Layered Symbolic Security Analysis in DY*

Karthikeyan Bhargavan[1], Abhishek Bichhawat[2], Pedram Hosseyni[3], Ralf Küsters[3], Klaas Pruiksma[3], Guido Schmitz[4], Clara Waldmann[3], and Tim Würtele[3]

[1] INRIA Paris, France karthikeyan.bhargavan@inria.fr
[2] IIT Gandhinagar, India abhishek.b@iitgn.ac.in
[3] University of Stuttgart, Germany {pedram.hosseyni, ralf.kuesters, klaas.pruiksma, clara.waldmann, tim.wuertele}@sec.uni-stuttgart.de
[4] Royal Holloway, University of London, UK guido.schmitz@rhul.ac.uk

Abstract. While cryptographic protocols are often analyzed in isolation, they are typically deployed within a stack of protocols, where each layer relies on the security guarantees provided by the protocol layer below it, and in turn provides its own security functionality to the layer above. Formally analyzing the whole stack in one go is infeasible even for semi-automated verification tools, and impossible for pen-and-paper proofs. The DY* protocol verification framework offers a modular and scalable technique that can reason about large protocols, specified as a set of F* modules. However, it does not support the *compositional* verification of layered protocols since it treats the global security invariants monolithically. In this paper, we extend DY* with a new methodology that allows analysts to modularly analyze each layer in a way that compose to provide security for a protocol stack. Importantly, our technique allows a layer to be replaced by another implementation, without affecting the proofs of other layers. We demonstrate this methodology on two case studies. We also present a verified library of generic authenticated and confidential communication patterns that can be used in future protocol analyses and is of independent interest.

1 Introduction

Modern Web applications combine a variety of cryptographic mechanisms and protocols to achieve their security goals. For example, to log in to a banking website or code repository, a user typically first enters a username and password over HTTPS. The server may then ask for a second-factor authentication via an independent secure channel with the user's phone. Only when both authentication mechanisms succeed is the user allowed to access any sensitive resource. Each such security mechanism may in turn rely on a whole stack of cryptographic protocols underneath it, each with its own security assumptions and guarantees.

Consider the password-based login mechanism, where the user sends a username and secret password to a website over the HTTPS protocol, which implements a confidential request-response communication pattern between an unauthenticated client and authenticated server. The HTTPS exchange is encoded

within the duplex encrypted data streams provided by the Record sub-protocol of Transport Layer Security (TLS); the keys encrypting these streams are set up by an authenticated key exchange implemented by the TLS Handshake sub-protocol. TLS itself relies on the X.509 public key infrastructure (PKI) for server authentication, a trusted cryptographic library, and an untrusted TCP/IP networking stack for communication.

Consequently, the security and functionality of the password-based login mechanism relies on the correct design and implementation of the stack of protocols depicted on the right. Each layer depends on the security guarantees provided by the layer below and offers new functionality and guarantees to the layer above. The protocol at each layer may well be secure in isolation, but if it is used incorrectly by the layers above it, or if there is any secret value or state shared between two layers, the composite stack may well be insecure. For example, the Triple Handshake attacks [12] demonstrated how three different key exchange protocols that are secure on their own break when composed together. Hence, it is important to analyze the stack as a whole, proving security for the green layers, under precise security assumptions on the crypto, treating the untrusted network as controlled by the adversary.

One option would be to model all the green layers together and prove them secure in a single proof, but this effort can quickly become too large and untenable for pen-and-paper proofs and even automated protocol verification tools. The problem is that although many protocol analysis approaches are effective on small protocols, they are not modular, compositional, or scalable enough to analyze large and complex protocol stacks.

We say that a protocol specification methodology is *modular* when each protocol can be modeled in its own module(s) with a succinct interface that describes its assumptions, functionality, and security guarantees. Further, we say that a protocol analysis framework is *compositional* if it allows different protocols to also be verified independently and then composed without needing to redo the analysis. Finally, we say that a protocol analysis tool is *scalable* if the verification time and effort grows proportionately with the size and complexity of the protocol. We believe that all these three properties are needed to cleanly model and feasibly analyze stacks of layered real-world protocols.

Automated whole-protocol analysis tools like ProVerif [14] and Tamarin [30] work well for small-to-medium protocols, but suffer from not having these three properties. Indeed, it can take hours to analyze a monolithic model of TLS 1.3 using these tools [19,9], without even considering the PKI or the application. Recognizing this drawback, a line of work on symbolic protocol composition studies conditions under which protocol proofs built with such tools can be composed (see e.g. [17,26]). Computational cryptographic provers like EasyCrypt [3], SSProve [1], CryptoVerif [13] model cryptography more precisely but are less ef-

fective than symbolic tools and have only been applied to constructions and small protocols. For these tools, composability is even more important to enable the analysis of large protocols by breaking them into sub-protocols.

In this work, we adopt the type-based machine-checked protocol analysis methodology of DY⋆ [5], which natively supports modular specification and enables proofs that are scalable, since proofs can be type-checked in time linear in the size of the protocol. We observe, however, that the DY⋆ framework is not compositional in that it requires the security invariants for all protocols in a stack to first be specified together, and then each protocol can be independently analyzed with respect to these monolithic security predicates. Changing any protocol layer requires the full stack to be verified again.

Contribution. We design and implement an extension to DY⋆ that enables compositional protocol verification. We use this extension to develop verified implementations of several generic layers, including PKI, TLS, and a library of communication patterns that includes HTTPS-style request-response exchanges. We use these verified libraries to build and analyze protocol stacks for two case studies.[5] We show how each layer can be verified independently and safely composed. We also show how one implementation of a layer can be replaced by another, without re-verifying all other layers. We believe our extension to DY⋆ to be the first symbolic protocol verification methodology that applies to executable protocols and allows for mechanized analysis in a modular, scalable, and compositional way, thereby producing machine-checked proofs.

Structure of the paper. We first briefly recall the DY⋆ framework. We then, in Section 3, present the two mentioned simple case studies, which we use as running examples through the paper. We outline our general approach of layered analysis in DY⋆ in Section 4, with instantiations for a generic PKI layer and a communication layer, built on top of the PKI layer, presented in Section 5. The analysis of our case studies based on the latter two layers is given in Section 6. Related work is discussed in Section 7. Section 8 concludes.

2 The DY⋆ Framework

DY⋆ is a framework for symbolic security analysis of protocol code written in the F⋆ [35] programming language. DY⋆ has been successfully used to verify a variety of cryptographic protocols, including classic protocols like Needham-Schroeder-Lowe and ISO [7], ratcheted key exchange protocols like Signal [5], modern standards like ACME [6], secure channel frameworks like Noise [27], and group protocols like TreeSync [36]. Proofs in DY⋆ are not fully automated and require manual annotations, but in return, DY⋆ offers many advantages over fully automated symbolic analysis frameworks like ProVerif and Tamarin.

First, DY⋆ proofs have access to the full F⋆ proof assistant, and hence can handle arbitrary recursion in protocols using inductive proofs, unlike Tamarin and ProVerif, which only have limited support for induction. Second, DY⋆ supports

[5] Code for all of these implementations can be found in [8].

executable protocol specifications that can be tested to simulate full protocol runs and attacks. Third, DY⋆ uses a type-based proof methodology that scales linearly in the size of the protocol, since every protocol function is analyzed independently. While automated tools are more effective and convenient than DY⋆ for small protocols, they tend to blow up on large protocols like ACME, Signal, and Noise, which is where DY⋆ starts to shine.

In the following, we briefly describe DY⋆ focusing on the aspects that are relevant for the rest of the paper. We refer to [5] for details on the design of DY⋆ and to [7] for a tutorial-style introduction to this framework.

Trace-based Semantics. DY⋆ explicitly encodes the global run-time semantics of distributed protocol executions in terms of a *global trace* and the symbolic security analysis is proved sound with respect to this semantics *within* the verification framework itself.

DY⋆ models the global interleaved execution of a set of protocol participants (or *principals*) as a trace of observable protocol actions (or *entries*). As a principal executes a role in some run of a protocol, it can send and receive messages, generate random values, log security events, and store and retrieve its state (consisting of sessions), and each of these operations either reads from or extends the global trace. The protocol code for each principal cannot directly read from or write to the trace, but instead must use a typed trace API that enforces an append-only discipline on the global trace.

Symbolic Cryptographic Library. DY⋆ also provides a library for the manipulation of bytes. The interface of this library treats bytes abstractly and provides functions for creating constants, concatenating and splitting bytes, and applying various cryptographic primitives such as public-key encryption and signatures, symmetric encryption and message authentication codes, hashing, Diffie-Hellman, and key derivation, which are treated as black-boxes.

The library interface also provides a series of lemmas relating to these functions that effectively form an equational theory, stating, for example, that decryption is an inverse of encryption, or that splitting concatenated bytes returns its components, or that signature verification always succeeds on a validly generated signature. Bytes can only be manipulated by using the functions of this cryptographic API. This ensures that all byte manipulations adhere to the equational theory. For example, signing keys cannot be extracted from a signature and hash functions cannot be inverted, in particular by the attacker.

Dolev-Yao Adversary. The standard attacker model captured by DY⋆ is the symbolic Dolev-Yao active network attacker [21]. This adversary is modeled as an (arbitrary) F⋆ program that is given full access to the cryptographic API and limited access to the global trace API. That is, it can call functions to generate its own random values, send a message from any principal to any principal, and read any message from the trace. Notably, it cannot read any random values or logged security events from the trace, and *a priori* it cannot read the session states stored by any principal. However, the attacker is given a special function that it can call at any time to compromise other principals' states (fully or partly), which marks the respective state as compromised in the

trace and unlocks access to its contents. DY⋆ defines a predicete that captures the knowledge that the adversary can possibly gain at any point in a trace, and we can use this predicate to reason about fine-grained confidentiality guarantees.

Symbolic Execution and Testing. The code for protocol models in DY⋆ can be executed symbolically to obtain traces that can be printed and inspected for debugging. This feature is invaluable to test the model and ensure that it behaves as expected. For example, we can ensure that there isn't a bug in the protocol code that prevents protocol runs from finishing, or we can write example attacker code and test potential attacks against our protocol.

Authentication and Confidentiality Goals. The security goals of a protocol are stated as predicates over all reachable global traces. The trace predicate has full visibility over all entries in the global trace, including sent messages, logged events, and states stored at any principal. To specify an authentication goal, we typically state that certain events must be recorded in a certain order with matching parameters (e.g., when principal B accepts a session with A, then A needs to have initiated this session). To specify confidentiality, we state conditions on the attacker's knowledge at specific points in the trace.

Proof Methodology. The main proof technique in DY⋆ is to establish an invariant over all reachable traces that capture relevant aspects of the modeled protocol and prove that this invariant implies the desired goals. In particular, we need to prove that all functions that can modify the trace, either on behalf of honest protocol code or the attacker, preserve the invariant. To this end, DY⋆ offers a modular proof methodology, where programmers only need to define and prove local protocol-specific state invariants and security goals, and the framework completes the proof by filling in generic security invariants that are proved once-and-for-all for all protocols.

DY⋆ defines a library of *labeled* APIs that enforce a labeling discipline on the usage of cryptography to simplify reasoning about secrecy. The labels explicitly capture the intended set of principals that may know certain bytes, and the labeled APIs enforce that only this set of principals can access the bytes. This library defines a computational effect LCrypto that enforces a global trace invariant called valid_trace. The labeled APIs have valid_trace as both pre- and post-condition for all functions by using the LCrypto effect. The global trace invariant consists of several components, some generic invariants and some predicates that have to be defined for each protocol.

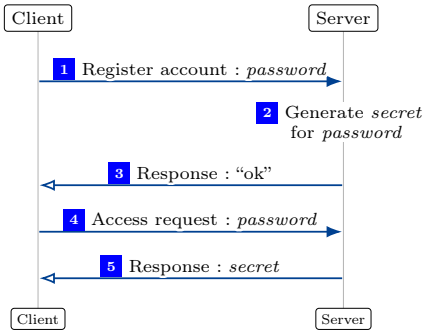
Protocol-Specific Predicates. For each protocol, we specify predicates on the usage of cryptographic functions, pre-conditions for logged events, and invariants on the session states stored by protocol participants. The predicates on the usage of cryptographic functions restrict the application of cryptographic functions to certain messages and keys. For example, the usage predicate for public key encryption (can_pke_enc) may state that honest principals encrypt only messages of a certain form, if certain events have occurred on the global trace, or nonces have a certain label, which in turn gives other honest principals decrypting such messages these guarantees. We note that the attacker/dishonest principals are not restricted in any way.

3 Motivating Examples

In the rest of this paper, we use two high-level security protocols to illustrate our key concepts and our compositional verification methodology. These case studies do not themselves use much cryptography, but they rely on lower-layer cryptographic protocols to provide various kinds of secure channels. Consequently, the analysis of these examples should depend only on the guarantees of the underlying channels but not on the details of how these channels are implemented.

Basic Authentication (BA). The first example, depicted below, is a *basic authentication* protocol, inspired by the Basic HTTP Authentication scheme [32].

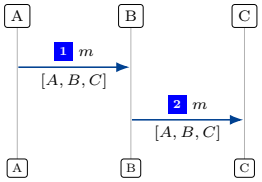
A client can send two different kinds of requests to a server. First, a request to register an account at the server, containing a password (Step 1). When receiving such a request, the server generates a long-term secret (essentially a resource) and stores the secret along with the password (Step 2). The client can send a second type of request (access request) to retrieve the long-term secret, which needs to include the password used for account generation (Step 4). Upon receiving such a request, the server checks whether an account identified by the password exists and returns the corresponding long-term secret (Step 5).



The security guarantee we want to show for this example is that if an honest (i.e. uncorrupted) Client and Server communicate via a server-authenticated confidential channel (like TLS), then the long-term secret stays confidential.

Source Routing (SR). Our second example is a simple *source routing* protocol where a message is to be sent along a pre-specified path of participants.

On the right, we show the protocol for three participants where the message m should take the path $[A, B, C]$. (Note that the protocol itself works for paths of any length.) A initiates the flow by sending the message and the planned path to B, the next participant on the path. B processes the message and sends it on to the next participant C. Once C receives the message the protocol ends.



The security guarantees of the source routing example depend on the types of channels that are used. For example, if all principals send the messages over authenticated channels, then we would like to show that the message indeed took the specified path, as long as none of the participants on the path gets corrupted. Similarly, we would like to prove confidentiality guarantees for the message if the channels are also confidential.

4 Layered Symbolic Protocol Analysis

As mentioned in Section 2, DY* enables the modular and scalable analysis of protocols by relying on the expressiveness of F*, like inductive reasoning and type-based proofs. However, there remain some limitations that make proofs of large protocols in DY* difficult and fragile. For example, consider the BA example described above. The security of this protocol relies on a server-authenticated confidential request-response channel between a client and a server. In practice, this channel is implemented by HTTPS, which in turn relies on TLS, the X.509 PKI, and a crypto library. To fully verify the security of this protocol, we have to model all these layers. Doing so in a monolithic proof framework like ProVerif or Tamarin is infeasible, both due to the effort involved and the verification time.

In DY*, we can rely on the modularity provided by F* to put the modeling *code* for each layer into a separate module and verify them separately. However, even if the code for different layers is independent, the *predicates* that are used in the security proof are shared between all layers. Consequently, we have to globally define the state invariants, event preconditions, the predicates for cryptographic primitives, such as encryption, signatures, and MACs, all in once place. If any two layers use the same cryptographic construction, e.g. public key encryption, we have to instantiate the predicate for public-key encryption, `can_pke_enc` (see also Section 2), in a way that both layers still typecheck, which in turn requires a proof that the uses of this predicate in the two layers are disjoint, i.e. they do not conflict with each other. These kinds of proofs are not just unpleasant, but also non-compositional. If we wanted to change (say) the channel implementation from TLS to some other protocol with the same guarantees, we would have to change the predicates and reverify the full stack.

In short, DY* offers a scalable proof methodology and a modular specification technique, but does not support composable proofs for sub-protocols or protocol layers. Even to prove simple protocols like our case studies, the analyst must read, edit, and verify a set of global predicates that include details of lower-level protocols and higher-layer applications that they may not be familiar with.

The layered predicates approach. In this paper, we propose a new methodology for the *layered analysis* of protocols modeled in DY*. Our key insight is to separately model both the code and the predicates of each protocol layer in its own module, and specify rules on how these predicates are composed. Essentially, each lower layer takes the higher-layer predicates as opaque parameters and incorporates them in its analysis. Consequently, the security proof of the lower layer is done only once for *all* instantiations of the higher-layer predicates. Unlike in classic DY*, this proof does not need to be redone even if the higher-layer protocol changes. Conversely, each higher-layer protocol is aware of the lower-layer it depends upon. If the lower layer changes, the higher layer may need to be re-verified but we carefully restrict the new proofs to a minimal set of properties about overlapping cryptographic usage.

We illustrate the general concept using state invariants. In DY*, each principal stores and maintains local state for every protocol session it participates in. For example, both participants in a confidential channel protocol usually store

a symmetric key that is used to encrypt messages between them. In addition, they may store session state, such as passwords, used by higher-layer protocols. For the security proof, we need to show that this stored data satisfies certain properties, which are captured by state invariants. For example, the symmetric key (and the password) must have a secrecy label that ensures that it can only be read by the principal and its peer. In DY*, these invariants are so far defined monolithically for all protocol layers in a global state invariant.

In contrast, in our layered approach the state invariant of each layer is defined independently, only taking higher_layer_preds as parameter, as illustrated below:

```
let state_invariant higher_layer_preds principal state =
  match state with
  | CommunicationState sym_key responder →
    ... (*Invariants needed by communication layer*)
  | HigherLayerState higher_layer_state → (*must satisfy higher−layer invariant*)
    higher_layer_preds.state_invariant principal higher_layer_state
  | _→⊥
```

Here, the state invariant for the communication layer (of Section 5) says that the stored state is structured in two disjoint parts, one part for itself and one for all higher layers. Each part enforces its own state invariant. This style allows us to easily compose multiple layers in a stack. Indeed, the communication layer invariant itself serves as a higher-layer predicate for the layers below it.

Lifting cryptographic functions. If different layers overlap in the cryptographic functions and keys they use, this can, in principle, result in an insecure composition, even if both layers are secure by themselves. For example, if a secure channel protocol makes its internal encryption key available also to higher layers, then it may be possible for the attacker to inject messages encrypted by the higher layer into the secure channel, undermining the integrity of the channel. If two layers do not conflict in this way, we say that they satisfy *implicit disjointness*, reusing a term from the setting of Universal Composability [28].

To verify a stack of protocols, we therefore need to prove that every pair of protocols is pairwise disjoint. Using our layered approach, we turn this global property into a local condition at every layer. Each layer redefines (or *lifts*) all the cryptographic functions it uses, defines local predicates specifying its own usage of these functions, and specifies disjointness conditions for the safe usage of these functions in higher layers. For any crypto function not used in a layer, these functions and predicates are simply passed through to the next layer.

As an example of the simplest case, consider a layer that does not use MACs. The local MAC usage predicate of this layer is equivalent to the higher-layer predicate, without any additional local conditions. Its MAC disjointness condition for higher layers is the same as the disjointness predicate for its lower layer.

```
let mac_disjoint key_usage key msg =
  Lower_layer.mac_disjoint key_usage key msg
let mac_predicate higher_layer_preds key_usage key msg =
  higher_layer_preds.mac_predicate key_usage key msg
```

The lifted MAC function provided by this layer has mac_disjoint and the higher-layer mac_predicate as preconditions, which means that any higher layer is free to use this MAC function, in accordance with its own local MAC usage predicate, as long as it ensures disjointness with the layers below.

Suppose a layer does use a cryptographic function, say AEAD encryption, but defines its own local keys which are not shared with any other layer. This is the most common (and most advisable) design pattern. In DY*, each key is associated with a usage string. For example, the communication layer creates symmetric keys with the usage string CommunicationLayerSymKey, and uses this key to encrypt requests and responses (see Section 5 for details). If this key usage is not used in any other layer, then the AEAD predicate of this layer can cleanly distinguish between its own usage and that of higher layers. No additional disjointess condition is needed, only those imposed by lower layers (as in MAC).

Implicit Disjointness in the General Case. The most complicated case is when the same cryptographic function and key may be used in multiple layers. This failure of key independence between protocol layers is tricky to handle in security proofs, but can unfortunately often occur in real-world protocols.

In our approach, we use the disjointness predicate to ensure that if a higher layer uses a key that is also used by some lower layer, then the messages that it uses (e.g., encrypts) with this key are disjoint from (e.g., have different formats than) those used in lower layers. For example, the communication layer's AEAD disjointness predicate requires that the higher layer only encrypts messages with keys or formats that do not conflict with this or lower layers:

```
let aead_disjoint key_usage key plaintext =
  key_usage == "CommunicationLayerSymKey" ⟹
  match split plaintext with
  | Success ("CommunicationLayerRequest", message) → ⊥
  | Success ("CommunicationLayerResponse", message) → ⊥
  | Success _ | Error e → Lower_layer.aead_disjoint key_usage key plaintext
```

As long as the higher layer meets this condition, it can freely use the AEAD key and enforce its own local AEAD usage predicate.

The communication layer then defines its own local AEAD usage predicate, encompassing all the ways that AEAD may be used by this or higher layers:

```
let aead_predicate higher_layer_preds key_usage key plaintext =
  match key_usage with
  | "CommunicationLayerSymKey" →
   (match split plaintext with
    | Success ("CommunicationLayerRequest", message) →
      communication_layer_request_predicate higher_layer_preds message
    | Success ("CommunicationLayerResponse", message) →
      communication_layer_response_predicate higher_layer_preds message
    | Success _ | Error e → higher_layer_preds.aead_predicate key_usage key plaintext)
  | _ → higher_layer_preds.aead_predicate key_usage key plaintext
```

This predicate states that if the key usage is CommunicationLayerSymKey, and the plaintext matches the format of the communication layer's request

or response, then the inner message must satisfy the communication layer's request_predicate or response_predicate, which may in turn take into account additional conditions specified in the higher layer predicates. If the key has a different usage or the plaintext has a different format, then they must satisfy the higher layer's AEAD encryption predicate. Hence, the higher layer may either (1) call the communication layer to encrypt plaintexts, by obeying its request or response API, or (2) use independent keys to encrypt its own plaintexts, or (3) use the same key but with a disjoint message format. In the latter two cases, the key and message must satisfy the higher-layer usage predicate.

Compositional Verification. Importantly, when verifying the higher layer, we do not need to understand the possibly complex details of the lower-layer protocol implementation encoded in its usage predicate; we only need to prove the lower-layer disjointness predicate and the higher-layer usage predicate. We also note that these predicate definitions are verified, not trusted. If a protocol designer incorrectly makes them too strong or too weak, then typechecking will fail at the lower layer or at the higher layer.

Altogether, our changes extend DY* to a fully compositional layered protocol verification framework. Each layer only needs to be verified once, and changes to any layer implementation affect only those higher layers that reuse the same crypto functions and keys. So far, we have only considered vertical and sequential compositions of layers into a protocol stack. In future work, we intend to extend this framework to account for other composition patterns, such as horizontal compositions. Note that such composite protocols are already verifiable in DY*, but they can not benefit from the compositional proof technique in this paper.

5 Instantiation: Generic PKI and Communication Layers

We now illustrate how to instantiate the approach presented in the previous section by two layers, a simple PKI and a communication layer that uses it.

5.1 A Layer for Public-Key Infrastructure

The PKI layer models the functionality of a certificate authority, and hence, the correct distribution of public keys, but, importantly, also the generation of public/private key-pairs and their management/storage at principals. Keys can have different types (public-key encryption, Diffie-Hellman key exchange, signing, MACing, etc.) and usages. The PKI layer exposes APIs to generate and retrieve keys of the desired type with an intended usage. Using labels, it additionally guarantees that the private keys of principals indeed belong to (and are only known by) the respective principals, and that the predicates that hold true at the higher layer, in particular, regarding the state of principals, also hold true in the PKI layer, following the principle outlined in Section 4.

5.2 A Layer for Confidential and Authenticated Communication

As a second instance of our layering approach, we design a communication layer providing APIs to exchange messages with different types of security guarantees.

We model sending *authenticated* and/or *confidential* (single) messages as well as *request-response pairs*. This layer is built on top of the PKI layer.

For all functions in the interface of the communication layer we give *implementations*, showing that the pre- and post-conditions can be realized based on cryptographic primitives. For some functions/channel types, we even have multiple implementations, including one inspired by TLS 1.3., showing that our guarantees can be achieved by real-world protocols.

Interface and Guarantees. At a high level, the sender of a message using the communication layer can convey not just the message itself, but also some proof information, using new predicates exposed by the layer for applications to define. The guarantees that the receiver of the message gets depend on the type of communication (e.g. authenticated). These guarantees may talk about the contents of the message, but also may convey information about the state of the sender or about past events in the trace, which greatly facilitates scalable and composable analysis of protocols.

We now examine specific examples of the guarantees provided by the communication layer, in the context of our source routing protocol from Section 3.

Intuitively, the receiver of an *authenticated* message should be guaranteed that the sender of the message, if honest, followed the protocol when creating the message. Since the details of message creation depend on the specific application being modeled, the application may specify the exact properties that should hold for an honest sender, by defining the predicate authenticated_send_pred exposed by the communication layer. In the source routing example, this predicate states that if all participants on the path are honest, then the previous participant processed the message.

Similarly, a *confidential* message should guarantee the receiver that its contents do not leak to the attacker in transit. As in the authenticated case, the details of what parties should be allowed to know are application-specific, and can be defined in the confidential_send_pred exposed by the communication layer. We note that while secrecy properties are natural candidates for this predicate, we can also include more general guarantees, as in the authenticated case. In the source routing example, the predicate says that the content of the message can only be known by the participants on the path.

We can also send messages that are both authenticated and confidential, using the authenticated_confidential_send_pred predicate to specify the guarantees the application expects, which are a combination of those for authenticated-only and confidential-only messages. Similarly, we can send request/response pairs, which resemble a confidential (and optionally authenticated) message, responded to by an authenticated and confidential message. These pairs use their own predicates request_pred and response_pred, which are similar to the other predicates, but have slightly more expressive power, e.g., the response predicate can refer to both the request and the response.

In addition to the communication functions, the layer also exposes lifted versions of the cryptographic functions provided by DY*, as described in Section 4. The communication layer uses a symmetric key for securing request/response

pairs (as is common in practice), and exposes this key to the higher layer, which may freely use this key as long as it does not interfere with the communication layer, as already discussed in Section 4.

Implementation. As a sanity check and to prove that the interface of the communication layer, and the guarantees that come with it, can be realized, we provide implementations of the interface for the various channels and prove (in DY*) that these typecheck against the interface, and hence, provide the desired guarantees. Our implementations are rather straightforward and are based on public-key cryptography, which is why they are based on the PKI layer. However, as mentioned, we also provide a simplified implementation of TLS 1.3 (see below).

The implementation for sending authenticated messages adds a signature to the original message, while confidential messages simply encrypt the original message with the public key of the receiver. Messages which are both confidential and authenticated use an encrypt-then-sign scheme. For request/response pairs, we use a hybrid encryption scheme where the request contains a fresh symmetric key, encrypted asymmetrically with the public key of the receiver. We provide two variants for the encryption of the request body, one using this symmetric key, and one using the public key of the receiver. In either case, the receiver then uses the symmetric key from the request to encrypt the response.

The guarantees of the communication layer can then be derived (internally to the communication layer) from the guarantees of the cryptographic functions used. In this way, we implement the predicates exposed by the communication layer (e.g. request_pred) from the lower-level predicates exposed by the PKI layer to the communication layer.

TLS Implementation. The Transport Layer Security Protocol [33,34] is a widely used cryptographic protocol ensuring end-to-end security of messages exchanged by applications running on top of it. Various prior works [9,11,19,20] have identified flaws and presented proofs and verified implementations of TLS.

We provide a second implementation of the request/response pattern of the communication layer based on a simplified version of the latest TLS version, namely TLS 1.3, which illustrates that our communication layer can have multiple, including real-word, implementations. Importantly, as explained in Section 4, typically higher layers, e.g., those using the communication layer, can be analyzed independently of the specific implementations of lower layers.

Our model of TLS is itself modularized into two layers: one for the *handshake* protocol for key-exchange (TLS AKE in Section 1), and the other is the *record* layer for the transmission of messages (TLS Stream in Section 1). The handshake layer involves the exchange of three messages: (1) the initiator/client generates a Diffie-Hellman (DH) key pair and sends their public key to the server; (2) the responder/server generates its own DH keypair and shares the public key with the client signed with the server's signature key; (3) the client acknowledges the receipt of the server's public key signed with their signature key. At the end of the protocol, both client and server share a secret alongside authenticating themselves with each other. The guarantees for the keys used in the three steps are obtained from the PKI layer on top of which this layer is implemented.

6 Analysis of Running Examples

Next, we present our security analysis of the BA example from Section 3, showing how the communication layer greatly facilitates this analysis and makes it independent of the concrete implementation of that layer; the analysis of the source routing example can be found in Section 6.1.

Obviously, we model the BA example on top of the communication layer: The client has two functions for creating and sending the requests, and two for receiving the corresponding responses. The client stores the secret it receives from the server in its state. Further, there is a function for the server to receive and respond to each request. When it receives an account registration request it generates a new secret and stores this secret next to the password in its state.

The main property that we want to prove for this example is that the newly created secret is only known to the client and server, provided neither is corrupted. Our formalization is shown below. It is written from the perspective of the client, so that the client has some guarantee of the secrecy of the secret it receives in the protocol. It states that for all traces t0, clients client, servers server, and secrets secret if client has stored secret received from server into its state and neither client nor server is corrupt at the end of t0, then the attacker does not know secret at the end of t0.

```
val secrecy_client: ... → client:principal → server:principal → secret:bytes →
  LCrypto unit (...)
  (requires (λ t0 → ( ... ∧
       is_secret_in_client_state ... client server secret ∧
       ¬(corrupt_at (len t0) client ∨ corrupt_at (len t0) server) )))
  (ensures (λ t0 _t1 → ... ∧ is_unknown_to_attacker_at (len t0) secret ))
```

The core of the proof is a set of global trace invariants, which we show are preserved by each protocol participant, and which are strong enough to imply our security property. Our main invariant (see below) is an invariant on the state of clients, which talks about the secret secret stored in the state of a client client who at the outset of the protocol wants to communicate with a server server.

```
generated_before idx secret (readers [client; server]) (nonce_usage "Secret") ∨
corrupt_at idx client ∨ corrupt_at idx server
```

It essentially says that if secret is stored (in the client's storage), then secret has label (readers [client; server]) or either client or server is corrupt. The security property then immediately follows from this invariant together with the preconditions, in particular that client stores the secret in its state, and the soundness of the DY* labeling system, namely that labels guarantee that secrets are only known to those parties mentioned in the labels.[6]

The bulk of the proof then lies in proving that this is indeed an invariant of valid traces, in that the protocol functions preserve it. The secret is only written

[6] The soundness of the labeling system has been mechanically proved once and for all in DY* itself.

into the state of a client in the function where a client receives a response to an access request. If both parties are honest at that time, the communication layer guarantees that the predicate response pred (excerpt shown below) holds:

```
generated_before idx secret (get_label ... password) (nonce_usage "Secret")
```

This predicate states that the label of secret is the same as the one of password sent in the corresponding request.

Note how the communication layer makes it simple to convey information from one principal to another — in this particular case, the server needs to prove the response_pred before it can send a response containing a secret, and the client can then make use of this same predicate upon receiving the message. This means that the server, who does not know the label of the password, can still convey to the client that the labels of the secret and the password are the same (trivially so, since the server generates the secret with this property).

Another part of a client's state invariant states that the password is labeled with exactly the client and the server it interacts with. The client, knowing the label of the password, is then able to determine precisely the label of the secret, allowing us to establish the state invariant for the client.

Simplicity of analysis and independence from channel implementations. This case study highlights several benefits of the layered approach. First, the higher-level interface of the communication layer makes it quite natural to model the BA example, just as secure communication libraries simplify protocol implementation by abstracting away from cryptographic primitives. Moreover, the security of the BA example can also be proven at a much higher level of abstraction, using the guarantees provided by the communication layer (and customizable by higher layers via predicates), again hiding fine details of cryptography. Since the BA example does not directly use any cryptography, implicit disjointness with the communication layer comes for free, as described in Section 4. In particular, this means that we can switch our implementation of the communication layer (e.g. between our simple and TLS implementations) without changes to the analysis of the BA example.

Implicit disjointness. We also implemented and analyzed a variant of the BA example where the server uses the symmetric key exposed by the communication layer to encrypt the secret, in addition to the underlying encryption used by the communication layer. This illustrates the flexibility of our layered approach. As outlined in Section 4, the server must then prove that it satisfies the implicit disjointness predicate whenever it encrypts messages using the exposed symmetric key. Since the definition of this predicate depends on the underlying implementation of the communication channel, the analysis of the BA example may need to be adjusted when this implementation changes, but only in this specific aspect. This dependence on the implementation is inherent if the same key material is used across layers, as implicit disjointness properties need to be established, and depend on the specific format of messages.

Informal Benchmark. While we do not have an unlayered version of this example to compare to, we can briefly examine the sizes of each component of the stack the BA example is built on, and the verification times for different components of the stack. We expect that a version of this example without layers would take at least as long to verify as the sum of the times required for the separate components of the stack, and likely longer, as the predicates involved would be more complex. This lets us get a rough idea of the time savings of being able to recheck changes to the example without needing to reverify the full stack. The BA example is built on top of the communication layer, which in turn is built on the PKI layer, which is built on core DY*. In our benchmarks, we verify the PKI layer together with base DY*, but this still gives a sense of the overall time taken for the full stack. The results of this (informal) benchmark can be found in Table 1.

	Size (LoC)	Verif. Time (s)
SR example	1779	114
BA example	1216	117
Comm. Layer	2125	142
PKI Layer and Core DY*	4736	358

Table 1: Size (in lines of code) and verification time (in seconds) of components of the BA and SR examples

Despite the relative simplicity of the stack in the BA example, only around 19 percent of the total verification time is needed for the BA example in isolation. With more layers and more complex layers, we expect that this full-stack verification time increases even more, further increasing the benefits of the layering approach.

6.1 Source Routing Example

In the source routing example, we send a message along a specified path of participants. The guarantees we expect to get at the final receiver of the message depend on whether the messages used in the protocol are authenticated, confidential, or both. We focus on the (more interesting) case where messages are authenticated. In this case, we would like to guarantee the final receiver that the message has indeed followed the specified path (as long as all participants in the path are honest).

In modelling this property, we make use of DY* *events* to mark when parts of the protocol have happened. For instance, we generate an event each time a participant processes a message, before sending out its own message to the next participant in the list, as well as one event when the final participant receives a message and the protocol is complete. These message processing events, as well as the messages themselves, contain a counter indicating which position in the list they correspond to. For instance, the first participant in the protocol sends a message with counter value 0, which is then received by the second participant, who creates an event with counter value 1 before sending out a message with

counter value 1 to the next participant. In this way, the counter value tracks for events which participant in the list is currently processing a message, and for messages which participant in the list sent the message.

Our formal property (see below) then shows first that these message processing events occur in the correct order (that is, with increasing counter values), and then shows that between each pair of events (with counter values $i < j$) lies a corresponding message (with counter value $j - 1$ — the message whose receipt leads to the creation of the event with counter j).

```
val authenticated_path_integrity:
  trace_idx → receiver → principal_list → nonce → counter →
LCrypto unit ...
(requires (λ t0 → ... ∧
  did_event_occur_at trace_idx receiver (finished principal_list nonce counter) ∧
  (∀ p. mem p principal_list ⟹ ¬(corrupt_at (trace_len t0) p))
))
(ensures (λ t0 _ _→ ... ∧ (
  counter = length principal_list − 1 ∧
  (∀ (i:nat). (i ≤ counter) ⟹ (∃ (k_i m_i:timestamp). ... ∧
    (* a message processing event with counter [i] was created
       and a message with counter [i−1] was sent *)
      did_event_occur_at k_i ... (processed_message principal_list nonce i)
      ∧ (i > 0 ⟹ (let recv_msg = MsgWithCounter principal_list (i−1) nonce in
                   is_authenticated_message_sent_at m_i ... recv_msg ... ))))
  ∧
  (∀ (k_j:timestamp) i j. (i < j) ∧ (j ≤ length principal_list − 1) ⟹
  (* the message processing events with counters [i] and [j] happen in the right order
     and between these events there is a message with counter [j−1] *)
    (did_event_occur_at k_j ... (processed_message principal_list nonce j) ⟹
      (∃ (k_i m_j:timestamp). (k_i ≤ m_j) ∧ (m_j ≤ k_j)
        ∧ (did_event_occur_at k_i ... (processed_message principal_list nonce i))
        ∧ (let recv_msg = MsgWithCounter principal_list (j−1) nonce in
           is_authenticated_message_sent_at m_j ... recv_msg ... ))))))
```

We can conclude from this that not only do the events occur in the correct order, but so do the messages, and so the specified path was indeed followed.

The key predicate in proving this property is the event predicate for the message processing events. This predicate (shown below) requires that if an event containing a path principal_list, a secret nonce and a counter i with $i > 0$ is on the trace at index t, the trace must contain at some previous indices both the event with counter $i - 1$ and the message with counter $i - 1$, with the event occurring before the message:

```
∃(ev_idx msg_idx:timestamp). (ev_idx < msg_idx) ∧ (msg_idx < t)
  ∧ did_event_occur_at ev_idx ... (processed_message principal_list nonce (i−1))
  ∧ (let recv_msg = MsgWithCounter principal_list (i−1) nonce in
     is_authenticated_message_sent_at msg_idx ... recv_msg .... )
```

This is essentially a one-step version of the overall property that we want to prove, and we can establish the security property from this event predicate by induction.

Every time a participant receives and processes a message, we would like to add a corresponding event to the trace, but to do so, we must first establish the event predicate. The receiver learns from the receive function's postconditions that the message it receives (with counter $i-1$) was indeed sent at some previous timestamp, and so needs only to know that the event with counter $i-1$ is on the trace prior to this message being sent. The sender of the message can communicate this information to the receiver via the authenticated_send_pred predicate of the communication layer. The predicate specifies that a message containing a path principal_list, a counter counter, and a secret nonce can be sent at index t only if the desired event with the same counter value happened at some prior time:

```
did_event_occur_before t ... (processed_message principal_list nonce counter))
```

As with the BA example, the communication layer enables us to easily transfer information needed for proofs alongside messages, without needing to include that information as part of the message itself (where it might impact the protocol flow).

7 Related Work

This paper presents a symbolic protocol analysis methodology that extends an existing semi-automated verification framework (DY⋆), in order to enable modular, scalable, and compositional protocol proofs. Many prior and concurrent works present related results. In this section, we compare our approach with closely related work on developing machine-checked compositional protocol proofs. A wider survey of the area may be found in [2].

Symbolic Protocol Analysis. Tools that verify protocols in the symbolic or Dolev-Yao model [21] rely on a simplified abstraction of cryptography that makes it easier to automate security proofs and to find logical flaws and attacks on protocols. In particular, tools like ProVerif [14] and Tamarin [30] have been successful in performing fully automated proofs of protocols like TLS 1.3 [19,9]. However, these tools do not support modular specifications or compositional proofs, do not allow inductive proofs, and do not scale well to larger protocols.

Type-based approaches like DY⋆ [5] and its predecessors [4,10] have been used for the modular proofs of large protocols like ACME [6] and Signal [5]. This paper represents a substantial extension to DY⋆ that enables layered analysis by changing the way security predicates are defined and composed. An early idea for an authenticated channel layer appeared in the ACME analysis but our treatment in this paper extends it with a full set of communication patterns (confidential channels, request-response exchanges) and implementations of these patterns by multiple protocols, including TLS.

Symbolic Protocol Composition. Prior works [18,25,31,17,24,26] have explored conditions under which symbolic security proofs of two cryptographic protocols can be safely composed. The composition patterns considered include parallel composition of unrelated protocols, sequential composition—where one protocol uses a secret generated by another, and vertical composition of protocols that are layered one above another. In all these works, the key idea is to limit the interactions between composed protocols or to characterize under what conditions protocols can safely be composed; along with the analysis of individual protocols, the conditions for their secure composition need to be checked.

Most of these prior works do not support machine-checked proofs. [26] formalizes the composition proof in Isabelle but the individual protocol proofs are usually done using a different tool (PSPSP) and protocols are not expressed in a full-fledged programming language, rather in a much simpler domain specific language. In contrast, in our work, the full development, including soundness proofs, composition proofs, and individual protocol proofs are all in the same framework ($\mathsf{F}^\star$), and by this, come with all the benefits of such a fully-fledged programming environment (see also Section 2). Furthermore, by extending an existing expressive tool, we automatically support all the cryptographic primitives one can express in $\mathsf{DY}^\star$, including Diffie-Hellman, which are essential for protocols like TLS, but are not modeled in prior works like [26].

Compositional Cryptographic Proofs. Modularity and composability have long been guiding principles for provable security, perhaps best exemplified by the line of work on universal composability (UC) [15,29]. More recently, a series of tools [22,16,1] seek to apply modular design principles to mechanized cryptographic proofs. One of the most recent works is Owl [23], which produces proofs of protocols using information flow types, and unlike, $\mathsf{DY}^\star$ aims at full automation for protocols specified in a domain-specific language and restricted to static corruption, as the framework is based on UC.

In contrast to these works on computational cryptographic proofs, our work is in the symbolic model, and hence makes less precise assumptions about cryptographic primitives. In return, our framework is capable of analyzing large protocols with ease, which still remains a challenge for computational tools.

8 Conclusion

In this paper, we presented a layered approach to symbolic protocol analysis, as an extension to $\mathsf{DY}^\star$. This approach allows us to *compositionally* analyze stacks of layered protocols, looking at each layer individually. While $\mathsf{DY}^\star$ already allows for modular and scalable analyses, it does not enable compositional proofs. So, when any layer changed, the entire protocol stack had to be re-proved. In our approach, lower layer proofs never have to be redone if higher layers are modified. Furthermore, if a higher layer does not use the same cryptographic primitives or the same key material as its lower layers, then it does not have to be reproved if the lower layer implementation changes.

Our approach also accounts for cases where different layers use the same cryptographic functions and key material. By lifting cryptographic functions at each layer and explicitly expressing sufficient disjointness conditions, dependencies between layers are kept to a minimal set of predicates. Only these disjointness conditions need to be re-proved when a lower-layer implementation changes, while the rest of the proof remains unchanged.

We highlight the utility of this approach by means of two independently useful layers for security protocols, namely a PKI and a secure communication layer. The communication layer allows for the analysis of applications based on abstract secure communication, without the need to consider or verify details of the underlying cryptographic primitives and communication. We use these layers to implement two case studies that use various communication patterns. While it takes around 10 minutes to verify the entire stack of one of our examples, each example takes only around 2 minutes to verify on its own, illustrating the time savings of being able to verify layers independently.

As pointed out in Section 4, so far, we have only considered vertical and sequential compositions of layers into a protocol stack. In future work, we intend to extend this approach to horizontal compositions, where the main challenge lies in determining how predicates from different protocols at the same layer of the protocol stack can be safely merged. Furthermore, all our analysis is in the symbolic model. Investigating whether our layered approach can also be applied to computational verification tools is an interesting topic for future research.

References

1. Abate, C., Haselwarter, P.G., Rivas, E., Muylder, A.V., Winterhalter, T., Hriţcu, C., Maillard, K., Spitters, B.: SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In: 2021 IEEE 34th Computer Security Foundations Symposium (CSF). pp. 1–15. IEEE Computer Society (2021)
2. Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., Parno, B.: SoK: Computer-aided cryptography. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 777–795 (2021), `https://eprint.iacr.org/2019/1393`
3. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-Aided Security Proofs for the Working Cryptographer. In: CRYPTO. LNCS, vol. 6841, pp. 71–90. Springer (2011)
4. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM Transactions on Programming Languages and Systems (TOPLAS) **33**(2), 1–45 (2011)
5. Bhargavan, K., Bichhawat, A., Do, Q.H., Hosseyni, P., Küsters, R., Schmitz, G., Würtele, T.: DY*: A modular symbolic verification framework for executable cryptographic protocol code. In: IEEE European Symposium on Security and Privacy (EuroS&P). pp. 523–542 (2021)
6. Bhargavan, K., Bichhawat, A., Do, Q.H., Hosseyni, P., Küsters, R., Schmitz, G., Würtele, T.: An in-depth symbolic security analysis of the ACME standard. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021. pp. 2601–2617. ACM (2021)

7. Bhargavan, K., Bichhawat, A., Do, Q.H., Hosseyni, P., Küsters, R., Schmitz, G., Würtele, T.: A tutorial-style introduction to DY*. In: Dougherty, D., Meseguer, J., Mödersheim, S.A., Rowe, P.D. (eds.) Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday. Lecture Notes in Computer Science, vol. 13066, pp. 77–97. Springer (2021)
8. Bhargavan, K., Bichhawat, A., Hosseyni, P., Küsters, R., Pruiksma, K., Schmitz, G., Waldmann, C., , Würtele, T.: DY* layering source code (2023), `https://publ.sec.uni-stuttgart.de/esorics23-layered-symbolic-security-analysis-in-dystar-code.zip`
9. Bhargavan, K., Blanchet, B., Kobeissi, N.: Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In: IEEE S&P. pp. 483–502 (2017)
10. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 445–456 (2010)
11. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zanella-Béguelin, S.: Proving the TLS handshake secure (as it is). In: Advances in Cryptology – CRYPTO 2014. pp. 235–255 (2014)
12. Bhargavan, K., Lavaud, A.D., Fournet, C., Pironti, A., Strub, P.Y.: Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In: 2014 IEEE Symposium on Security and Privacy (2014)
13. Blanchet, B.: CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In: Dagstuhl seminar "Formal Protocol Verification Applied. vol. 117, p. 156 (2007)
14. Blanchet, B.: Modeling and verifying security protocols with the applied pi calculus and ProVerif. In: Found. Trends Priv. Secur. vol. 1, pp. 1–135 (2016)
15. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001). pp. 136–145. IEEE Computer Society (2001)
16. Canetti, R., Stoughton, A., Varia, M.: EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In: 2019 IEEE 32th Computer Security Foundations Symposium (CSF). pp. 167–183. IEEE Computer Society (2019)
17. Cheval, V., Cortier, V., Warinschi, B.: Secure composition of PKIs with public key protocols. In: 30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017. pp. 144–158. IEEE Computer Society (2017)
18. Ciobâca, S., Cortier, V.: Protocol composition for arbitrary primitives. In: 23rd IEEE Computer Security Foundations Symposium. pp. 322–336 (2010)
19. Cremers, C., Horvat, M., Hoyland, J., Scott, S., van der Merwe, T.: A Comprehensive Symbolic Analysis of TLS 1.3. In: ACM CCS. pp. 1773–1788 (2017)
20. Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S., Bhargavan, K., Pan, J., Zinzindohoue, J.K.: Implementing and proving the TLS 1.3 record layer. In: IEEE S&P. pp. 463–482 (2017)
21. Dolev, D., Yao, A.C.: On the Security of Public-Key Protocols. IEEE Transactions on Information Theory **29**(2), 198–208 (1983)
22. Fournet, C., Kohlweiss, M., Strub, P.: Modular code-based cryptographic verification. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011. pp. 341–350. ACM (2011)

23. Gancher, J., Gibson, S., Singh, P., Dharanikota, S., Parno, B.: Owl: Compositional verification of security protocols via an information-flow type system. Cryptology ePrint Archive, Paper 2023/473 (2023), https://eprint.iacr.org/2023/473, https://eprint.iacr.org/2023/473
24. Gondron, S., Mödersheim, S.: Vertical composition and sound payload abstraction for stateful protocols. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021. pp. 1–16. IEEE (2021)
25. Groß, T., Mödersheim, S.: Vertical protocol composition. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011. pp. 235–250. IEEE Computer Society (2011)
26. Hess, A.V., Mödersheim, S.A., Brucker, A.D.: Stateful protocol composition in isabelle/hol. ACM Trans. Priv. Secur. **26**(3) (apr 2023)
27. Ho, S., Protzenko, J., Bichhawat, A., Bhargavan, K.: Noise*: A library of verified high-performance secure channel protocol implementations. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022. pp. 107–124. IEEE (2022)
28. Küsters, R., Tuengerthal, M.: Composition Theorems Without Pre-Established Session Identifiers. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011). pp. 41–50. ACM Press (2011), https://publ.sec.uni-stuttgart.de/kuesterstuengerthal-ccs-2011.pdf
29. Küsters, R., Tuengerthal, M., Rausch, D.: The IITM Model: a Simple and Expressive Model for Universal Composability. J. Cryptol. **33**(4), 1461–1584 (2020), https://publ.sec.uni-stuttgart.de/kuesterstuengerthalrausch-iitm-joc-2020.pdf
30. Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN prover for the symbolic analysis of security protocols. In: CAV. LNCS, vol. 8044, pp. 696–701. Springer (2013)
31. Mödersheim, S., Viganò, L.: Sufficient conditions for vertical composition of security protocols. In: Moriai, S., Jaeger, T., Sakurai, K. (eds.) 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014. pp. 435–446. ACM (2014)
32. Reschke, J.: The 'basic' HTTP authentication scheme. RFC 7617 (Sep 2015), https://www.rfc-editor.org/info/rfc7617
33. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Aug 2018), https://www.rfc-editor.org/info/rfc8446
34. Rescorla, E., Dierks, T.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Aug 2008), https://www.rfc-editor.org/info/rfc5246
35. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent types and multi-monadic effects in F*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 256–270 (2016)
36. Wallez, T., Protzenko, J., Beurdouche, B., Bhargavan, K.: TreeSync: Authenticated group management for messaging layer security. IACR Cryptol. ePrint Arch. p. 1732 (2022), https://eprint.iacr.org/2022/1732