(ℓ*) := {ms_0, ms_1}
  b' ← A_2^{KEF,DEC}(ms_0, c*)
  (pv, h, ms, c) := Q_2(ℓ*)
  return c
```

Both adversaries simulate oracles KEF and DEC as in game RCCA_4, except that all checks are implemented using the PCO oracle rather than the secret key:

```
Oracle KEF(p, pms, ℓ) ≜
  if (p, pms, ℓ) ∉ dom(Q_1) then
  Q_1(p, pms, ℓ) ← $
  if ℓ ∈ dom(Q_2) then
    (pv, h, ms, c) := Q_2(ℓ)
    if (pv, h) = p ∧ PCO(pv, pms, c)
    then Q_1(p, pms, ℓ) ← ms
  return Q_1(p, pms, ℓ)
```

```
Oracle DEC(p, ℓ, c) ≜
  if ℓ ∈ L ∨ p ∉ P then return ⊥
  L := L ∪ {ℓ}
  if (p, ℓ, c) = (p*, ℓ*, c*) then return ⊥
  (pv, h) := p; pms := ⊥
  foreach (p', pms', ℓ') ∈ dom(Q_1) do
    if p = p' ∧ ℓ = ℓ' ∧ PCO(pv, pms', c) then pms := pms'
  if pms ≠ ⊥ then ms := Q_1(p, pms, ℓ) else ms ← $
  Q_2(ℓ) := (p, ms, c)
  if ms ∈ K(ℓ) then return ⊥
  return ms
```

We have $\Pr[\mathsf{RCCA}_4 : \exists \ell, (p^\star, pms^\star, \ell) \in \mathsf{dom}(Q_1)] \leq \mathbf{Adv}^{\mathsf{OW\text{-}PCA}}_{pv^\star, P'}(\mathcal{B})$ and

$$\Pr[\mathsf{RCCA}_4 : \ell^\star \in \mathsf{dom}(Q_2) \ \wedge \mathbf{let}\ (pv, h, ms, c) = Q_2(\ell^\star)\ \mathbf{in}\ (pv, h) = p^\star \wedge c \neq c^\star \wedge \mathsf{dec}(pv, sk, c) = pms^\star] \leq \mathbf{Adv}^{\mathsf{NR\text{-}PCA}}_{pv^\star, P'}(\mathcal{C})$$

Putting all the above results together,

$$\Pr[\mathsf{RCCA} : b' = b] - 1/2 \leq \mathbf{Adv}^{\mathsf{OW\text{-}PCA}}_{pv^\star, P'}(\mathcal{B}) + \mathbf{Adv}^{\mathsf{NR\text{-}PCA}}_{pv^\star, P'}(\mathcal{C}) + 2^{|pv|-|pms|}\,(q_{\mathsf{KEF}} + q_{\mathsf{DEC}})$$

from which the bound in the statement follows. Moreover, observe that under the convention that oracle calls have unit cost, the overhead of $\mathcal{B}$ and $\mathcal{C}$ is dominated by the cost of simulating decryption queries, which is $O(q_{\mathsf{KEF}})$ for a single query and $O(q_{\mathsf{DEC}} \cdot q_{\mathsf{KEF}})$ overall. □

3.3 Committed RCCA Security

The RCCA game has a seemingly artificial restriction, namely that an adversary has to query ENC on a label ℓ before using the same ℓ in a decryption query. Unless one designs reductions carefully, it is unlikely that such a restriction will be met by an arbitrary adversary in an interactive protocol. Indeed in TLS the adversary is in control of the network, and upon learning a server's nonce (completing a label), can ask it to decrypt a ciphertext under that label before sending the nonce on to the client. We found that [39] and an earlier version of our proof of the handshake did not account for such attackers.

The following slightly weaker but superficially more complicated *committed RCCA* definition removes this usage restriction, and replaces it with the requirement that the reduction (the adversary in the game) calls a COMMIT oracle before calling the DEC oracle. This is natural for TLS, where the server can commit to a label when it generates its nonce. The definition also replaces a result of $\perp$ upon decryption of a challenge master secret, by ideal decryption using table lookup. This makes the oracles more easy to use and similar to idealized libraries.

Definition 4 (Committed RCCA Security)**.** *Let* (KeyGen, Enc, Dec) *be an agile labeled KEM, P a set of agility parameters and $p^\star$ a public parameter (not necessarily in P). Consider the following game played between an adversary $\mathcal{A}$ and the challenger:*

```
Game CRCCA ≜
  pk, sk ← KeyGen()
  S, T := ∅
  b ← {0,1}
  b' ← A^{COMMIT,ENC,DEC}(pk)
  return (b' = b)

Oracle COMMIT(ℓ)
  if S(ℓ) ≠ ∅ then return ⊥
  S(ℓ) := S(ℓ) ∪ {c}
  if b then
    k_0, c ← Enc(p*, pk, ℓ)
    k_1 ← $
    T(ℓ) := (c, k_0, k_1)

Oracle ENC(ℓ) ≜
  if e ∈ S(ℓ) then return
  S(ℓ) := S(ℓ) ∪ {e}
  if b then
    if c ∉ S(ℓ) then
      k_0, c ← Enc(p*, pk, ℓ);
      k ← $
      T(ℓ) := (c, k_0, k)
    else (c, k_0, k) := T(ℓ)
  else k, c ← Enc(p*, pk, ℓ)
  return k, c

Oracle DEC(p, ℓ, c) ≜
  if c ∉ S(ℓ) ∨ d ∈ S(ℓ) ∨ p ∉ P
  then return ⊥
  S(ℓ) := S(ℓ) ∪ {d}
  k ← Dec(p, sk, ℓ, c)
  if b then
    (c_0, k_0, k_1) := T(ℓ)
    if k = k_0 then k := k_1
  return k
```

The challenger maintains a set of flags $S(\ell)$ for each label ℓ. $S(\ell)$ is initially $\varnothing$, flag c *is added when the adversary* commits *to ℓ,* e *when it queries* ENC *on ℓ and* d *when it queries* DEC *on ℓ. Encrypting or decrypting twice with the same label yields uninformative answers, and the adversary can only query both* ENC *and* DEC *on ℓ only if it first committed to the label.*

The IND-CRCCA advantage of $\mathcal{A}$, $\mathbf{Adv}^{\mathsf{CRCCA}}_{p^\star,P}(\mathcal{A})$ is defined as $2\Pr[\mathsf{CRCCA} : b' = b] - 1$. *We say the KEM is $(\epsilon, t, p^\star, P)$-secure against IND-CRCCA if the advantage of any adversary $\mathcal{A}$ running in time t is at most ϵ.*

Let ENC' and DEC' refer to the oracles of RCCA. An adversary $\mathcal{A}$ against CRCCA that makes q_{ENC} and q_{DEC} decryption queries, respectively, can be turned into an RCCA adversary $\mathcal{B}$ that achieves essentially the same advantage, but makes q_{ENC} extra decryption queries to ENC'. All that $\mathcal{B}$ has to do is to explicitly query its oracle ENC' on ℓ when $\mathcal{A}$ makes a decryption query with label ℓ; $\mathcal{B}$ answers using its oracle DEC', returning the key it gets from $\mathsf{ENC}'(\ell)$ if DEC' returns $\perp$.

Tolerating Weak Hash Functions and Ad Hoc Long-Term-Key Usage As shown in §1.2, many servers still accept MD5 for backward compatibility, so it is pragmatically important to protect (at least) clients that never accept MD5. To this end, instead of assuming a global random oracle for KEF, §B.1 provides a more realistic definition for *pms*-KEMs that suffices to prove security of the *ms*-KEM construction despite weak hash algorithms at the server.

Another practical concern is the sharing of long-term secret keys between signatures and KEMs. Accordingly, §B.2 gives joint security definitions, one for signatures schemes with a *ms*-KEM decryption oracle, and one for *ms*-KEM schemes with a signing oracle. This merely makes this real-world deployment assumption explicit—its assessment is left for future work.

4 Defining Agile Security for Multiple 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 of a handshake protocol Π through oracle queries, as 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.
- $\mathsf{Init}(\mathcal{R}, \mathit{cfg}_{\mathcal{R}})$ creates an instance with role $\mathcal{R}$ and local configuration $\mathit{cfg}_{\mathcal{R}}$; it returns a fresh handle i.
- $\mathsf{Send}_i(\mathit{frag})$ lets an existing instance 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.)
- $\mathsf{Control}_i(\mathit{env})$ 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.

Instances maintain private local state (e.g. using local variables and the state machine depicted in Figure 4). Each instance can go through a sequence of epochs (e.g. recording the number of cycles in the state machine). Each epoch records a list of *variable assignments*, extended as the result of calls to Send and $\mathsf{Control}$. 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 security definition focuses on assigned variables, which summarize what the Client and Server locally know so far about each epoch, rather than the fragments sent and received by the handshake. We use assignments to express the main goals of the protocol, for instance assigning a fresh random value to the connection-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 information extracted from the first flight of messages sent by the server.
$cert_{\mathcal{C}}$, $cert_{\mathcal{S}}$	client and server certificate chains. In TLS certificates are optional; for instance the assignment $cert_{\mathcal{C}} := \perp$ denotes the absence of client certificate.
$ex_{\mathcal{C}}$, $ex_{\mathcal{S}}$	client and server exchange variables, potentially secret, used below to specify safety.
k	record key for the epoch; in TLS, depending on a, it is usually split into 4 keys for MAC & encrypt.
complete	successful completion flag, marking the end of the handshake for the current 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 session key k, preventing its leakage through subsequent messages once assigned a random value.)[3] The agility-parameter variable a determines the algorithms and constructions used by the handshake. Our security properties are conditioned by a strength predicate, $\alpha(a)$, indicating whether those algorithms are strong enough to secure the epoch.

We deliberately avoid modelling certificate chains: for the handshake, they are treated as bitstrings. Certificates are faithfully modeled, but without security guarantees, to enable (as future work) the modeling of an application-level certificate infrastructure above the miTLS API. We assume existence of a public specification function $\mathsf{pk}(cert)$ that returns either the public key associated with a certificate chain, or $\perp$. 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.

Definition 5 (Honesty, Safety, Matching Algorithms, and Completion)**.** *For a handshake protocol* Π *and a strength predicate* $\alpha(\cdot)$*, an adversary that calls* KeyGen*,* Init*,* Send*, and* Control *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 the 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 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*

[3] In particular, the adversaries we consider, which in a compositional proof of TLS control the record layer, can read the session key k and thus compute encrypted finished messages.

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—and conversely for safe server sessions.*
  (Said otherwise, a session is safe *if $\alpha(a)$ holds and all static exchange variables and ephemeral peer-exchange variable assignments are honest.)*
- *A resumption is* safe *if $\alpha(a)$ holds and $\ell_{session}$ is the identifier of a safe and complete session.*
- *A epoch has matching algorithm $r = \mathsf{record}(a)$ when there is a peer epoch with the same ℓ and r.*
- *An epoch is* complete *when it includes the assignment complete $:= 1$.*

For TLS—jumping ahead to §5 for a concrete example—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 value $ex_{\mathcal{S}}$ 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.[4]

Definition 6 (Handshake Security)**.** *Let Π be a handshake protocol, $\alpha(\cdot)$ a strength predicate, and $\mathcal{A}$ an adversary that interacts with Π by calling* KeyGen*,* Init*,* Send*, and* Control *any number of times. Consider the following security 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:*** *informally, 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 one epoch has the following properties when $\mathcal{A}$ terminates: $\alpha(a)$ holds; the public key of the peer is honest; and the assignment to the peer exchange value is not honest (i.e. it was 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 protocol instances agree on all prior assignments.*
Let $\mathbf{Adv}^{\mathrm{I}}(\mathcal{A})$ be the probability that the following holds when $\mathcal{A}$ terminates: an instance created by $\mathsf{Init}(\mathcal{R}, cfg)$ assigns complete $:= 1$ in a safe epoch; and no instance created by $\mathsf{Init}(\overline{\mathcal{R}}, cfg')$ begins with a series of epochs with the same assignments to all variables (up to, but possibly excluding complete $:= 1$).

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

Discussion The properties 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 unique identifiers, authenticated using (4); we use these identifiers for matching client and server sessions. Property (2) enables, for instance, a client that trusts

[4] The use of *ms* instead of the KEM ciphertext and other public values allows us to prove security of the handshake, even if PKCS#1v1.5 ciphertexts are re-randomizable, despite NR-PCA being broken, as long as the *ms*-KEM is still IND-RCCA secure.

both the negotiated algorithm and the server certificates to deduce that the server-exchange variables are honest, and conclude that the session is safe.

Property (3) idealizes the derived key; this is key usability. Recall that TLS uses the key before the two parties actually agree on the record algorithms (4). Accordingly, (3) idealizes only when the record algorithms match. (§B.5 defines an alternative property for constructions that deliver fixed-sized keys irrespective of the algorithm, but such constructions require record agility.) As Krawczyk et al. [39] our formal development for miTLS does not consider forward secrecy, but we discuss forward secure variants of Verified Safety and Agile Key Derivation in §B.5.

Property (4) applies to all variable assignments of the peers since their creation, not just those 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 only holds thanks to the (mandatory) secure renegotiation extension, which links each epoch to its predecessor. On the other hand, the final assignment to *complete* is not itself authenticated, as the two instances asynchronously complete the epoch. Similarly, the *ordering* of assignments at the client and at the server may differ, as illustrated in Figure 1.

Compared with classic key exchange definitions, and the key exchange part of ACCE [30], our definition guarantees 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 (see §7.2). Our definition also provides (some) security for anonymous connections, which can be composed with other authentication mechanisms to achieve application-level security. For example, renegotiation with client and server certificates may provide mutual authentication on top of an initial safe but anonymous handshake; and [9] show that late, application-level, client password authentication can yield mutual authentication with miTLS.

Unlike previous analyses of TLS, our definitions also account for resumption. Property (4) guarantees agreement in the identifier $\ell_{session}$ of the resumed session. Hence, if the resumed epoch is safe,[5] and if session secrets are securely stored and correctly used, an application obtains agreement on all variables assigned in the original session. In particular, the peer agrees on the client and server identities, even though these variables are not re-exchanged during resumption.

Note that agreements on renegotiation (the sequence of epochs in a connection) and resumption (the original session) do not directly compose—enabling our triple-handshake attacks despite TLS meeting our definitions. It may seem desirable to guarantee a stronger property: for all safe and complete epochs, there is a safe epoch in the other role such that not only the two instances, but also the two whole trees of instances connected by additional $\ell - \ell_{session}$ edges, agree on all prior assignments. However, this is not guaranteed by TLS (nor our definition), e.g., an unsafe resumption followed by a safe & complete renegotiation does not guarantee agreement on the resumed session. While secure applications can be built on top of our current interface, we found that mainstream TLS applications incorrectly assume that the renegotiation indication extension already implies the stronger property. This leads to practical man-in-the-middle attacks over TLS, much like the renegotiation attack of Ray [55]. The stronger property can be achieved by a protocol extension that includes a hash of the log of the resumed session in resumption handshakes. We describe both the attack and its countermeasure briefly in §7.2 and in more detail in a paper in the online materials.

[5]Our triple-handshake attack involves unsafe resumptions.

5 Proving Agile Security for TLS Handshakes

We are now ready to reduce the security of TLS handshakes to the security of three agile libraries $\mathcal{S}$, $\mathcal{E}$, $\mathcal{D}$ for signing (§2), key encapsulation (§3), and KDF-MAC (§B.3). This last library provides an intermediate abstraction, keyed by master secrets and used both for deriving record keys (using KDF) and producing finished-message tags (using MAC). In §B.3, we define its security and show that the construction used in TLS, essentially a keyed hash with separate labels for key derivation and for MACing, is secure under the agile-PRF assumption proposed by Acar et al. [2].

In §B.3 we model key derivation in two steps, first as an agile family of PRFs, then as an agile functionality that separates its different usages and ensures that the derived record key is used with the same algorithms by the client and by the server. To elide details handled in the miTLS implementation, such as output lengths depending on agile parameters, we assume the output of PRF is long enough to cover all TLS ciphersuites. Let $\lfloor . \rfloor_r$ and $\lfloor . \rfloor_p$ be functions that truncate to the record-key and MAC sizes, respectively. We define functionally correct algorithms by truncations: $\mathsf{KDF}(p, ms, \ell, r) \triangleq \lfloor \mathsf{PRF}(p, ms, \texttt{"key expansion"} \,\|\, \overline{\ell}) \rfloor_r$ and $\mathsf{MAC}(p, ms, t, v) \triangleq \lfloor \mathsf{PRF}(p, ms, t \,\|\, v) \rfloor_p$ where t is either `"client finished"` or `"server finished"` and $\overline{\ell}$ is ℓ after swapping the client and server randoms.

We structure the proof to apply simultaneously to the protocol, illustrated in Figures 1 and 2 (for easy reference), and to its miTLS reference implementation. Figure 1 shows the assignments performed by two successive TLS handshakes on the same connection: a static session, followed by a resumption. Figure 2 in the appendix similarly shows an ephemeral session. In particular, we restructure the game-based definitions for KEM and PRF respectively in such a way that the oracles of the libraries $\mathcal{E}$ and $\mathcal{D}$ reflect the flow of the protocol and preserve the input-output behaviour of the cryptographic primitives. Following the flow, the server first calls $\mathcal{E}.\mathsf{Commit}(e, p_{\mathcal{E}}, pk, \ell)$ and $\mathcal{D}.\mathsf{Commit}(r, p_{\mathcal{D}}, \ell)$ to fix input values for these algorithms to be used later with a particular nonce ℓ, e.g. the record algorithm r for key derivation. As a first step, our proof thus involves hybrid arguments for the signature game (see §2), and these extended KEM and KDF games (see §3.3 and §B.3) that range over all honest keys to lift security to multi-key libraries. These libraries also implement weak algorithms and support dishonest keys. This yields the constructions $\mathcal{S}$, $\mathcal{E}$, and $\mathcal{D}$ of the figures, tightly related to the modules of our reference implementation of the handshake.

The agility parameter a of the handshake indicates which algorithm to use for each underlying functionality. We write for instance $s, p := \mathsf{sig}(a)$ to retrieve the signature algorithm and public parameter of Definition 9. Figure 4 in §C depicts when these assignments are performed in the state machine of the miTLS implementation.

Our second main theorem reduces the security of TLS to the security of its underlying algorithms, via the definition of the *strength predicate* α on agility parameters. Its proof is in §B.4, and relies on intermediary definitions in §3.3 and §B.3.

Theorem 4 (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 5) such that the following assumptions hold:*

(1) *If $\alpha(a)$ and $s, p := \mathsf{sig}(a)$, then S_s is $(\epsilon_{s,p}, t_{s,p}, p, P_s)$-secure against EUF-CMA.*

(2) *If $\alpha(a)$ and $e, p := \mathsf{kem}(a)$, then E_e is $(\epsilon_{e,p}, t_{e,p}, p, P_e)$-secure against IND-RCCA-n_{ms}.*

(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*

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

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

Discussion The definition of sets P_s, P_e, and P above considers 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 Code-Based Verified Implementation

We jointly programmed the TLS handshake and developed its proof. We finally present 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 §C. 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 (see Table 1). It deals mostly with the protocol aspects; indeed, our cryptographic proof for Theorem 3, conducted with EasyCrypt, concerns less than 200 lines of F#. Conversely, Theorem 4 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. [23], as previously applied to miTLS by Bhargavan et al. [see 9, §2]. The miTLS library consists of 45 modules, not counting application code or platform libraries, as depicted in Figure 3. 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 state machine). Each module's 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 outline the design of three important components that we modified during the course of this paper. *TLSInfo* defines agility parameters and logical predicates (corresponding to Definition 5) that specify algorithmic strength (α), honesty for both long-term-keys and ephemeral secrets, matching record algorithms, and handshake completion events. This new logical model is more detailed than Bhargavan et al. [9]; 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 6(4)) depends on it, 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), shown in Figure 4 for the client. Its code is not as simple as suggested by the KEMs of §3, since the protocol standard employs different sequences of messages for (say) RSA and DHE. 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 our prior results on the rest of miTLS [9] (the record and alert layers, the top-level API, and various applications) yield agile, verified application security for TLS as it is.

7 Related Work

7.1 Prior Security Results on the TLS Handshake

Research on secure key exchange usually follows either a game-based approach or a simulation-based approach, as pioneered by Bellare and Rogaway [7] and Canetti and Krawczyk [14], respectively. Gajek et al. [24] outline a proof of security of TLS in the simulation-based model of [13].

However, Küsters and Tuengerthal [42] correctly note that their (ab)use of a crucial theorem to obtain multi-session security relies on pre-established identifiers not available in TLS, and suggest a framework for overcoming this limitation.

Most of the cryptographic work on TLS follows the game-based approach. Jonsson and Kaliski [33] analyze the core of the RSA ciphersuites. Morrissey et al. [50] analyze a variant of the protocol using a modular approach that decomposes the handshake into *pre-master secret*, *master secret*, and *record-key derivation* phases. Both of these works influence our analysis. To pinpoint some differences, Jonsson and Kaliski already propose to model part of the handshake as a KEM with one-time nonces, but their KEM includes the record-key derivation and finished messages, and is thus not modular in the sense of Morrissey et al.. Although Morrissey et al. show how to boost security using a weakly secure (only one-way secure) pre-master secret phase, they do not separately model this phase as a KEM. As an advantage of their construction, same master secret can be used to derive multiple keys. However, they still rely on one-way security for record-key derivation, hence their analysis is more globally dependent on random oracles than ours.

Recently, there has been renewed interest in the security (and insecurity) of TLS. Jager et al. [30] perform a game-based security analysis of the `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA` ciphersuite, relying on the analysis of Paterson et al. [52] for the record protocol. A defining feature of their analysis is that they do not give a definition of TLS handshake security. Instead they define authenticated and confidential channel establishment (ACCE) security for the whole TLS protocol. Similarly, Kohlar et al. [37] study the ACCE security of TLS-RSA ciphersuites when instantiated with an IND-CCA secure key transport encryption scheme. Again, this defeats the modularity goals of Morrissey et al.. Brzuska et al. [12] propose a more composable game-based analysis technique and use TLS as a case study. They do, however, also assume that the key transport encryption scheme is IND-CCA. Giesen et al. [26] extend the work of Jager et al. with an analysis of secure session renegotiation, while Krawczyk et al. [39] extend it to support RSA and server-only authentication ciphersuites without having to assume IND-CCA security for PKCS#1v1.5. Similarly to Jonsson and Kaliski and us, they use a KEM abstraction for the cryptographic core of TLS. However, their analysis is for one fixed ciphersuite at a time, and all bets are off if the adversary tricks the client and server into using different algorithms. Moreover, it inherits the monolithic structure of ACCE, which makes it hard to reason modularly, e.g. to cover resumption.

The first step in this direction is the work of Bhargavan et al. [9] on a security proof conducted on a reference implementation of the TLS standard, using a combination of type checking and automated verification tools. In the present work, starting from the same code base, we develop a more abstract, human-readable, game-based proof that improves on Bhargavan et al. and makes their results more accessible. Like them we support renegotiation, resumption, and multiple ciphersuites. In the process, we clarify their definitions and modular structure. In particular we adapt the KEM concept to reason about both the pre-master secret and master secret phases, which allows us to generalize the result of Jonsson and Kaliski, similarly to Krawczyk et al. but without sacrificing modularity (Krawczyk et al. consider KEM keys that include unencrypted finished messages). Moreover, we use EasyCrypt to machine check the proof of this theorem.

Recently, and independently of our work, Dowling et al. [19] studied the ACCE security of multi-ciphersuite protocols that reuse long-term keys. They "open" the ACCE definition, and show that under a global condition on key reuse, single ciphersuite security implies multi-ciphersuite security. Their positive results, however, only regard SSH. They make mention of our work and acknowledge that the TLS protocol is in general not multi-ciphersuite secure and that a finer analysis is necessary to establish whether particular combinations of ciphersuites preserve security.

In parallel with our work, Kohlweiss et al. [38] conduct an extensive proof of TLS following the constructive cryptography paradigm of Maurer [46]. Their results and ours co-evolved. In particular, they adopted our approach for proving TLS-RSA modularly based on the assumption that PKCS#1v1.5 ciphertexts are hard to re-randomize. In a nutshell, their work can be seen as a simulation-based and single-ciphersuite analogue to ours.[6] It demonstrates the power, and some limitations, of the constructive cryptography approach to deal with real-world protocols. Irrespective of the elegance of the modeling language, we are, however, convinced that some amount of tool support is crucial to deal with the haystack of details bestowed upon us by the TLS standard.

7.2 Attacks involving Multiple Algorithms and Handshakes

Meyer and Schwenk [49] conducted a survey of previous attacks on SSL and TLS. Here, we mention a few attacks to motivate our definitions and theorems. We begin with historical attacks and end with new attacks discovered by us.

Version and Ciphersuite Rollback Attacks SSL version 2.0 is vulnerable to both version and ciphersuite rollback attacks [60], because its handshake protocol does not protect the integrity of these parameters. Hence, if a client and server support both TLS 1.0 and SSL2, a man-in-the-middle adversary can force them to use SSL2. Furthermore, he can force them to use a weak authenticated encryption scheme, e.g. 40-bit RC2 even if they both support AES.

All TLS versions since SSL3 protect the integrity of the full handshake and SSL2 has been deprecated [59]. miTLS does not support SSL2, and our Theorem 4 guarantees agreement over all handshake parameters, including the version and ciphersuite, on safe epochs, that is, when both peers are honest and negotiate strong handshake algorithms.

Key Exchange Confusion Attacks on Server Signatures The `ServerKeyExchange` message in the TLS handshake typically contains a signature on the KEM's public key. For example, in DHE ciphersuites, this key consists of the server-chosen Diffie-Hellman group and the server's public key. In ECDHE, it indicates the elliptic curve and contains the server's public key. In the (now rarely used) ephemeral RSA KEMs, it is a short-lived RSA modulus and exponent.

If a server signature generated for one KEM can be successfully used at a recipient who is using a different KEM, i.e. if the public keys of different KEM schemes can be confused, then an adversary can potentially impersonate the server without needing to know its private key. Wagner and Schneier [60]

[6] To our knowledge, in this case "simulation-based" does not imply that their definitions are strictly stronger than ours. Rather, they are of a similar flavor, but because of the sheer amount of details most likely formally incomparable.

show how DHE public keys can be confused with ephemeral RSA, and Mavrogiannopoulos et al. [48] show how ECDHE public keys can be confused with DHE. The success probability of these attacks depends on implementation details; in practice, this is small but not negligible.

In miTLS, the *Sig* module that implements signatures specifies all the possible usages of a signature key, including the possible contents of `ServerKeyExchange` and `ClientCertificateVerify`. If the same key may be used to sign two different messages, we must prove that the formats of these messages are disjoint and hence, that the signature is unambiguous. miTLS does not support ECDHE or ephemeral RSA, but we prove, for example, that the implementation cannot confuse client logs with DHE group parameters. When adding new KEMs to the implementation, we would need to prove such disjointness properties for those KEM's public keys as well.

Client Impersonation Attacks on Renegotiation A mutually authenticated TLS handshake communicates client and server identities in the clear. To increase privacy, one may instead start a TLS connection with a handshake where one or both peers are anonymous, and then run a new mutually-authenticated renegotiation handshake within the protected channel. There may also be other reasons to use renegotiation, such as rekeying a long-lived connection, upgrading to a different ciphersuite, or replacing expired certificates.

However, whenever a key exchange protocol is tunneled within another, it becomes vulnerable to a generic man-in-the-middle attack on the outer protocol [3]. Indeed, two instances of such attacks were found on TLS renegotiation by Ray [55] and Rex [57]. In the first instance, if a client starts an initial handshake with a server, an adversary could forward these handshake messages as a renegotiation within an existing TLS connection between the adversary and the server. Both client and server will successfully complete the handshake. However, the server will believe the client's messages to be a continuation of the adversary's connection, whereas the client is oblivious to this tunneling and believes it is beginning a new connection.

The recommended countermeasure is to link the renegotiation handshake with its preceding epoch, and has been standardized as a mandatory extension for all versions of TLS [56]. miTLS supports this extension and consequently, Theorem 4 guarantees that at the completion of a safe epoch, both client and server agree upon all previous epochs on the connection. However, this guarantee does not carry over to link different connections that resume the same original, as we discuss below.

Triple Handshake Attacks on Renegotiation after Resumption As a consequence of our formal investigation of the TLS handshake, we discovered a new man-in-the-middle attack on TLS renegotiation [7] of comparable severity to Ray and Rex's attacks. To summarize briefly, a man-in-the-middle adversary can (still) impersonate an honest client during renegotiation, if the renegotiation occurs on a new connection, after session resumption.

This triple-handshake attack relies on three weaknesses of the TLS handshake. First, the RSA and DHE KEMs allow unknown key-share attacks: if a client connects to a malicious server, the server can set up the same master secret (and record keys) on a different connection to an honest server. Second, the resumption handshake does not explicitly re-authenticate the full session; its transcript depends only on the master secret and selected elements of the session, such as the ciphersuite and (non-unique) session identifier, but notably not the client and server identities. Third, the renegotiation handshake (even with the mandatory extension) guarantees agreement only for variable assignments of previous epochs on the same connection, but not for the assignments of any session being resumed. Using these three weaknesses,

[7] https://www.secure-resumption.com/

we are able to set up a malicious server such that if an honest client connects to our server and presents a client certificate, the server can impersonate the client at an honest server, by using a sequence of three handshakes over two connections (initial, resumption, renegotiation.)

The handshake implementation for miTLS carefully indexes sessions by a unique resumption identifier which allows careful applications to verify that the resumption and subsequent renegotiation corresponds to the intended recipients. As an example, we build miHTTPS a secure implementation of the HTTPS protocol and verify its security on top of the miTLS API, even as it uses arbitrary combinations of resumptions and renegotiations. Unfortunately, most applications are not that careful and require additional protocol-level countermeasures.

More generally, the above weaknesses in the TLS handshake reveal a family of attacks on authentication protocols over TLS.

Plaintext Recovery Attacks on Encrypted Extensions Many recent proposed extensions to TLS optimistically send encrypted data even before the handshake is fully complete. One motivation is to improve latency by reducing the number of roundtrips that a client needs to wait for before sending application data. For example, the False Start extension of Langley and Moeller [45] allows the client to send data immediately after the `ClientFinished` message, without waiting for `ServerFinished`. This extension is implemented by all Google websites, and by Chrome and Firefox. A second motivation is to improve the privacy of the handshake by sending some messages encrypted. The Next Protocol Negotiation (NPN) extension of Langley [44] (implemented by all major websites and browsers) sends an encrypted message after the `ChangeCipherSpec` message but before the `Finished` message. Such extensions are fragile against both implementation flaws and ciphersuite weaknesses. We outline a concrete plaintext-recovery attack against some client implementations and then discuss the agility requirements imposed by such extensions.

We found that some client implementations, such as Firefox and Chrome, only validate the server certificate (say against the server name) at the end of the handshake. So, if an active attacker replaces the server certificate with his own, all messages sent before the handshake is complete are encrypted for the adversary, leading to a plaintext recovery attack. When the handshake completes, the invalid certificate is detected and the connection is torn down, but it is too late for the messages that were already sent. We mounted such attacks on encrypted NPN messages sent by Firefox and Chrome. More seriously, we were also able to recover encrypted user-identifying Channel IDs of [4] sent by Chrome.

The confidentiality of optimistically encrypted messages relies on the ciphersuites accepted by the client, since a man-in-the-middle adversary will be able to downgrade the client to its weakest ciphersuite regardless of the server; this ciphersuite rollback will be detected only when the handshake completes. As a countermeasure, extensions like False Start restrict the agility of the TLS handshake by requiring the ciphersuite to use symmetric ciphers with at least 128 bit keys (RC4!, AES) and strong key-exchange methods (DHE_RSA, ECDHE_RSA, DHE_DSS, ECDHE_ECDSA). However, MD5 is still allowed as a hash algorithm during False Start.

In our implementation, we forbid sending application and handshake data between `ChangeCipherSpec` and `Finished`. Our handshake definition does not guarantee confidentiality for keys before handshake completion. To support False Start, we would need to modify our definition as described in B.5 and would require record algorithms that satisfy stronger agile security properties, since the algorithms used by the client for encryption and the server for decryption may differ. More generally, using the same record keys with different algorithms makes security proofs more difficult. Instead, we advocate a new master secret derivation algorithm (also described in the draft paper at `https://www.secure-resumption.com/`) that ensures that record keys are context-bound to their intended ciphersuites.

References

[1] M. Abdalla, M. Bellare, and P. Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2001.

[2] T. Acar, M. Belenkiy, M. Bellare, and D. Cash. Cryptographic agility and its relation to circular encryption. In *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 403–422. Springer, 2010.

[3] N. Asokan, V. Niemi, and K. Nyberg. Man-in-the-middle in tunnelled authentication protocols. In *11th International Workshop on Security Protocols*, volume 3364 of *Lecture Notes in Computer Science*, pages 28–41. Springer, 2005.

[4] D. Balfanz and R. Hamilton. Transport Layer Security (TLS) Channel IDs. IETF Internet Draft draft-balfanz-tls-channelid-01, 2013.

[5] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[6] G. Barthe, D. Pointcheval, and S. Zanella-Béguelin. Verified security of redundancy-free encryption from Rabin and RSA. In *19th ACM Conference on Computer and Communications Security, CCS 2012*, pages 724–735. ACM, 2012.

[7] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology – CRYPTO 1993*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1993.

[8] M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2000.

[9] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptograhic security. In *2013 IEEE Symposium on Security and Privacy, SP 2013*, pages 445–469. IEEE, 2013.

[10] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

[11] B. Brumley, M. Barbosa, D. Page, and F. Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In *Topics in Cryptology – CT-RSA 2012*, volume 7178 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2011.

[12] C. Brzuska, M. Fischlin, N. P. Smart, B. Warinschi, and S. C. Williams. Less is more: Relaxed yet composable security notions for key exchange. Cryptology ePrint Archive, Report 2012/242, 2012. `http://eprint.iacr.org/`.

[13] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001*, pages 136–145. IEEE, 2001.

[14] R. Canetti and H. Krawczyk. Universally composable notions of key exchange and secure channels. In *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2002.

[15] R. Canetti, H. Krawczyk, and J. B. Nielsen. Relaxing chosen-ciphertext security. In *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 565–582. Springer, 2003.

[16] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[17] J. P. Degabriele, A. Lehmann, K. G. Paterson, N. P. Smart, and M. Strefler. On the joint security of encryption and signature in EMV. In *Topics in Cryptology - CT-RSA 2012*, volume 7178 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2012.

[18] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2, 2008.

[19] B. Dowling, F. Giesen, F. Kohlar, J. Schwenk, and D. Stebila. Multi-ciphersuite security and the SSH protocol. Cryptology ePrint Archive, Report 2013/813, 2013. http://eprint.iacr.org/.

[20] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking SSL development in an appified world. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13*, pages 49–60. ACM, 2013.

[21] M. Fischlin, A. Lehmann, and D. Wagner. Hash function combiners in TLS and SSL. In *Topics in Cryptology – CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2010.

[22] P.-A. Fouque, D. Pointcheval, and S. Zimmer. HMAC is a randomness extractor and applications to TLS. In *ASIACCS*, pages 21–32, 2008.

[23] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *ACM CCS*, pages 341–350, 2011.

[24] S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, and J. Schwenk. Universally composable security analysis of TLS. In *ProvSec*, pages 313–327, 2008.

[25] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *CCS*, pages 38–49, 2012.

[26] F. Giesen, F. Kohlar, and D. Stebila. On the security of TLS renegotiation. Cryptology ePrint Archive, Report 2012/630, 2012. http://eprint.iacr.org/.

[27] S. Haber and B. Pinkas. Securely combining public-key cryptosystems. In M. K. Reiter and P. Samarati, editors, *ACM Conference on Computer and Communications Security*, pages 215–224. ACM, 2001.

[28] S. Haber and B. Pinkas. Securely combining public-key cryptosystems. In *8th ACM Conference on Computer and Communications Security, CCS'01*, pages 215–224. ACM, 2001.

[29] IANA. Transport Layer Security (TLS) parameters. http://www.iana.org/assignments/tls-parameters/.

[30] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, pages 273–293, 2012.

[31] T. Jager, K. G. Paterson, and J. Somorovsky. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *NDSS*. The Internet Society, 2013.

[32] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1, 2003. RFC 3447.

[33] J. Jonsson and B. S. Kaliski. On the security of RSA encryption in TLS. In *CRYPTO*, pages 127–142, 2002.

[34] J. Kelsey, B. Schneier, and D. Wagner. Protocol interactions and the chosen protocol attack. In B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, editors, *Security Protocols Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 91–104. Springer, 1997.

[35] V. Klíma and T. Rosa. Further results and considerations on side channel attacks on rsa. In B. S. K. Jr., Çetin Kaya Koç, and C. Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2002.

[36] V. Klima, O. Pokorny, and T. Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, pages 426–440, 2003.

[37] F. Kohlar, S. Schge, and J. Schwenk. On the security of TLS-DH and TLS-RSA in the standard model. Cryptology ePrint Archive, Report 2013/367, 2013. http://eprint.iacr.org/.

[38] M. Kohlweiss, U. Maurer, C. Onete, B. Tackmann, and D. Venturi. (De-)Constructing TLS. Cryptology ePrint Archive, Report 2014/020, 2014. http://eprint.iacr.org/.

[39] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448. Springer, 2013.

[40] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the tls protocol: A systematic analysis. Cryptology ePrint Archive, Report 2013/339, 2013. http://eprint.iacr.org/.

[41] U. Kühn, A. Pyshkin, E. Tews, and R.-P. Weinmann. Variants of bleichenbacher's low-exponent attack on pkcs#1 rsa signatures. In A. Alkassar and J. H. Siekmann, editors, *Sicherheit*, volume 128 of *LNI*, pages 97–109. GI, 2008.

[42] R. Küsters and M. Tuengerthal. Composition theorems without pre-established session identifiers. In *CCS*, pages 41–50, 2011.

[43] A. Langley. Unfortunate current practices for HTTP over TLS, 2011. http://www.ietf.org/mail-archive/web/tls/current/msg07281.html.

[44] A. Langley. Transport Layer Security (TLS) Next Protocol Negotiation Extension. Internet Draft, 2012.

[45] N. M. Langley, A. and B. Moeller. Transport Layer Security (TLS) False Start. Internet Draft, 2010.

[46] U. Maurer. Constructive cryptography: A new paradigm for security definitions and proofs. In *TOSCA 2011—Theory of Security and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2011.

[47] N. Mavrogiannopoulos. Preventing cross-protocol attacks in TLS protocol. Internet Draft, http://www.cosic.esat.kuleuven.be/publications/article-2222.pdf, 2012.

[48] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *CCS*, pages 62–72, 2012.

[49] C. Meyer and J. Schwenk. Lessons learned from previous SSL/TLS attacks - a brief chronology of attacks and weaknesses. Cryptology ePrint Archive, Report 2013/049, 2013. http://eprint.iacr.org/.

[50] P. Morrissey, N. Smart, and B. Warinschi. A modular security analysis of the TLS handshake protocol. In *ASIACRYPT'08*, pages 55–73, 2008.

[51] D. Naccache and I. E. Shparlinski. Divisibility, Smoothness and Cryptographic Applications. *ArXiv e-prints*, Oct. 2008.

[52] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT 2011*, pages 372–389, 2011.

[53] K. G. Paterson, J. C. N. Schuldt, M. Stam, and S. Thomson. On the joint security of encryption and signature, revisited. In D. H. Lee and X. Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 161–178. Springer, 2011.

[54] Qualys SSL labs. SSL server test. https://www.ssllabs.com/ssltest/analyze.html.

[55] M. Ray. Authentication gap in TLS renegotiation. http://extendedsubset.com/Renegotiating_TLS.pdf, 2009.

[56] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. TLS renegotiation indication extension. RFC 5746, 2010.

[57] M. Rex. MITM attack on delayed TLS-client auth through renegotiation. http://www.ietf.org/mail-archive/web/tls/current/msg03928.html, 2009.

[58] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. Cryptology ePrint Archive, Report 2009/111, 2009. http://eprint.iacr.org/.

[59] S. Turner and T. Polk. Prohibiting secure sockets layer (SSL) version 2.0. RFC 6176, 2011.

[60] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *2nd USENIX Workshop on Electronic Commerce (WOEC'96)*, pages 29–40. USENIX Association, 1996.

A Empirical Results on TLS Configurations

We present empirical results on the TLS configurations deployed at 215 out of the 500 most popular websites as measured by Alexa. These results were compiled with the aid of Qualys SSL Labs analyzer [54].

Supported Protocol Versions

SSL2	7	3.26 %
SSL3	212	98.60 %
TSL 1	214	99.53 %
TSL 1.1	129	60.00 %
TSL 1.2	124	57.67 %

Avg. supported TLS versions per host: 3.19

Popular Protocol Extensions

Secure renegotiation	185	86.05 %
Session ticket	128	59.53 %

Agility Summary

Ciphersuites count	64
Ciphersuites avg. per host	11.88
Ciphersuites std. dev.	6.44

Avg. hash algorithms per host	2.52
Avg. encryption algorithms per host	5.36
Avg. signature algorithms per host	1.06
Avg. KEMs per host	1.73

Hash algorithms

MD5	149	69.30 %
SHA	215	100.00 %
SHA256	103	47.91 %
SHA384	74	34.42 %

Signature algorithms

ECDSA	13	6.05 %
RSA	215	100.00 %

KEMs

DHE	61	28.37 %
ECDH	2	0.93 %
ECDHE	94	43.72 %
RSA	215	100.00 %

Encryption algorithms

3DES_EDE_CBC	207	96.28 %
AES_128_CBC	212	98.60 %
AES_128_GCM	78	36.28 %
AES_256_CBC	212	98.60 %
AES_256_GCM	74	34.42 %
CAMELLIA_128_CBC	34	15.81 %
CAMELLIA_256_CBC	34	15.81 %
DES40_CBC	17	7.91 %
DES_CBC	23	10.70 %
IDEA_CBC	14	6.51 %
NULL	3	1.40 %
RC2_CBC_40	17	7.91 %
RC2_CBC_56	1	0.47 %
RC4_128	195	90.70 %
RC4_40	17	7.91 %
RC4_56	3	1.40 %
SEED_CBC	11	5.12 %

Supported Ciphersuites

SSL_CK_DES_192_EDE3_CBC_WITH_MD5	7	3.26%	SSL_CK_DES_64_CBC_WITH_MD5	6	2.79%
SSL_CK_IDEA_128_CBC_WITH_MD5	1	0.47%	SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5	6	2.79%
SSL_CK_RC2_128_CBC_WITH_MD5	6	2.79%	SSL_CK_RC4_128_EXPORT40_WITH_MD5	6	2.79%
SSL_CK_RC4_128_WITH_MD5	7	3.26%	TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	5	2.33%
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	57	26.51%	TLS_DHE_RSA_WITH_AES_128_CBC_SHA	61	28.37%
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	9	4.19%	TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	9	4.19%
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	61	28.37%	TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	9	4.19%
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	9	4.19%	TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	25	11.63%
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	25	11.63%	TLS_DHE_RSA_WITH_DES_CBC_SHA	8	3.72%
TLS_DHE_RSA_WITH_SEED_CBC_SHA	6	2.79%	TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	13	6.05%
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	13	6.05%	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	13	6.05%
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	13	6.05%	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	13	6.05%
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	13	6.05%	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	13	6.05%
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	13	6.05%	TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	77	35.81%
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	94	43.72%	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	74	34.42%
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	73	33.95%	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	92	42.79%
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	72	33.49%	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	73	33.95%
TLS_ECDHE_RSA_WITH_NULL_SHA	1	0.47%	TLS_ECDHE_RSA_WITH_RC4_128_SHA	75	34.88%
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA	2	0.93%	TLS_ECDH_anon_WITH_AES_128_CBC_SHA	2	0.93%
TLS_ECDH_anon_WITH_AES_256_CBC_SHA	2	0.93%	TLS_ECDH_anon_WITH_NULL_SHA	1	0.47%
TLS_ECDH_anon_WITH_RC4_128_SHA	2	0.93%	TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA	3	1.40%
TLS_RSA_EXPORT1024_WITH_RC2_CBC_56_MD5	1	0.47%	TLS_RSA_EXPORT1024_WITH_RC4_56_MD5	1	0.47%
TLS_RSA_EXPORT1024_WITH_RC4_56_SHA	3	1.40%	TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	17	7.91%
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	17	7.91%	TLS_RSA_EXPORT_WITH_RC4_40_MD5	17	7.91%
TLS_RSA_WITH_3DES_EDE_CBC_SHA	207	96.28%	TLS_RSA_WITH_AES_128_CBC_SHA	210	97.67%
TLS_RSA_WITH_AES_128_CBC_SHA256	96	44.65%	TLS_RSA_WITH_AES_128_GCM_SHA256	76	35.35%
TLS_RSA_WITH_AES_256_CBC_SHA	210	97.67%	TLS_RSA_WITH_AES_256_CBC_SHA256	96	44.65%
TLS_RSA_WITH_AES_256_GCM_SHA384	72	33.49%	TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	33	15.35%
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	33	15.35%	TLS_RSA_WITH_DES_CBC_SHA	22	10.23%
TLS_RSA_WITH_IDEA_CBC_SHA	14	6.51%	TLS_RSA_WITH_NULL_MD5	3	1.40%
TLS_RSA_WITH_NULL_SHA	3	1.40%	TLS_RSA_WITH_RC4_128_MD5	149	69.30%
TLS_RSA_WITH_RC4_128_SHA	194	90.23%	TLS_RSA_WITH_SEED_CBC_SHA	10	4.65%

Clients statistics

Browser	**TLS version (max)**	**Secure renegotiation**	**# Ciphers**
Chrome 30.0.1599.69 (MAC,win8)	TLS1.2	Yes	20
Firefox 24 (MAC,win8)	TLS1	Yes	36
Safari 6.0.5 (MAC)	TLS1	No	27
Opera 12.16 (MAC)	TLS1	Yes	27
Opera 16 (win8)	TLS1.1	Yes	20
IE 11.0.9431 (win8)	TLS1.2	Yes	19
Chrome 30.0.1599.82 (android)	TLS1.2	Yes	38
Android Browser 4.2.2 (android)	TLS1	Yes	33
Dolphin v10 (android)	TLS1	Yes	33
CyanogenMod/10.1.3 (android)	TLS1	Yes	33
Safari (iOS 6.1.3)	TLS1.2	Yes	43

Browser	**KEM**	**Hash**	**Signature**
Chrome 30.0.1599.69 (MAC,win8)	ECDHE, DHE, RSA	SHA, SHA256, MD5	ECDSA, RSA
Firefox 24 (MAC,win8)	ECDHE, DHE, ECDH, RSA	SHA, MD5	ECDSA, RSA, DSS, FIPS
Safari 6.0.5 (MAC)	ECDHE, ECDH, RSA, DHE	SHA, MD5	ECDSA, RSA, DSS
Opera 12.16 (MAC)	DHE, DH, RSA	SHA, MD5	RSA, DSS
Opera 16 (win8)	ECDHE, DHE, RSA	SHA, MD5	ECDSA, RSA, DSS
IE 11.0.9431 (win8)	RSA, ECDHE, DHE	SHA256, SHA, SHA384	RSA, ECDSA, DSS
Chrome 30.0.1599.82 (android)	ECDHE, DHE, RSA	SHA, SHA256, MD5	ECDSA, RSA
Android Browser 4.2.2 (android)		SHA, MD5	ECDSA, RSA, DSS
Dolphin v10 (android)	ECDHE, SRP, DHE, RSA	SHA, MD5	ECDSA, RSA, DSS
CyanogenMod/10.1.3 (android)	ECDHE, SRP, DHE, ECDH, RSA	SHA, MD5	RSA, ECDSA, DSS
Safari (iOS 6.1.3)	ECDHE, ECDH, RSA, DHE	SHA256, SHA, MD5	ECDSA, RSA

Browser	Encryption
Chrome 30.0.1599.69 (MAC,win8)	AES_256_CBC, RC4_128, AES_128_CBC, 3DES_EDE_CBC
Firefox 24 (MAC,win8)	AES_256_CBC, CAMELLIA_256_CBC, RC4_128, AES_128_CBC, CAMELLIA_128_CBC, SEED_CBC, 3DES_EDE_CBC
Safari 6.0.5 (MAC)	AES_256_CBC, AES_128_CBC, RC4_128, 3DES_EDE_CBC
Opera 12.16 (MAC)	AES_256_CBC, AES_128_CBC, RC4_128, 3DES_EDE_CBC
Opera 16 (win8)	AES_256_CBC, AES_128_CBC, RC4_128, 3DES_EDE_CBC
IE 11.0.9431 (win8)	AES_128_CBC, AES_256_CBC, 3DES_EDE_CBC
Chrome 30.0.1599.82 (android)	AES_256_GCM, AES_256_CBC, RC4_128, AES_128_CBC, 3DES_EDE_CBC
Android Browser 4.2.2 (android)	AES_256_CBC, 3DES_EDE_CBC, AES_128_CBC, RC4_128
Dolphin v10 (android)	AES_256_CBC, 3DES_EDE_CBC, AES_128_CBC, RC4_128
CyanogenMod/10.1.3 (android)	AES_256_CBC, 3DES_EDE_CBC, AES_128_CBC, RC4_128
Safari (iOS 6.1.3)	AES_256_CBC, AES_128_CBC, RC4_128, 3DES_EDE_CBC, NULL

B Additional Materials and Proofs for Sections 3–5

B.1 Tolerating Weak Hash Functions

The extent to which we still have to trust MD5 ciphersuites, even if clients are configured to never negotiate a ciphersuite that uses it, is an important practical concern. Assume, for instance, that it is easy to compute MD5 pre-images. An attacker could intercept the client's encrypted *pms* in a session configured to use a strong hash function h and forward it to the same server in a session configured to use MD5. Once the server starts using the master secret derived using MD5, this could reveal information about the key derived using h.

To study the extent to which one-wayness of hash functions in H is sufficient for agile IND-RCCA security we define agile variants of NR-PCA and OW-PCA security: *non-randomizability under plaintext checking oracle and key extraction oracle attacks* (NR-PCA-KEF) and *one-wayness under plaintext checking oracle and key extraction oracle attacks* (OW-PCA-KEF).

Definition 7 (NR-PCA-KEF)**.** *Let* $(\mathsf{keygen}, \mathsf{enc}, \mathsf{dec})$ *be an agile unlabeled KEM,* P *be a set of agility parameters and* $p^\star$ *a public parameter (not necessarily in* P*). Let* KEF *be an agile KEF and* P' *a set of agility parameters for it (the sets* P *and* P' *need not be related in any meaningful way). Consider the game below for an adversary* $\mathcal{A}$ *given oracle access to* PCO *and* EXT*:*

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

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

Oracle $\mathsf{EXT}(p, p', \ell, c) \triangleq$
if $p' \notin P'$ **then return** $\bot$
$k \leftarrow \mathsf{dec}(p, sk, c)$
if $k = \bot$ **then** $k \leftarrow p \| \$$
return $\mathsf{KEF}(p', k, \ell)$

The NR-PCA-KEF advantage of $\mathcal{A}$*,* $\mathbf{Adv}^{\mathsf{NR\text{-}PCA\text{-}KEF}}_{p^\star, P, P'}(\mathcal{A})$ *is defined as the probability that the* $\mathsf{NR\text{-}PCA\text{-}KEF}$ *game returns* true*. The scheme* $(\mathsf{keygen}, \mathsf{enc}, \mathsf{dec})$ *is* $(\epsilon, t, \mathsf{KEF}, P, P')$*-secure against NR-PCA-KEF if the advantage of any adversary* $\mathcal{A}$ *running in time* t *is at most* ϵ*.*

Definition 8 (OW-PCA-KEF)**.** *Let* $(\mathsf{keygen}, \mathsf{enc}, \mathsf{dec})$ *be an agile unlabeled KEM,* P *be a set of agility parameters and* $p^\star$ *a public parameter (not necessarily in* P*). Let* KEF *be an agile KEF and* P' *a set of agility parameters for it (the sets* P *and* P' *need not be related in any meaningful way). Consider the game below for an adversary* $\mathcal{A}$ *given oracle access to* PCO *and* EXT*:*

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

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

Oracle $\mathsf{EXT}(p, p', \ell, c) \triangleq$
if $p' \notin P'$ **then return** $\bot$
$k \leftarrow \mathsf{dec}(p, sk, c)$
if $k = \bot$ **then** $k \leftarrow p \| \$$
return $\mathsf{KEF}(p', k, \ell)$

The OW-PCA-KEF advantage of $\mathcal{A}$, $\mathbf{Adv}^{\mathsf{OW\text{-}PCA\text{-}KEF}}_{p^\star,P,P'}(\mathcal{A})$ is defined as the probability that the OW-PCA-KEF *game returns* true. *The scheme* (keygen, enc, dec) *is* $(\epsilon, t, \mathsf{KEF}, P, P')$*-secure against OW-PCA-KEF if the advantage of any adversary $\mathcal{A}$ running in time t is at most ϵ.*

Theorem 5 (IND-RCCA from NR-PCA-KEF and OW-PCA-KEF)**.** *Let $\mathcal{A}$ be an adversary against the single-challenge RCCA security of the generic TLS ms-KEM with $p^\star = (pv^\star, h^\star)$ assuming $\mathsf{KEF}(p^\star,\cdot,\cdot)$ is a random oracle. Assume $\mathcal{A}$ runs in time $t_\mathcal{A}$, makes at most q_{KEF} queries to the random oracle and at most q_{DEC} queries to the decryption oracle. Then, there exist a OW-PCA-KEF adversary $\mathcal{B}$ and an NR-PCA-KEF adversary $\mathcal{C}$ against the underlying pms-KEM, both running in time $t_\mathcal{A} + O(q_{\mathsf{DEC}} \cdot q_{\mathsf{KEF}})$ such that*

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

where $P' \triangleq \{pv \mid (pv,h) \in P\}$.

The proof is similar to Theorem 3, except that the reductions simulate $\mathsf{KEF}(p^\star,\cdot,\cdot)$ as a random oracle, while queries of the form $\mathsf{KEF}(p,k,\ell)$ with $p \neq p^\star$ are answered using the concrete key extraction function. Decryption queries for $p = (pv,h) \neq p^\star$ are answered using $\mathsf{EXT}(pv,p,\cdot,\cdot)$ and the rest as in Theorem 3.

B.2 Tolerating Unorthodox Long-term Key Usage

In theory we know from [27, 53] how to define the joint security of encryption and signature schemes. Analogously, a combined signature and key derivation scheme consists of algorithms (KeyGen, Sign, Verify, Enc, Dec). We extend the agile notions of EUF-CMA and IND-RCCA security by giving the attacker additional access to a decryption and signing oracle respectively. Both definitions are parameterized by two sets of agility parameters P and P':

Definition 9 (Dual-purpose EUF-CMA)**.** *Let* (KeyGen, Sign, Verify) *be an agile signature scheme, Dec the decryption algorithm of a labeled KEM, $p^\star$ a parameter, and P and P' sets of parameters; and consider the following forgery game:*

```
Game EUF ≜
  pk, sk ← KeyGen();  M, L := ∅
  m', σ ← A^{SIGN,DEC}(pk)
  return m' ∉ M ∧ Verify(p*, pk, m', σ)

Oracle SIGN(p, m) ≜
  if p ∉ P then return ⊥
  M := M ∪ {m}
  return Sign(p, sk, m)

Oracle DEC(p, ℓ, c) ≜
  if ℓ ∈ L ∨ p ∉ P' then return ⊥
  L := L ∪ {ℓ}
  k ← Dec(p, sk, ℓ, c)
  return k
```

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

Definition 10 (Dual-purpose IND-RCCA)**.** *Let* (KeyGen, Enc, Dec) *be an agile labeled KEM,* Sign *a signature algorithm, $p^\star$ a parameter, P and P' sets of parameters; and consider the following game:*

```
Game RCCA ≜
  pk, sk ← KeyGen()
  K, L := ∅
  b ← {0,1}
  b' ← A^{ENC,DEC,SIGN}(pk)
  return (b' = b)

Oracle ENC(ℓ) ≜
  if ℓ ∈ L then return ⊥
  k_0, c ← Enc(p*, pk, ℓ)
  k_1 ← $
  K(ℓ) := K(ℓ) ∪ {k_0, k_1}
  return k_b, c

Oracle DEC(p, ℓ, c) ≜
  if ℓ ∈ L ∨ p ∉ P
  then return ⊥
  L := L ∪ {ℓ}
  k ← Dec(p, sk, ℓ, c)
  if k ∈ K(ℓ) then return ⊥
  return k

Oracle SIGN(p, m) ≜
  if p ∉ P' then return ⊥
  return Sign(p, sk, m)
```

The IND-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 $(\epsilon, t, p^\star, P)$*-dual-purpose-IND-RCCA-n-secure when the advantage of any adversary* $\mathcal{A}$ *running in time* t *and making at most* n *queries to* ENC *is at most* ϵ*.*

By and large, our goal in this work is not to minimize the assumptions that TLS relies upon, but to make them explicit and to provide the correct notation for talking in a constructive manner about them. If one is reluctant to make such assumptions about the primitives employed by TLS—as one indeed should be, then one should only consider only those keys *honest* that have very restricted key usages: only decryption, or only signing, only for use in server authentication or in client authentication, with one common/DNS name, and no other defined/allowed usages.

B.3 Agile PRFs, Key Derivation, and Finished Messages

An *agile PRF* is a family of functions $\mathsf{Prf}(p, \cdot, \cdot)$ parameterized by p. We define the PRF security of Prf for a fixed $p^\star$ as the indistinguishability of $\mathsf{Prf}(p^\star, k, \cdot)$ from a random function, even when given oracle access to $\mathsf{Prf}(p, k, \cdot)$ for $p \in P$, where P is a set of agility parameters.

Definition 11 (PRF security)**.** *Let* Prf *be an agile PRF,* $p^\star$ *a parameter, and* P *a set of parameters, and consider the indistinguishability game:*

Game PR $\triangleq$	**Oracle** $\mathsf{PRF}(p, x) \triangleq$
$k \leftarrow \$;\ Q := \varnothing$	**if** $p \notin P$ **then return** $\perp$
$b \leftarrow \{0,1\}$	**if** $p \neq p^\star \vee \neg b$ **then return** $\mathsf{Prf}(p, k, x)$
$b' \leftarrow \mathcal{A}^{\mathsf{PRF}}()$	**if** $y \notin \mathsf{dom}(Q)$ **then** $Q(x) \leftarrow \$$
return $(b' = b)$	**return** $Q(x)$

The PRF advantage of $\mathcal{A}$, $\mathbf{Adv}^{\mathsf{PRF}}_{p^\star,P}(\mathcal{A})$ *is defined as* $2\Pr[\mathsf{PR} : b' = b] - 1$.

Prf *is an* $(\epsilon, t, p^\star, P)$*-secure PRF when the advantage of any adversary* $\mathcal{A}$ *running in time* t *is at most* ϵ*.*

This definition implicitly requires that the algorithms $\mathsf{Prf}(p, \cdot, \cdot)$ with $p \in P$ do not leak the key k; we assume that the output of Prf is long enough to cover all TLS ciphersuites. This allows us to elide details handled in the miTLS implementation, such as variable output lengths for different agility parameters.

The key-derivation and MAC scheme $D_p = (\mathsf{Kdf}, \mathsf{Mac})$ of TLS is constructed as: $\mathsf{Kdf}(p, ms, \ell, r) \triangleq \lfloor \mathsf{Prf}(p, ms, \texttt{"key expansion"} \,\|\, \bar{\ell}) \rfloor_r$ and $\mathsf{Mac}(p, ms, t, v) \triangleq \lfloor \mathsf{Prf}(p, ms, t \,\|\, v) \rfloor_p$, defined only for $t = \texttt{"client finished"}$ or $t = \texttt{"server finished"}$, where $\bar{\ell}$ is ℓ after swapping the client and server random and $\lfloor . \rfloor_r$ and $\lfloor . \rfloor_p$ truncate to the right length.

We give a definition for KDF & MAC schemes which in addition to a MAC oracle has $\mathsf{COMMIT}(\ell, r)$, $\mathsf{KDF}_{\mathcal{C}}(p, \ell, r)$, and $\mathsf{KDF}_{\mathcal{S}}(p, \ell, r)$ oracles. The definition is analogous to PRF security, except that (p^*, ℓ, r) queries to $\mathsf{KDF}_{\mathcal{C}}$ are only answered with a random value (for $b = 1$) if (ℓ, r) was queried to COMMIT, and queries to $\mathsf{KDF}_{\mathcal{S}}$ are answered with the same value when $\mathsf{KDF}_{\mathcal{C}}$ is queried on (p^*, ℓ, r).

Definition 12 (Joint KDF & MAC security)**.** *Let* $\mathsf{Kdf}(p, \cdot, \cdot)$ *and* $\mathsf{Mac}(p, \cdot, \cdot, \cdot)$ *be agile functions parameterized by* p*,* P *a set of agility parameters, and* $p^\star$ *a public parameter (not necessarily in* P*). Consider the* *following game played between an adversary $\mathcal{A}$ and the challenger:*

$$
\begin{array}{lll}
\textbf{Game } \mathsf{KDF\text{-}MAC} \triangleq & \textbf{Oracle } \mathsf{MAC}(p,t,v) \triangleq & \textbf{Oracle } \mathsf{COMMIT}(\ell,r) \triangleq \\
\quad x \leftarrow \$ & \quad \textbf{if } p \notin P \textbf{ then return } \bot & \quad \textbf{if } \ell \in \mathsf{dom}(S) \textbf{ then return } \bot \\
\quad Q,R,K,S := \varnothing & \quad \textbf{if } p \neq p^\star \vee \neg b \textbf{ then} & \quad S(\ell) := \mathsf{c};\ R(\ell) := r \\
\quad b \leftarrow \{0,1\} & \quad\quad \textbf{return } \mathsf{Mac}(p,x,t,v) & \\
\quad b' \leftarrow \mathcal{A}^{\mathsf{COMMIT},\mathsf{MAC},\mathsf{KDF}_\mathcal{C},\mathsf{KDF}_\mathcal{S}}() & \quad \textbf{if } (p,t,v) \notin \mathsf{dom}(Q) \textbf{ then} & \\
\quad \textbf{return } (b' = b) & \quad\quad Q(p,t,v) \leftarrow \$ & \\
 & \quad \textbf{return } Q(p,t,v) &
\end{array}
$$

$$
\begin{array}{ll}
\textbf{Oracle } \mathsf{KDF}_\mathcal{C}(p,\ell,r) \triangleq & \textbf{Oracle } \mathsf{KDF}_\mathcal{S}(p,\ell,r) \triangleq \\
\quad \textbf{if } p \notin P \textbf{ then return } \bot & \quad \textbf{if } p \notin P \textbf{ then return } \bot \\
\quad k \leftarrow \mathsf{Kdf}(p,x,\ell,r) & \quad k \leftarrow \mathsf{Kdf}(p,x,\ell,r) \\
\quad \textbf{if } p = p^\star \wedge S(\ell) = \mathsf{c} \wedge R(\ell) = r \textbf{ then} & \quad \textbf{if } p = p^\star \wedge S(\ell) = \mathsf{d} \wedge b \textbf{ then} \\
\quad\quad \textbf{if } b \textbf{ then } k \leftarrow \$_r & \quad\quad \textbf{if } r = R(\ell) \textbf{ then} \\
\quad\quad S(\ell) := \mathsf{d};\ K(\ell) := k & \quad\quad\quad k := K(\ell) \textbf{ else } k \leftarrow \$_r \\
\quad \textbf{else } S(\ell) := \mathsf{f} & \quad S(\ell) := \mathsf{f} \\
\quad \textbf{return } k & \quad \textbf{return } k
\end{array}
$$

The challenger maintains a state variable $S(\ell)$ for each label ℓ. The state $S(\ell)$ is initially $\bot$, transitions to c *when the adversary* commits *to use ℓ with a particular parameter r, to* d *once it is used in a $\mathsf{KDF}_\mathcal{C}$ query, and finally to* f *once it is used in a $\mathsf{KDF}_\mathcal{S}$ query. If this order is not respected, the state is set to* f *and the result of any further query with that label is independent of b.* MAC *queries can be freely interleaved, and for $b = 0$ are answered using the shared* key *x.*

The joint KDF & MAC advantage of $\mathcal{A}$, $\mathbf{Adv}^{\mathsf{KDF\text{-}MAC}}_{p^\star,P}(\mathcal{A})$, is $2\Pr[\mathsf{KDF\text{-}MAC} : b' = b] - 1$. We say that Kdf *and* Mac *are jointly $(\epsilon, t, p^\star, P)$-secure if the advantage of any adversary $\mathcal{A}$ running in time t is at most ϵ.*

We easily confirm the following lemma by verifying that Prf is used by KDF and MAC on disjoint domains.

Lemma 2 (KDF & MAC)**.** *If* Prf *is an $(\epsilon, t, p^\star, P)$-secure PRF, then* (Kdf, Mac) *are jointly $(\epsilon, t', p^\star, P)$-secure, where t' is t plus a small cost for multiplexing between different functions.*

From a protocol design viewpoint, more robust, modern constructions such as SP-800-108 additionally hash the target algorithm and key length for the derived key, to ensure that different algorithms always yield (computationally) independent keys. This is however not required by our definition, as it does not idealize keys in case of algorithm mismatch.

Discussion. Agreeing on the parameter r as the key is derived is important for compositional proofs, and in particular to ensure that our model of the handshake fits within our model for the whole TLS protocol. Assume given a generic family of schemes $(\vec{\mathsf{o}}_r(k, \dots))$ whose algorithms are parameterized by a key k. These schemes may provide, for instance, authenticated encryption $(\mathsf{Enc}_r(k,t), \mathsf{Dec}_r(k,c))$, or more advanced LHAE variants, such as those used in the TLS record layer of the miTLS implementation. Suppose their security is expressed using a game of the form $k \leftarrow \$; \mathcal{A}^{\vec{\mathsf{O}}_r(k,\cdot)}$. Then, for each safely-derived key k for algorithm r, relying on the fact that *all* users of k will use that key with (at most) the algorithm r, we can create a shared instance for r and continue the proof with the corresponding game—provided the algorithms denoted by r are secure in isolation. Conversely, if both parties may start using the same fresh key k, or parts of it, with (potentially) different algorithms r_1 and r_2, then we would need a joint, stronger, agile security assumption for these schemes.

B.4 Proof of Theorem 4

Initial hybrids The code of [9] implements cryptographic libraries for signatures, the *ms*-KEM, and key derivation. In addition to being compiled in the concrete way, these libraries can be compiled with an #ideal flag; the resulting code then expresses an idealized functionality, whose stronger properties can be checked and used for automated verification. For example, for any instance with an honest key and a strong algorithm, the ideal implementation of signatures rejects the messages that were not previously signed. Similarly, the ideal code for key encapsulation and key derivation provides fresh random master secrets and record keys. Each idealization step may depend on others. For example, key derivation assumes that the master secret is random; it will thus be idealized only after idealizing key encapsulation. (These dependencies are checked by type-checking.) Like ideal functionalities, idealized libraries can intuitively be understood as implemented by a trusted third party that performs the checks and distributes perfectly random keys to the instances involved. In the MITLS code, we implement them (in code flagged by #ideal) using table lookup with tables only accessible from the MITLS implementation.

MITLS provides multi-key, a.k.a. multi-user [8], variants of the primitives described and proven secure in §2, §3, and §B.3. Let $\alpha_{\mathcal{L}}$, $\mathcal{L} \in \{\mathcal{S}, \mathcal{E}, \mathcal{D}\}$ be library specific strength predicates. Library $\mathcal{L}$ compiled with the #ideal flag set provides ideal functionality for all agility parameters $a_{\mathcal{L}}$ for which $\alpha_{\mathcal{L}}$ holds. Formally, for each $(\epsilon, t, \alpha_{\mathcal{L}})$-secure library, we prove that the implementations compiled with and without the #ideal flag are computationally indistinguishable.

Next, we show how to match these requirements to the definitions and proofs in this paper, relying on hybrid-arguments to deal with multiples instances. Let P_s, P_e, P be algorithm specific agility sets, either defined statically for the worst case, or dynamically updated as part of the experiment, as discussed above.

Lemma 3 (Signature library)**.** *If for all s, p for which $\alpha_{\mathcal{S}}(s,p)$, the signature scheme S_s is $(\epsilon_{s,p}, t_{s,p}, p, P_s)$-secure against EUF-CMA, then the signature library $\mathcal{S}$ is $(\sum_s \sum_p n_s \epsilon_{s,p}, t, \alpha_{\mathcal{S}})$-secure, letting s and p range over all strong algorithmic choices, n_s bound the number of keys generated for algorithm s, and $t_{s,p}$ be at most t plus the maximum cost of the corresponding reductions $\mathcal{B}_{i,j}$.*

PROOF SKETCH: Let $\mathcal{A}$ be an adversary against $\mathcal{S}$. The proof is via a hybrid argument over honest signature public keys for strong algorithms. Assume agility parameters are totally ordered by $<$. Define the hybrid library $\mathcal{S}_{i,j}$ as follows: up to the i-th honest public key and any agility parameter, and for the i-th honest key and $p \leq j$, it behaves as if #ideal is set. For the i-th honest key and agility parameter $p > j$, and for the rest of the honest keys, behaves as if #ideal is not set. Let s be the public key algorithm of the i-th honest signature key. We describe a reduction $\mathcal{B}_{i,j}$ that uses an $\epsilon_{s,j}$ difference in the advantage of $\mathcal{A}$ between two hybrids to break EUF-CMA security. For the i-th public key, the reduction uses the public key from the EUF-CMA game. The reduction uses its oracle SIGN to sign using the corresponding private key.

Until $\mathcal{A}$ produces a forgery for the i-th key and agility parameter j, the reduction $\mathcal{B}_{i,j}$ behaves exactly like hybrid $\mathcal{S}_{i,j-1}$ or $\mathcal{S}_{i,j}$ (respectively hybrid $\mathcal{S}_{i-1,p_{max}}$ or $\mathcal{S}_{i,j}$ at key borders). When $\mathcal{A}$ terminates, $\mathcal{B}_{i,j}$ simply forwards the output of $\mathcal{A}$, and thus succeeds when $\mathcal{A}$ does.

□

The key encapsulation library $\mathcal{E}$ is a multi-scheme and multi-key version of the agile *ms*-KEM defined and constructed in §3. Like Definition 4, the $\mathcal{E}$ library provides a $\mathsf{Commit}(pk, \ell, p)$ function which, when the #ideal flag is set, calls Enc to derive a KEM ciphertext and a master secret k_0 and samples a fake master secret k_1. It stores $(pk, \ell, e, p_{\mathcal{E}}, c_0, k_0, k_1)$ to answer both encryption and decryption queries related to public key pk and label ℓ.

Lemma 4 (Key encapsulation library)**.** *If for all e, p for which $\alpha_{\mathcal{E}}(e,p)$, the key encapsulation scheme E_e is $(\epsilon_{e,p}, t_{e,p}, p, P_e)$-secure against IND-CRCCA, then the key encapsulation library $\mathcal{E}$ is $(\sum_e \sum_p n_e \epsilon_{e,p}, t, \alpha_{\mathcal{E}})$-secure, letting e and p range over all strong algorithms, n_e bound the number of keys generated for algorithm e, and $t_{e,p}$ be at most t plus the maximum cost of the corresponding reductions $\mathcal{B}_{i,j}$.*

Proof sketch: Let $\mathcal{A}$ be an adversary against $\mathcal{E}$. The proof is via a hybrid argument over honest KEM keys for strong algorithms. Assume agility parameters are totally ordered by $<$. Consider hybrid libraries $\mathcal{E}_{i,j}$ defined as follows: up to the i-th honest public key, $\mathcal{E}_{i,j}$ uses KEMs with random master secrets. For the i-th honest key and $p \leq j$ it uses random master secrets in safe instances; for $p > j$ it uses concretely generated master secrets. For the rest of public keys, it uses KEMs with concretely generated master secrets.

Let e be the public key algorithm of i-th honest KEM. We describe a reduction $\mathcal{B}_{i,j}$ that uses an $\epsilon_{e,j}$ difference in the probabilities of $\mathcal{A}$ between two hybrids to break IND-CRCCA security. For the i-th honest public key $\mathcal{B}_{i,j}$ use the public key of the CRCCA game. Upon a call to $\mathsf{Commit}(pk_i, \ell, j)$, call $\mathsf{COMMIT}(\ell)$. Upon a call to Enc for the i-th public key and agility parameter j, call $\mathsf{ENC}(\ell)$ to obtain c and k. For other agility parameters, run the concrete KEM encryption on demand. For the i-th public key, the reduction uses calls to DEC to compute the key returned by the Dec library function. Depending on the bit b of CRCCA, reduction $\mathcal{B}_{i,j}$ behaves exactly like hybrid $\mathcal{E}_{i,j-1}$ or $\mathcal{E}_{i,j}$ (respectively hybrid $\mathcal{E}_{i-1,p_{max}}$ or $\mathcal{E}_{i,j}$ at key borders). $\mathcal{B}_{i,j}$ simply forwards the guess of A.

□

The key derivation and finish MAC library $\mathcal{D}$ is a multi-key (multi-*ms*) version of the agile joint KDF & MAC scheme defined and constructed in §B.3.

Lemma 5 (Key derivation and finish MAC library)**.** *If for all p for which $\alpha_{\mathcal{D}}(p)$ the joint KDF & MAC scheme D_p is (ϵ_p, t_p, p, P)-secure, then the key derivation and finished MAC library $\mathcal{D}$ is $n_{ms}(\sum_p \epsilon_p, t, \alpha_{\mathcal{D}})$-secure, letting p range over all strong algorithms, n_{ms} bound the number of (honest) master secrets, and t_p be at most t plus the maximum cost of the corresponding reductions $\mathcal{B}_{i,j}$.*

Proof sketch: Let $\mathcal{A}$ be an adversary against $\mathcal{D}$. The proof is via a hybrid argument over the safe KDF keys *ms* and their strong algorithms. Assume agility parameters are totally ordered by $<$. Consider hybrid libraries $\mathcal{D}_{i,j}$ defined as follows: up to the i-th master secret, $\mathcal{D}_{i,j}$ randomly samples keys using $\mathsf{KeyGen}_r()$ and produces random MAC tags (idealized output). For the i-th master secret with $p \leq j$ it also provides idealized output; for $p > j$ it uses concretely generated keys and MAC tags. For honest master secrets greater than i, it uses KDFs with concretely generated keys and tags.

We now describe a reduction $\mathcal{B}_{i,j}$ that uses an ϵ_j difference in the advantage of $\mathcal{A}$ between two hybrids to break joint KDF & MAC security. For the i-th master secret, $\mathcal{B}_{i,j}$ uses the KDFMAC game. It calls $\mathsf{COMMIT}(\ell, \mathsf{r})$ when the corresponding Commit function is called in the library. It calls $\mathsf{KDF}_{\mathcal{C}}$ to obtain the keys of client epochs and $\mathsf{KDF}_{\mathcal{S}}$ to obtain the keys of server epochs. The reduction uses calls to MAC to obtain MAC tags for both client and server finished messages. Depending on the bit b of KDFMAC, the reduction behaves exactly like hybrid $\mathcal{D}_{i,j-1}$ or $\mathcal{D}_{i,j}$ (respectively hybrid $\mathcal{D}_{i-1,p_{max}}$ or $\mathcal{D}_{i,j}$ at master secret borders). $\mathcal{B}_{i,j}$ simply forwards the guess of A. □

We are now ready to employ these lemmas in the proof of our main theorem. We look both at full (sessions) and abbreviated handshakes (resumptions) at once, as the proof and the bounds are shared.

Proof Outline. *(1) Uniqueness*. Let n be the total number of epochs. Irrespective of timestamps, the length of the randomness in client and server nonces is 224 bits. The probability that n randomly generated 224 bit values give rise to a collision is approximately $\binom{n}{2}2^{-224}$. This is the worst case as it assumes that the adversary controls half of ℓ and that all of them are of the same role. We thus bound $\mathbf{Adv}^{\mathrm{U}}(\mathcal{A})$ by $n^2 2^{-225}$. This also implies uniqueness both for sessions and resumptions. □

(2) Verified Safety. We need to show that, if there is a peer signature, its public key is honest, and its signing algorithm is strong, then there is a peer session with the same assignments to all peer-exchange variables.

For anonymous peers there is nothing to prove: they do not have public keys and their communication partners cannot verify the safety of their session. Conversely, the servers of static cipher-suites like RSA and static Diffie-Hellman have only static server exchange values: their safety may be independently inferred by the application, e.g. by validating their certificate chains, but this is outside our TLS handshake model. This leaves two cases that require a reduction proof to agile EUF CMA (Definition 9): Clients using ephemeral Diffie-Hellman verify a signature on the server's ephemeral DH contribution.[8] Conversely, servers with authenticated clients verify a signature on the clients transcript up to sending the `ClientKeyExchange` fragment. Since we allow the same signature keys to be used both by clients and servers, we consider both cases at once. The proof involves two games.

- *Game 1* is the original verified safety game, in which $\mathcal{A}$ interacts with the TLS handshake protocol by calling KeyGen, Init, Send, and Control any number of times, in any order.
- *Game 2* is the same as Game 1, except that signature verification is corrected to fail (irrespective of the tag) when the signature scheme is strong, the signing key honest, and yet (1) for client verification on pk_e, there is no server epoch that assigns pk_e to its server-exchange variable; and (2) for server verification of the log till `CertificateVerify` excluded, there is no client epoch with a matching log.

In this final game, we check that the attacker never wins, because

(1) Clients and servers sign only payloads formatted from their local exchange variables and logs.

(2) Client and server signed payloads have disjoint formats, so their respective signatures cannot be confused.

(3) Client-signed payloads are injective in the inputs to the MS computation, so the server will compute and assign the same client-exchange variable.

(4) Server-signed payloads are injective in (i.e., unambiguously determines) the DH KEM, so the client will assign the same server-exchange variable. In this presentation, we do not support both DHE and ECDHE simultaneously, so formally there is no risk of confusing their signed exponentials [47]; otherwise we would require that the honestly-signed payloads for DHE and ECDHE have disjoint formats and that clients in addition to verifying signatures check for these format differences.

As the assumptions in Theorem 4 are sufficient to derive that library $\mathcal{S}$ is $(\epsilon_{\mathcal{S}}, t_{\mathcal{S}}, \alpha_{\mathcal{S}})$-secure, we have that the difference of the advantage of $\mathcal{A}$ in G_1 and G_2 is bounded by $\epsilon_{\mathcal{S}}$. Moreover, because in Game 2 the advantage of $\mathcal{A}$ is zero, we have $\mathbf{Adv}^{\mathrm{S}}(\mathcal{A}) \leq \sum_s \sum_p n_s \epsilon_{s,p}$, as required. □

(To prove Verified safety we only had to consider sessions. Our proof does not rely on the freshness of the nonces. In a more general model, e.g. when the adversary can eventually decrypt KEM ciphertexts, we

[8] Some legacy ciphersuites also support ephemeral variants of RSA key transport; they could be modeled in a similar fashion, but are not supported by miTLS.

would insert an intermediate game between Games 1 and 2 and then rely on the freshness of the verifier's nonce to exclude KEM replay attacks.)

(3) Agile Key Derivation. The proof proceeds using a sequence of games. Let $\Pr[G_i : b' = 1]$ be the probability that $\mathcal{A}$ outputs 1 in Game i.

- *Game 1.* This is the agile key derivation game for $b = 0$.
- *Game 2.* This is the same as Game 1, except that we abort if there are colliding nonces. We bound the probability of aborting by the Uniqueness advantage: $\Pr[G1 : b' = 1] - \Pr[G_2 : b' = 1] \leq \mathbf{Adv}^{\mathrm{U}}(\mathcal{A})$.
- *Game 3.* The same as Game 1 except that we set the #ideal flag in $\mathcal{E}$. As the assumptions in Theorem 4 are sufficient to derive that library $\mathcal{E}$ is $(\epsilon_{\mathcal{E}}, t_{\mathcal{E}}, \alpha_{\mathcal{E}})$-secure, we have that

$$\Pr[G_2 : b' = 1] - \Pr[G_3 : b' = 1] \leq \epsilon_{\mathcal{E}} = \sum_e \sum_p n_e \epsilon_{e,p} \ .$$

- *Game 4.* Same as Game 3, except that we set the #ideal flag in $\mathcal{D}$. This means that we sample fresh keys (in exactly the same way as in the $b = 1$ branch).
  As the assumptions in Theorem 4 are sufficient to derive that library $\mathcal{D}$ is $(\epsilon_{\mathcal{D}}, t_{\mathcal{D}}, \alpha_{\mathcal{D}})$-secure, we have that

$$\Pr[G_3 : b' = 1] - \Pr[G_4 : b' = 1] \leq \epsilon_{\mathcal{D}} = n_{ms} \sum_p \epsilon_p \ .$$

- *Game 5.* Same as Game 4 except that we unset the #ideal flag in $\mathcal{E}$. This means that we revert to generating master secrets for the $b = 0$ branch as in Game 2. Again, as the assumptions in Theorem 4 are sufficient to derive that library $\mathcal{E}$ is $(\epsilon_{\mathcal{E}}, t_{\mathcal{E}}, \alpha_{\mathcal{E}})$-secure, we have that

$$\Pr[G_4 : b' = 1] - \Pr[G_5 : b' = 1] \leq \epsilon_{\mathcal{E}} = \sum_e \sum_p n_e \epsilon_{e,p} \ .$$

- *Game 6.* Same as Game 5, but we revert to allowing collisions on ℓ. We bound the probability of aborting by the Uniqueness advantage: $\Pr[G_5 : b' = 1] - \Pr[G_6 : b' = 1] \leq \mathbf{Adv}^{\mathrm{U}}(\mathcal{A})$.

Game 6 behaves just like the agile key derivation game for $b = 1$, thus

$$\mathbf{Adv}^{\mathrm{K}}(\mathcal{A}) \leq \Pr[G_1 : b' = 1] - \Pr[G_6 : b' = 1]| \leq 2 \cdot \left(\mathbf{Adv}^{\mathrm{U}}(\mathcal{A}) + \sum_e \sum_p n_e \epsilon_{e,p} \right) + n_{ms} \sum_p \epsilon_p \ . \qquad \square$$

(To prove Agile Key Derivation we had to consider sessions and resumptions simultaneously. Only changes in Game 3 and Game 5 do not affect resumptions, as the master secret is reused from the resumed session.)

(4) Agreement. The proof proceeds using a sequence of games.

- *Games 1-4* are the same as the corresponding games for agile key derivation, thus

$$\mathbf{Adv}^{G_1}(\mathcal{A}) - \mathbf{Adv}^{G_4}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{U}}(\mathcal{A}) + \epsilon_{\mathcal{E}} + \epsilon_{\mathcal{D}} \ .$$

As in Game 4 MACs of safe epochs are generated at random,

$$\mathbf{Adv}^{G_4}(\mathcal{A}) \leq 2 \cdot \left(\binom{n}{2} 2^{-\mathsf{min}_p |\mathsf{Mac}_p|} \right) \leq n^2 \cdot 2^{-\mathsf{min}_p |\mathsf{Mac}_p|}$$

by the collision probability of MAC tags.

Recall that the safe renegotiation extension requires that the log includes the MAC of the log of prior epochs which means that we authenticate all assignments up to the current epoch and thus

$$\begin{aligned}\mathbf{Adv}^{\mathrm{I}}(\mathcal{A}) &\leq \epsilon_{\mathcal{E}} + \epsilon_{\mathcal{D}} + n^2 \cdot 2^{-\mathsf{min}_p|\mathsf{Mac}_p|} \\ &\leq \mathbf{Adv}^{\mathrm{U}}(\mathcal{A}) + \sum_e \sum_p n_e \epsilon_{e,p} + n_{ms} \sum_p \epsilon_p + n^2 \cdot 2^{-\mathsf{min}_p|\mathsf{Mac}_p|}\ . \quad \square\end{aligned}$$

(To prove Agreement we had to consider sessions and resumptions simultaneously. Only the changes in Game 3 did not affect resumptions, as the master secret is reused from the resumed session.)

By taking the maximum of these bounds, we conclude

$$\epsilon = \sum_s \sum_p n_s \epsilon_{s,p} + \sum_e \sum_p n_e \epsilon_{e,p} + n_{ms} \sum_p \epsilon_p + n^2 \left(2^{-225} + 2^{-\mathsf{min}_p|\mathsf{Mac}_p|}\right)\ . \quad \square$$

B.5 Additional Handshake Security Properties

Definition 13 (Additional Handshake Games)**.** *Let* Π *be a handshake protocol and* $\mathcal{A}$ *an adversary that interacts with it by calling* KeyGen*,* Init*,* Send*, and* Control *any number of times, in any order. Consider the following security properties:*

(1) ***Forward Secure Verified Safety:*** *To model forward secrecy, give* $\mathcal{A}$ *an additional action* Corrupt *that returns the private key of a long-term key pair and marks the corresponding public key as no longer honest; otherwise the definition is unchanged from verified safety.*
Let $\mathbf{Adv}^{\mathrm{FS}}(\mathcal{A})$ *be the probability that one epoch has the following properties when* $\mathcal{A}$ *terminates:* $\alpha(a) = 1$*; the public key is honest for the signing algorithm indicated by* a*; and the assignment to the the peer exchange value is not honest (i.e. it was not assigned by any peer);*

(2) ***Raw Agile Key Derivation:*** *depending on a random bit* b*, replace the record key assigned in safe epochs with a fresh* k *of maximum length, i.e. as produced by* Prf*, assigning the same value to epochs that have the same identifier* ℓ*, algorithms* $\mathsf{kdf}(a)$ *and exchange variables or resumption identifier.*
Let $\mathbf{Adv}^{\mathrm{R}}(\mathcal{A}) = 2p - 1$ *where* p *is the probability that* $\mathcal{A}$ *returns* b*.*

(3) **Agile Forward Secure Key Derivation:** *give* $\mathcal{A}$ *access to an additional oracle* Corrupt *that returns the private key of a long-term key pair; depending on a random bit* b*, replace the record key assigned in safe ephemeral 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 $\mathbf{Adv}^{\mathrm{F}}(\mathcal{A}) = 2p - 1$ *where* p *is the probability that* $\mathcal{A}$ *returns* b*.*
(Analogously to above, define Raw Forward Secure Key Derivation by not require matching record algorithms r *and replace the keys with fresh random values of maximum key length.)*

Forward Secure Verified Safety. The proof of forward secure verified safety is identical to the proof of verified safety, as it is not affected by the corruption of long-term KEM keys and as nothing needs to be proven about corrupted signature keys.

Forward Secure Agile Key Derivation. The proof of forward secure key derivation is analogous to agile key derivation, except that in Game 3 and Game 5 only ephemeral sessions are idealized while in Game 4 only the keys derived from master secrets generated in ephemeral sessions are idealized. This means that in the proof static KEM keys are treated as dishonest by the $\mathcal{E}$ library.

Table 1: Supported protocol versions, ciphersuites and extensions.

Protocol Versions	Key exchange	Signature	Record encryption	Hash	Extensions
TLS 1.2	RSA	RSA	AES_256_GCM	SHA384	Secure renegotiation
TLS 1.1	DHE	DSA	AES_128_GCM	SHA256	Extended length-hiding
TLS 1.0	DH		AES_256_CBC	SHA	Session hashes
SSL3	DH_anon		AES_128_CBC	MD5	Secure resumption
			3DES_EDE_CBC		
			RC4_128		

Raw (Forward Secure) Key Derivation. The protocol in Figure 1 does not meet the *raw key derivation* property if KDF returns different keys for different record algorithms, as is the case in TLS since keys are cut to the required length. Raw forward security can be recovered by returning constant-size keys. The proof is similar to the proof above, except that the reduction calls Commit for both the client and the server with an a with a constant record algorithm. Note that *Agile Key Derivation* is not sufficient for providing guarantees for *False Start* as it guarantees that the same record keying material will never be used with different record algorithms. Instead, *False Start* requires *Raw Key Derivation* security for the handshake and stronger agile security properties for record algorithms that may share raw keys.

C Verified Reference Implementation of the miTLS Handshake

We refer to Bhargavan et al. [9, §2] for a description of the type-based cryptographic verification method used for miTLS. The full modular structure of the miTLS implementation is depicted in Figure 3 and the protocol features it supports are listed in Table 1. We highlight four aspects of the miTLS handshake implementation and our proofs, before presenting performance results.

C.1 Agility Parameters

The various cryptographic algorithms, protocol versions, and extensions supported by the implementation are defined in the modules: *TLSConstants* and *Extensions*. The module *TLSInfo* specifies agility parameters for various cryptographic constructions and indexes and data structures to represent sessions and connections. Its interface defines a series of predicates that define the strength various algorithms (e.g. *StrongKDF*, *StrongAE*), the honesty of various long-term keys and short-term secrets (e.g. *HonestSig*, *HonestPMS*), and safety for epochs.

C.2 The Handshake API

The application can control the TLS client and server by calling functions in the *TLS* module, which in turn calls the relevant functions in the *Handshake* module. The main functions in this interface are:

val $init$: $rl{:}Role \rightarrow c{:}config \rightarrow (ci{:}CI * s{:}(;ci)state)\{Config(ci,s) = c\ ...\}$
val $authorize$: $r{:}Role \rightarrow si{:}SessionInfo \rightarrow \text{unit}\ \{Authorize(r,si)\}$
val $resume$: $nextSID{:}sessionID \rightarrow c{:}config \rightarrow (ci{:}CI * s{:}(;ci)state)\{\ Config(ci,s) = c\ ...\}$

This interface formally corresponds to the adversary's *Control* interface. The *init* function creates a connection and initiates the first handshake on it. The *authorize* function enables the application to inspect and authorize a peer's certificate (and other session parameters) before the handshake is completed. Once a handshake is completed, the application may send data on the new epoch, but we do not show those record-layer functions here. An application may resume a previous session over a new connection by calling *resume*. Other function (not shown here) allow the application to renegotiate and resume sessions over the same connection.

C.3 Message Formats

After initialization, the *Handshake* module listens to messages from the network, which represent the adversary's *Send* interface. It parses each message and then calls the relevant function to modify the handshake state and adds the message to the log for eventual authentication in the Finished (and CertificateVerify) messages.

The *HandshakeMessages* module constructs and parses handshake messages. Detailed message formats are traditionally ignored in protocol models and cryptographic proofs, but are crucial in TLS to establish *Agreement*, which depends on both the client and server having the same parsed interpretation of their Handshake message logs. To give an example, the first message in the log, ClientHello, has the following format:

```
struct {
        ProtocolVersion client_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suites<2..2^16−2>;
        CompressionMethod compression_methods<1..2^8−1>;
        select (extensions_present) {
            case false: struct {};
            case true: Extension extensions<0..2^16−1>;
        };
    } ClientHello;
```

To ensure that this message can be parsed unambiguously at both client and server, we define a logical function *ClientHelloMsg*(*pv*,*crand*,*sid*,*cs*,*cl*,*ext*) that precisely details this message format. We then prove that the functions in *HandshakeMessages* that generate and parse client hello messages obey this logical specification. For example:

val *clientHelloBytes*: *c*:*config* → *cr*:*random* → *sid*:*sessionID* → *ext*:bytes → *m*:bytes{$B(m)$ =
ClientHelloMsg(*c.maxVer*,*cr*,*sid*,*c.ciphersuites*,*c.compressions*,*ext*)}

Then, we prove that the logical function is injective, so that there is a unique way to parse its components.

theorem !*pv*,*crand*,*sid*,*cs*,*cl*,*ext*,*pv'*,*crand'*,*sid'*,*cs'*,*cl'*,*ext'*.
ClientHelloMsg(*pv*,*crand*,*sid*,*cs*,*cl*,*ext*) = *ClientHelloMsg*(*pv'*,*crand'*,*sid'*,*cs'*,*cl'*,*ext'*) ⇔
($pv = pv' \wedge crand = crand' \wedge \ldots$)

Finally, we extend this injectivity theorem to the full handshake log. Any two equal logs must begin with the same ClientHello message, and hence with the same parameters. More generally, we show that they agree on all the handshake parameters and hence on all the variable assignments in the current epoch.

C.4 State Machine

The bulk of the protocol logic is encoded in the handshake state machine, as depicted in Figure 4. Encoding and verifying such a complex state machine is a challenge—not only does it implement different control flow paths for different key exchanges, different protocol versions, and client authentication modes, it must also be ready to receive messages that trigger any handshake at any time.

In such code, it is easy to make some implementation decisions that end up bypassing security. For example, if a client renegotiates a full handshake with the server, then during this second handshake it may continue to receive data over the connection established from the first handshake. It should accept this data until it receives the new ChangeCipherSpec message, at which point it should stop accepting data until the handshake is complete, since new keys have been committed on but not confirmed. However, many TLS implementations make the mistake of accepting data even in this inconsistent state. The miTLS implementation carefully enforces such state machine invariants.

As a second example, suppose a client has sent its Finished message and is waiting for the server's Finished message. It is tempting for the client to start sending its data already to reduce the latency of the TLS connection. This is the design of the TLS False Start extension, and a similar rationale is used in the TLS NPN NextProtocolMessage. In both cases, if the ciphersuite negotiated is strong enough, the confidentiality of the data being sent seems to be preserved. But cryptographically, it is difficult to justify a design where a client and server may use the same keys with different algorithms. Moreover, we found several conditions where such encrypted data may be sent too optimistically, and may be leaked to a network-based adversary. The miTLS implementation strictly forbids such early data transmission.

We verify that the miTLS state machine preserves its logical invariants; this proof is for a 1,700-line program module and requires the use of an SMT solver. We also verify that the state machine treats all secrets parametrically, as a precondition to the game-based transformations of earlier sections.

C.5 Performance Evaluation

We evaluate the performance of the miTLS implementation, written in F#and linked to the Bouncy Castle C# and the OpenSSL EVP cyrptographic providers, against two popular TLS implementations: OpenSSL 1.0.1e, written in C and using its own aforementioned cryptographic libraries (EVP), and Oracle JSSE 1.7, written in Java and using the SunJSSE cryptographic provider.

We tested clients and servers for each implementation against one another, running on the same host to minimize network effects. Figure 5 reports our results for different clients and ciphersuites with OpenSSL as server. We measured (1) the number of handshakes completed per second; and (2) the average throughput provided on the transfer of a 400 MB random data file.

At first glance, when comparing to OpenSSL, these results highlight that the miTLS reference implementation has been designed primarily for modular verification, and has not been optimized for speed. For example, all buffers are implemented using plain functional byte arrays which involve a lot of dynamic allocation and copying as record fragments are processed. However, when compared to VM-based languages, the slow-down is less prominent (order of magnitude of 2 for JSSE), and we consistently outperform the rudimentary TLS client distributed with Bouncy Castle. Moreover, when changing the miTLS crypto provider from BouncyCastle to OpenSSL EVP, one can notice that throughput is then 1.5 faster in the miTLS implementation than in the JSSE case.

Client		**Server**
$\ell_{\mathcal{C}} \leftarrow \$;\ a_{\mathcal{C}} := cfg_{\mathcal{C}}.a_{\mathcal{C}}$	——— `ClientHello`$[\ell_{\mathcal{C}}, a_{\mathcal{C}}]$ ——→	$\ell_{\mathcal{S}} \leftarrow \$;\ \ell := \ell_{\mathcal{C}} \Vert \ell_{\mathcal{S}};\ sid \leftarrow \$$ $cert_{\mathcal{S}} := cfg_{\mathcal{S}}.cert;\ cert_{\mathcal{C}} := \bot$ $pk := \mathsf{pk}(cert_{\mathcal{S}})$ $sk :=$ lookup sk using pk
$\ell := \ell_{\mathcal{C}} \Vert \ell_{\mathcal{S}};\ a := \mathsf{alg}_{\mathcal{C}}(cfg_{\mathcal{C}}, a_{\mathcal{S}})$ $pk := \mathsf{pk}(cert_{\mathcal{S}})$ $c, ms \leftarrow \mathcal{E}.\mathsf{Enc}(e, p_{\mathcal{E}}, pk, \ell)$	`ServerHello`$[\ell_{\mathcal{S}}, a_{\mathcal{S}}, sid]$ ←— `ServerCertificate`$[cert_{\mathcal{S}}]$ —— `ServerHelloDone`	$a, a_{\mathcal{S}} := \mathsf{alg}_{\mathcal{S}}(cfg_{\mathcal{S}}, a_{\mathcal{C}});\ ex_{\mathcal{S}} := pk$ $\mathcal{E}.\mathsf{Commit}(e, p_{\mathcal{E}}, pk, \ell)$ $\mathcal{D}.\mathsf{Commit}(r, p_{\mathcal{D}}, \ell)$
$ex_{\mathcal{S}} := pk;\ ex_{\mathcal{C}} := (pk, ms)$ $k := \mathcal{D}.\mathsf{KDF}(p_{\mathcal{D}}, ms, \ell, r)$ $log_{\mathcal{C}} := \langle$all prior epoch messages$\rangle$	——— `ClientKeyExchange`$[c]$ ——→	$ms \leftarrow \mathcal{E}.\mathsf{Dec}(e, p_{\mathcal{E}}, sk, \ell, c)$ $ex_{\mathcal{C}} := (pk, ms)$ $log_{\mathcal{C}} := \langle$all prior epoch messages$\rangle$
$tag_{\mathcal{C}} := \mathcal{D}.\mathsf{MAC}(p_{\mathcal{D}}, ms, \text{“C”}, log_{\mathcal{C}})$	——— `ClientFinished`$[tag_{\mathcal{C}}]$ ——→	$tag_{\mathcal{C}} \stackrel{?}{=} \mathcal{D}.\mathsf{MAC}(p_{\mathcal{D}}, ms, \text{“C”}, log_{\mathcal{C}})$ $k := \mathcal{D}.\mathsf{KDF}(p_{\mathcal{D}}, ms, \ell, r)$ $log_{\mathcal{S}} := \langle$all prior epoch messages$\rangle$ $tag_{\mathcal{S}} := \mathcal{D}.\mathsf{MAC}(p_{\mathcal{D}}, ms, \text{“S”}, log_{\mathcal{S}})$
$log_{\mathcal{S}} := \langle$all prior epoch messages$\rangle$ $tag_{\mathcal{S}} \stackrel{?}{=} \mathcal{D}.\mathsf{MAC}(p_{\mathcal{D}}, ms, \text{“S”}, log_{\mathcal{S}})$ $complete := 1$	←—— `ServerFinished`$[tag_{\mathcal{S}}]$ ———	$complete := 1;$ store (ℓ, sid, ms)
···········	*Client resumes session (ℓ, sid, ms) using configuration $a_{\mathcal{C}}$ and previous epoch's $tag_{\mathcal{C}}$, $tag_{\mathcal{S}}$*	···········
$\ell_{\mathcal{C}} \leftarrow \$;\ \ell_{\text{session}} := \ell$	—— `ClientHello`$[\ell_{\mathcal{C}}, a_{\mathcal{C}}, sid, tag_{\mathcal{C}}]$ ——→	lookup $(\ell_{\text{session}}, ms, tag_{\mathcal{S}})$ using sid $\ell_{\mathcal{S}} \leftarrow \$;\ \ell_{\text{session}} := \ell_{\text{session}}$
$\ell := \ell_{\mathcal{C}} \Vert \ell_{\mathcal{S}}$ $k := \mathcal{D}.\mathsf{KDF}(p_{\mathcal{D}}, ms, \ell, r)$ $log_{\mathcal{S}} := \langle$all prior epoch messages$\rangle$	← `ServerHello`$[\ell_{\mathcal{S}}, a_{\mathcal{S}}, sid, tag_{\mathcal{C}}, tag_{\mathcal{S}}]$ —	$\ell := \ell_{\mathcal{C}} \Vert \ell_{\mathcal{S}}$ $k := \mathcal{D}.\mathsf{KDF}(p_{\mathcal{D}}, ms, \ell, r)$ $log_{\mathcal{S}} := \langle$all prior epoch messages$\rangle$ $tag_{\mathcal{S}} := \mathcal{D}.\mathsf{MAC}(p_{\mathcal{D}}, ms, \text{“S”}, log_{\mathcal{S}})$
$tag_{\mathcal{S}} \stackrel{?}{=} \mathcal{D}.\mathsf{MAC}(p_{\mathcal{D}}, ms, \text{“S”}, log_{\mathcal{S}})$ $log_{\mathcal{C}} := \langle$all prior epoch messages$\rangle$	←—— `ServerFinished`$[tag_{\mathcal{S}}]$ ———	$log_{\mathcal{C}} := \langle$all prior epoch messages$\rangle$
$tag_{\mathcal{C}} := \mathcal{D}.\mathsf{MAC}(p_{\mathcal{D}}, ms, \text{“C”}, log_{\mathcal{C}})$ $complete := 1$	——— `ClientFinished`$[tag_{\mathcal{C}}]$ ——→	$tag_{\mathcal{C}} \stackrel{?}{=} \mathcal{D}.\mathsf{MAC}(p_{\mathcal{D}}, ms, \text{“C”}, log_{\mathcal{C}})$ $complete := 1$

We show two epochs on the same connection: the first handshake establishes a session without client authentication using non-ephemeral (RSA) keys; the second handshake resumes the session. The protocol uses libraries for signatures ($\mathcal{S}$), KEMs ($\mathcal{E}$) and KDF-MAC ($\mathcal{D}$). (1) Failed checks $\stackrel{?}{=}$ stop the instance; (2) We use $:=$ for assigning epoch variables and assume variables exchanged in messages are implicitly assigned. For instance, the client assigns ℓ_C before sending the first message, and the server assigns ℓ_C and a_C after parsing it. (3) We omit the extraction of the negotiated algorithms $e, p_{\mathcal{E}}, s, p_{\mathcal{S}}, p_{\mathcal{D}}, r$ from a. For instance, we write r for $\mathsf{record}(a)$. (4) We omit `ChangeCipherSpec` messages. (5) $\langle$all prior epoch messages$\rangle$ means the concatenation of all messages sent and received so far, starting from the latest `ClientHello` message.

Figure 1: Abstract model of the TLS handshake protocol (Static Handshake; Resumption)

Client		Server
$\ell_\mathcal{C} \leftarrow \$;\; a_\mathcal{C} := cfg_\mathcal{C}.a_\mathcal{C}$	$\longrightarrow$ `ClientHello`$[\ell_\mathcal{C}, a_\mathcal{C}]$ $\longrightarrow$	$\ell_\mathcal{S} \leftarrow \$;\; \ell := \ell_\mathcal{C} \,\|\, \ell_\mathcal{S};\; sid \leftarrow \$$ $cert_\mathcal{S} := cfg_\mathcal{S}.cert;\; cert_\mathcal{C} := \bot$ $pk := \mathsf{pk}(cert_\mathcal{S})$ $sk :=$ lookup sk using pk $a, a_\mathcal{S} := \mathsf{alg}_\mathcal{S}(cfg_\mathcal{S}, a_\mathcal{C})$ $sk_e, pk_e \leftarrow \mathcal{E}.\mathsf{KeyGen}(e)$ $\sigma \leftarrow \mathcal{S}.\mathsf{Sign}(s, p_\mathcal{S}, sk, pk_e)$ $\mathcal{E}.\mathsf{Commit}(e, p_\mathcal{E}, pk_e, \ell)$ $\mathcal{D}.\mathsf{Commit}(r, p_\mathcal{D}, \ell)$ $ex_\mathcal{S} := pk_e$
$\ell := \ell_\mathcal{C} \,\|\, \ell_\mathcal{S};\; a := \mathsf{alg}_\mathcal{C}(cfg_\mathcal{C}, a_\mathcal{S})$ $\mathcal{S}.\mathsf{Verify}(s, p_\mathcal{S}, pk, pk_e, \sigma) \stackrel{?}{=} 1$ $c, ms \leftarrow \mathcal{E}.\mathsf{Enc}(e, p_\mathcal{E}, pk_e, \ell)$ $ex_\mathcal{S} := pk_e;\; ex_\mathcal{C} := (pk_e, ms)$ $k := \mathcal{D}.\mathsf{KDF}(p_\mathcal{D}, ms, \ell, r)$ $log_\mathcal{C} := \langle$all prior epoch messages$\rangle$ $tag_\mathcal{C} := \mathcal{D}.\mathsf{MAC}(p_\mathcal{D}, ms, \text{“C”}, log_\mathcal{C})$	$\longleftarrow$ `ServerHello`$[\ell_\mathcal{S}, a_\mathcal{S}, sid]$ `ServerCertificate`$[cert_\mathcal{S}]$ `ServerKeyExchange`$[pk_e, \sigma]$ `ServerHelloDone` $\longleftarrow$ $\longrightarrow$ `ClientKeyExchange`$[c]$ $\longrightarrow$ $\longrightarrow$ `ClientFinished`$[tag_\mathcal{C}]$ $\longrightarrow$	$ms \leftarrow \mathcal{E}.\mathsf{Dec}(e, p_\mathcal{E}, sk_e, \ell, c)$ $log_\mathcal{C} := \langle$all prior epoch messages$\rangle$ $ex_\mathcal{C} := (pk_e, ms)$ $tag_\mathcal{C} \stackrel{?}{=} \mathcal{D}.\mathsf{MAC}(p_\mathcal{D}, ms, \text{“C”}, log_\mathcal{C})$ $k := \mathcal{D}.\mathsf{KDF}(p_\mathcal{D}, ms, \ell, r)$ $log_\mathcal{S} := \langle$all prior epoch messages$\rangle$ $tag_\mathcal{S} := \mathcal{D}.\mathsf{MAC}(p_\mathcal{D}, ms, \text{“S”}, log_\mathcal{S})$
$log_\mathcal{S} := \langle$all prior epoch messages$\rangle$ $tag_\mathcal{S} \stackrel{?}{=} \mathcal{D}.\mathsf{MAC}(p_\mathcal{D}, ms, \text{“S”}, log_\mathcal{S})$ $complete := 1$	$\longleftarrow$ `ServerFinished`$[tag_\mathcal{S}]$ $\longleftarrow$	$complete := 1;$ store (ℓ, sid, ms)

Figure 2: Abstract model of the TLS handshake protocol for ephemeral sessions

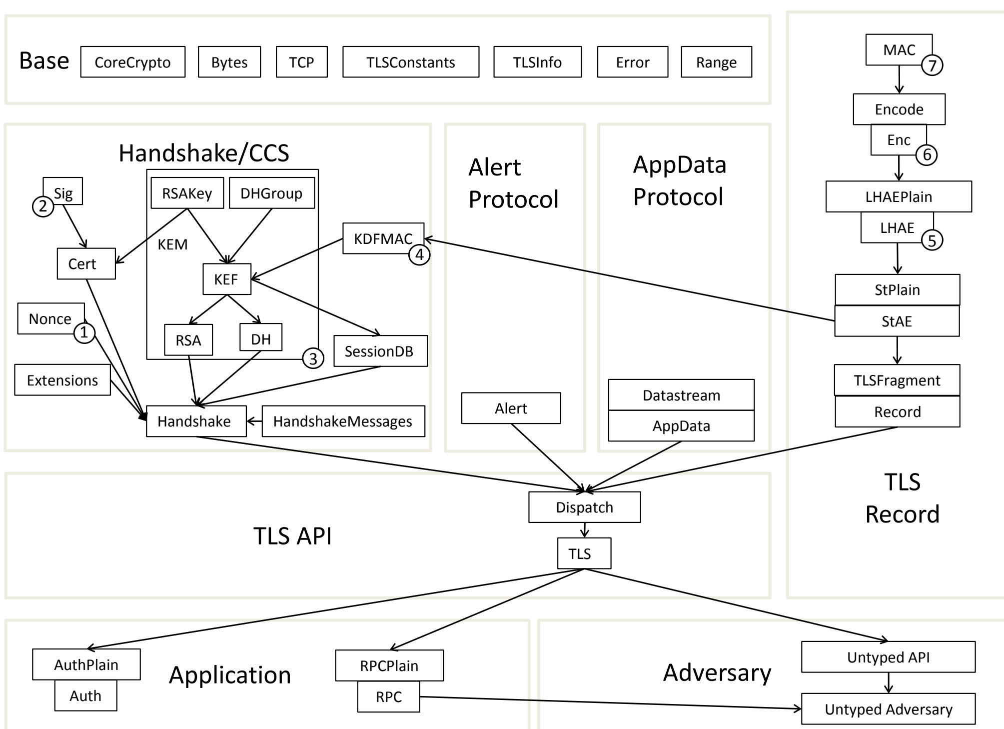

Figure 3: Modular structure of miTLS, and main sequence of game for its security proof.

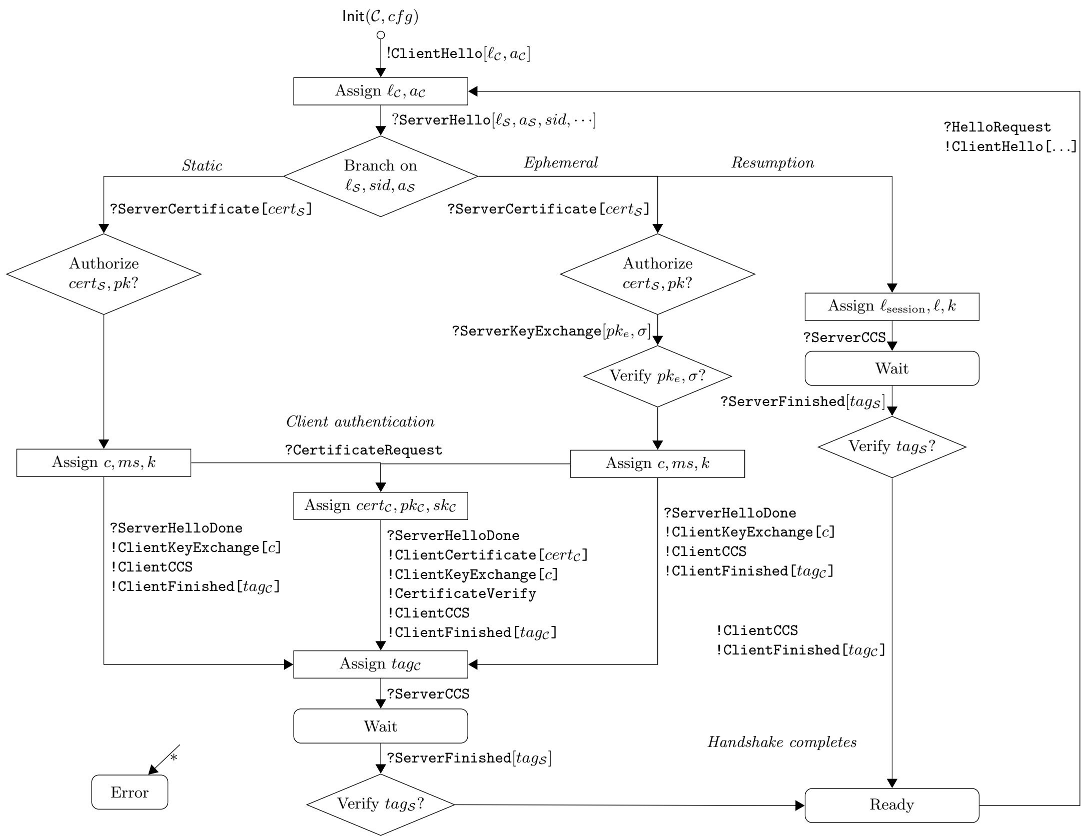

Figure 4: State machine of the client handshake module.

Ciphersuite			F#(BC)		F#(EVP)		OpenSSL		Oracle JSSE	
KEX	Enc	MAC	HS/s	MiB/s	HS/s	MiB/s	HS/s	MiB/s	HS/s	MiB/s
RSA	RC4	MD5	268.22	43.44	273.81	89.54	1257.50	255.99	410.55	64.59
RSA	RC4	SHA	272.32	38.13	270.84	84.76	1214.58	216.20	419.67	59.47
RSA	3DES	SHA	259.86	8.54	272.32	18.82	1147.40	22.12	383.58	10.47
RSA	AES128	SHA	266.23	22.84	269.96	50.10	1121.55	261.74	406.55	58.84
RSA	AES128	SHA256	268.80	19.37	271.13	43.12	1121.56	122.36	401.56	47.87
RSA	AES256	SHA	261.77	20.11	271.13	41.21	1185.66	221.06	-	-
RSA	AES256	SHA256	257.45	17.39	270.84	35.94	1087.29	111.88	-	-
DHE	3DES	SHA	20.83	8.46	20.96	18.32	336.92	22.19	-	-
DHE	AES128	SHA	21.02	22.69	20.85	47.72	343.43	277.64	-	-
DHE	AES128	SHA256	20.94	19.16	20.84	43.46	338.76	123.19	-	-
DHE	AES256	SHA	20.56	20.12	20.95	40.04	344.86	246.14	-	-
DHE	AES256	SHA256	21.11	17.62	20.79	35.69	339.22	113.37	-	-

Figure 5: Performance benchmarks (OpenSSL 1.0.1e as server).