Cryptis: Composition and Separation for Tagged Protocols

ARTHUR AZEVEDO DE AMORIM, Boston University, USA
AMAL AHMED, Northeastern University, USA
MARCO GABOARDI, Boston University, USA

Compositionality is recognized as a major challenge for the verification of cryptographic protocols. Modern verification tools can analyze sophisticated designs in isolation, but provide few guarantees when a protocol is part of a larger system, interacting with components that were not included in the analysis.

Prior work demonstrated that reasoning about composite protocols is simpler when their messages are *tagged*, which prevents them from being misused by other protocols running concurrently in the system. Unfortunately, tools that support this style of reasoning are confined to special-purpose type systems, and it is not clear they could be extended to handle more programming features or security properties.

We propose to fill in this gap with *Cryptis*, a new logic for verifying the correctness of cryptographic protocols. Cryptis provides first-class support for tagged composition through *tag invariants*. A tag invariant is an assertion that guarantees that every message tagged in a certain way satisfies some property. To prove a protocol correct, a user just needs to specify all the tag invariants that are needed for its proof. If different protocols use disjoint tags, they can safely execute in parallel, even if they share private keys or other secrets. In this sense, tag invariants are akin to the point-to assertions of separation logic: if two processes use disjoint parts of the heap, they can safely run together.

We have implemented Cryptis in Coq with the Iris framework, and used it to verify a variety of case studies. The case studies demonstrate that Cryptis is well-suited for reasoning about rich protocol features and guarantees, such as forward secrecy, branching control flow, loops and composition. Moreover, verifying a composite system can be done *modularly*, treating each component as a black box.

Additional Key Words and Phrases: keyword1, keyword2, keyword3

1 INTRODUCTION

Two agents, Alice and Bob, would like to communicate a secret s over an insecure network. Alice encrypts s using Bob's public key pk_B and sends it over, while Bob waits to receive a message and decrypts it using his secret key sk_B. This process is depicted as follows:

$$A \triangleq \mathsf{send}(\mathsf{enc}(pk_B, s)) \qquad\qquad B \triangleq \mathsf{dec}(sk_B, \mathsf{recv}()).$$

How would we argue that this protocol is secure? To simplify our analysis, let us ignore attacks that manipulate messages as raw bit strings, or that rely on timing or other covert channels. We'll assume that cryptographic operations behave as perfect black boxes—the so-called *symbolic* model of cryptography. Then, we can prove security by noting that the system preserves the following invariants:

(I_1) The key sk_B is not known to the attacker.
(I_2) The secret s is not known to the attacker.

Bob preserves the invariants because it never sends anything to the network. Alice does send something to the network, but in the symbolic model we assume that the message can only be decrypted if the attacker knows the secret key sk_B, and that it is impossible to extract the key from an encrypted message. Therefore, this action does not affect the attacker's knowledge, and s remains secret.

Authors' addresses: Arthur Azevedo de Amorim, Boston University, USA; Amal Ahmed, Northeastern University, USA; Marco Gaboardi, Boston University, USA.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

There are many approaches for formalizing this kind of reasoning. In this paper, we'll focus on semi-automated tools such as program logics and type systems [25, 14, 15, 8, 9], which require some manual intervention, but in return provide more modular, scalable analyses than fully automated tools. One such tool is DY* [15], a state-of-the-art F* library that leverages dependent types and first-class state invariants [47] to reason about protocols. Using DY*, we can use types to argue that each action preserves a protocol's invariants, thus obtaining modular proofs of security that can be checked efficiently.

Though such type systems and related techniques can substantially simplify invariant reasoning, they have trouble handling an important feature: composition. Nothing guarantees that an invariant remains valid if a protocol is part of a larger system, which includes actions that were not accounted for in the protocol's security proof. If we are not careful about how we compose a protocol, we might end up ruining its security guarantees even if it is formally verified. For example, suppose that Bob is running another program, which acts as a decryption oracle:

$$O \triangleq \mathsf{send}(\mathsf{dec}(sk_B, \mathsf{recv}())).$$

Alice's message might be delivered to O, which would end up leaking s and breaking the protocol's correctness. This scenario might seem contrived, but it is not that unrealistic. For instance, in some RSA-based signature schemes, signing a message is implemented using the same operation as decryption and encryption, and such a vulnerability might arise if sk_B is used for both signing and decryption.

Of course, composition is not strictly needed for this kind of analysis. The above flaw could also be detected by patching the definitions and verifying all components together, ensuring that all the invariants of the system are preserved. But this would be too expensive, both in terms of human labor and computational resources. To remedy this issue, several works have sought more compositional methods for protocol verification [25, 18, 5, 6, 31, 22, 32, 3]. Among semi-automated tools, a particularly relevant proposal is Protocol Composition Logic (PCL) [25]. PCL follows a composition methodology reminiscent of the Owicki-Gries method [44]: to compose the proof of correctness for two protocols, we must show that each protocol's invariants are preserved by the execution of the other protocol. To illustrate, let us consider how we might attempt to verify a composite system where A, B and O run simultaneously. After verifying A and B, we need to show that O is secure. Since O relays any message encrypted for Bob, we need to use the following property as an invariant:

(I_3) Every message encrypted under pk_B can be disclosed to the attacker.

This allows us to show that O does not leak any unintended messages. To conclude the proof of composition, we need to show that O preserves the invariants of A and B, and vice versa. The first step is easy since O does not directly leak sk_B or s. However, we run into trouble when attempting the next step: I_3 is *not* invariant under the execution of A, since it uses pk_B to encrypt s, which should not be known do the attacker. Therefore, the composition rule of PCL does not apply, and we are prevented from running an insecure system.

Assuming that there was a legitimate reason for us to run these protocols in the first place, we might attempt to patch their definition so that they can be composed securely. A common approach for preventing this type of issue is to use *tagged protocols* [22, 6, 5, 40, 19, 20, 3]. A tagged protocol is one where each type of message carries a particular tag that uniquely identifies it. If each protocol ignores messages that it is not supposed to use, they will not interfere with each other. For example, we could modify the previous protocols so that first one uses the tag `"p1"`, while the second one uses the tag "p2".

$$A' \triangleq \mathsf{send}(\mathsf{enc}(pk_B, \mathsf{tag}(\texttt{"p1"}, s))) \qquad B' \triangleq \mathsf{untag}(\texttt{"p1"}, \mathsf{dec}(sk_B, \mathsf{recv}()))$$
$$O' \triangleq \mathsf{send}(\mathsf{untag}(\texttt{"p2"}, \mathsf{dec}(sk_B, \mathsf{recv}()))).$$

The form $\mathsf{tag}(t, m)$ denotes the result of concatenating the tag t with the message m. We assume that messages are serialized so that it is possible to split such a message into t and m. The form $\mathsf{untag}(t, m')$ ensures that m' is of the form $\mathsf{tag}(t, m)$ and returns m; if this is not the case, the execution halts. With this modification, it is still possible to show that the compound system comprising A' and B' is secure with the invariants I_1 and I_2. To verify O', we can modify the invariant I_3 as follows:

(I_3') Every message encrypted under the tag "p2" can be disclosed to the attacker.

The PCL proofs of A, B and O could be adapted to these new versions without much effort. However, we would still need to apply the logic's composition rule to argue formally that the combined system is secure. This is not ideal: like the Owicki-Gries method, verifying a composite protocol can require a number of additional verification steps that grows quadratically in the total number of statements of the protocols. Moreover, since we have to inspect the code of all protocols after verification, we cannot reuse them as black boxes.

Rather than requiring all this extra work, the literature on tagged protocols advocates for a simpler, more robust approach [22, 5, 6, 40, 20, 19, 3]: just apply some general theorem that guarantees that protocols with disjoint tags can be composed securely. A simple syntactic check suffices to discharge many of the expensive invariant checks required in PCL. Unfortunately, this line of work usually targets a fixed set of properties for programs written in a specialized process calculus or type system, which limits its applicability. Moreover, many of these approaches are not implemented, making their guarantees less robust.

Our Proposal: Cryptis. In this paper, we propose to bridge the gap between expressive, mechanized logics and first-class support for reasoning about tagged protocols. We introduce *Cryptis*, an expressive logic for symbolic cryptography that enables protocol composition via tagging. In Cryptis, each protocol specifies, on a per-tag basis, what invariants its messages must satisfy. For example, to verify the protocol A' shown above, we can formalize its invariants as Cryptis assertions, guaranteeing that sk_B and s are not leaked to the attacker:

$$I_1 \triangleq \neg\, \mathsf{pterm}(sk_B) \qquad\qquad I_2 \triangleq \neg\, \mathsf{pterm}(s).$$

The predicate $\mathsf{pterm}(m)$ states that the term m is public and can be sent to the network with no harm. Cryptis uses public terms as an overapproximation of the attacker knowledge, so the two predicates above mean that those terms are unknown to the attacker.

To verify A', it suffices to show that it is safe to send out s to the network after tagging and encrypting it. Since the attacker might see that message, we need to show that it is public. This is expressed with the following assertion:

$$\mathsf{pterm}(\mathsf{enc}(pk_B, \mathsf{tag}(\texttt{"p1"}, s))).$$

Proving this assertion boils down to two conditions. First, we need to show that s can be made public if sk_B is public—after all, if the attacker knows sk_B, they can simply decrypt that message and extract s. This follows directly from I_1, since sk_B is not public.

Second, Cryptis requires each encrypted tagged message to satisfy some user-supplied invariant associated with that tag. This is expressed with the predicate $\mathsf{enc\_pred}(t, \varphi)$, which says that, for any encryption key k and any messagem m, m can be considered public after it is tagged with t

and encrypted with k, provided that $\varphi(k, m)$ holds. For this protocol, we do not require any special properties of encrypted messages, so we can attach a trivial invariant to "p1":

$$\mathsf{enc\_pred}(\texttt{"p1"}, \lambda km.\, \mathsf{true}).$$

This allows us to show that the encrypted term can be made public, which is enough to guarantee the security of the protocol.

Crucially, imposing this invariant on "p1" has no effect on protocols that use other tags, such as O'. To verify this protocol, we formalize the I_3' assumption above as tag invariant in Cryptis, stating that only public terms can be encrypted under "p2":

$$I_3' \triangleq \mathsf{enc\_pred}(\texttt{"p2"}, \lambda km.\, \mathsf{pterm}(m)).$$

This invariant implies that the term output by O' is already public, because it was tagged with "p2" when it was encrypted. Thus, O' is safe as well.

Tag invariants allow us to infer which tags a protocol uses simply by looking at its specification. This guarantees that O' is independent of A' and B', and we can show that they can safely run together simply by combining their proofs. Moreover, tag invariants are specified separately for each composed protocol, thus simplifying compositional reasoning.

The Meaning of Security. The properties encompassed by Cryptis include various flavors of *secrecy* and *authentication*. By "secrecy," we mean that certain terms cannot be decrypted by the attacker directly or be output as the result of a call to recv. By "authentication," we mean that the agents running the protocol should be capable of agreeing on certain parameters, such as their identities, a session key to encrypt communication, etc. (This notion hasn't showed up in our toy example, but it will play a more important role later on.)

Cryptis makes it possible to view secrecy and authentication through two different angles: *internally* and *externally*. The internal perspective is to state these concepts directly as formulas in the Cryptis assertion language. Secrecy is the negation of pterm, as we've seen in invariants I_1 and I_2, whereas authentication is expressed by a predicate specifying which parameters have been associated with a session.

The external perspective is to apply the *adequacy* of the Cryptis logic to relate such internal statements to properties of the operational semantics of Cryptis programs, which make no reference to the logic. This can be phrased in terms of *security games*, a popular criterion for the analysis of cryptographic protocols. In this work, a security game is simply some glue code involving the protocol that contains an assertion. By definition, the game is secure if the assertion cannot be violated, no matter what the attacker does. To state the secrecy of the protocol we've seen above, for instance, we might employ the following game, where ||| denotes parallel execution:

$$G \triangleq A' \,|||\, B' \,|||\, O'; \mathsf{assert}(\mathsf{recv}() \neq s).$$

Security follows from adequacy and from the secrecy assumption on s, since s cannot satisfy $\mathsf{pterm}(s)$ and $\neg\,\mathsf{pterm}(s)$ simultaneously. As we will see, we can formulate other games to characterize authentication as well.

Of course, protocol verification is only useful if it provides *robust* security guarantees, which hold against a large class of possible adversaries—the larger the class, the stronger the guarantees are. As in related formalisms, the adversary in Cryptis is embodied by the network (i.e., the send and recv functions), who is assumed to have the power to drop, delay and manipulate messages arbitrarily. These assumptions are captured in type-like contracts for the network: send takes a public term as its argument, whereas recv promises to return a public term. To increase the confidence in its attacker model, Cryptis provides a type system and accompanying library for implementing these functions: *any* function of the right type represents a valid attacker strategy. Besides basic programming features such as mutable state, the library provides operations for manipulating terms in the symbolic model, such as encrypting and decrypting messages or computing Diffie-Hellman exponentials. (The library is merely for convenience: the network types are actually normal Cryptis specifications, and any manually verified attacker implementation is also covered by a protocol proof.) Low-level bit operations, covert channels and probabilistic primitives are not included in the model.

Contributions and Outline. We present *Cryptis*, a compositional program logic for verifying cryptographic protocols. Cryptis is mechanized in Coq using Iris [35], an extensible higher-order concurrent separation logic, with support for ghost state and invariants. We make the following contributions:

- We introduce the notion of *tag invariant*, which allows us to impose invariants on tagged messages exchanged by a protocol. Tag invariants can be specified independently for each protocol, allowing them to be easily composed. We illustrate the use of tag invariants by means of the classic Needham-Schroeder-Lowe protocol [43, 39], discussing how we can obtain game-based security guarantees within the logic and guarantee secure composition (Section 2).
- We show that Cryptis is expressive enough to prove the soundness of a *type system* for writing attacker code. The type system demonstrates that the Cryptis guarantees are robust against a rich set of attacker behaviors (Section 3).
- We evaluate the expressiveness of Cryptis by using it to verify several case studies. In addition to the NSL protocol mentioned earlier, we consider a challenge-response authentication protocol based on digital signatures, Diffie-Hellman key exchange, and a multi-mode key-exchange scheme based on TLS 1.3. The case studies demonstrate that the logic can model various cryptographic operations, scale up to protocols with complex features (e.g. branching control flow), and provide rich guarantees such as forward secrecy, even when multiple protocols are running in parallel (Section 4).
- We implemented Cryptis in Coq using the Iris logic [35] and mechanized all our case studies using the Iris proof mode [38] (Section 5).

We discuss related work in Section 6 and conclude in Section 7.

2 CRYPTIS IN A NUTSHELL

To illustrate the basic features of Cryptis, let us see how we can encode a standard example from the literature: the Needham-Schroeder-Lowe public-key protocol [43, 39], or *NSL*, for short. The protocol is used to authenticate two participants, an initiator I and a responder R. We can depict their interaction as follows:

$$I \to R : \{\texttt{"m1"}, n_I, pk_I\}_{pk_R} \qquad R \to I : \{\texttt{"m2"}, n_I, n_R, pk_R\}_{pk_I} \qquad I \to R : \{\texttt{"m3"}, n_R\}_{pk_R}.$$

The initiator generates a fresh cryptographic nonce n_I and sends it to the responder, encrypted with their public key pk_R. The message is tagged with "m1" and includes the initiator's public key pk_I, which serves as its identity. The responder sends n_I back to the initiator along with another fresh nonce n_R. The responder's key pk_R is included in the response so that the initiator can verify the responder's identity (this information was absent in an earlier version of the protocol, which made it vulnerable to a man-in-the-middle attack [39]). The initiator concludes the handshake by sending the nonce n_R back to the responder. In the end, the participants share a fresh secret (n_I, n_R) they can use as a session key.

To model NSL in Cryptis, we use a simple untyped functional language, whose syntax is presented in Figure 1. Most of the language is standard, and includes features such as tuples, tagged unions,

$a \in A$		nonces
$l \in L$		locations
$c \in$ *string*		tags
$\oplus \in \{+, -, \times, =, \ldots\}$		operations
$V \ni k, v ::=$	$n \in \mathbb{Z} \mid a \in A \mid l \in L \mid b \in \mathbb{B} \mid \mathsf{rec}\ f\,x.e \mid (v_1, v_2) \mid ()$	values
	$\mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid \mathsf{ek}(k) \mid \mathsf{dk}(k) \mid \mathsf{enc}(\mathsf{ek}(k), v) \mid \mathsf{tag}(c, k) \mid \cdots$	
$E \ni e ::=$	$x \mid v \in V \mid e_1 \oplus e_2 \mid \neg e \mid e_1\ e_2 \mid !e \mid e_1 \leftarrow e_2 \mid \mathsf{new}(e) \mid (e_1, e_2) \mid \pi_1(e) \mid \pi_2(e)$	expressions
	$\mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{match}(e, e_l, e_r) \mid \mathsf{tag}(c, e) \mid \mathsf{untag}(c, e)$	
	$\mid \mathsf{send}(e) \mid \mathsf{recv}() \mid \mathsf{enc}(e_1, e_2) \mid \mathsf{dec}(e_1, e_2) \mid \mathsf{mknonce}()$	
	$\mid \mathsf{ek}(e) \mid \mathsf{dk}(e) \mid \mathsf{is\_enc\_key}(e) \mid \mathsf{is\_dec\_key}(e) \mid \mathsf{is\_pair}(e) \mid \mathsf{is\_int}(e) \mid \cdots$	

Fig. 1. Syntax (excerpt)

higher-order functions, mutable references, type tests, etc. Its cryptographic core includes operations for generating cryptographic nonces (mknonce), generating encryption and decryption keys from a seed (ek and dk), performing asymmetric encryption and decryption (enc and dec), and tagging and untagging messages. (Later, we'll extend the syntax with other cryptographic operations, such as hashing and Diffie-Hellman exponentiation.) For readability, we'll present examples using a more concise concrete syntax with complex pattern matching, equality patterns, let binding, operations on lists, and some other derived forms shown in Figure 2. All of these extensions can be encoded in the core language. We use the forms tenc and tdec to combine encryption and tagging, since these features are often used together in Cryptis. We'll also write the examples as if the language used fatal exceptions to represent errors, which are in reality managed with the option monad. Errors are used as the outcome of failing operations such as decrypting a message with the wrong key or matching a value with an incompatible pattern.

Figure 3 shows the encoding of the NSL protocol in Cryptis. Each function takes as a parameter the agent's public and secret keys. The initiator also takes the responder's public key pk_R as a parameter, whereas the responder learns the initiator's key during the protocol. In Cryptis, keys are values of the form $\mathsf{ek}(v)$ or $\mathsf{dk}(v)$, which represent encryption or decryption keys generated using the value v as a seed. There are no operations for extracting the seed from a key, but we assume that it is possible for agents to test whether a value is a key using the is_enc_key and is_dec_key functions. The code is a more detailed version of the diagram we've seen above: there are explicit operations for tagging, generating nonces (mknonce), decrypting a value (tdec), and communicating with the network. Note that, as in other protocol models, networking functions do not mention the sender and the receiver of the messages, since this information could be manipulated by an attacker.

Programs in Cryptis run with a small-step call-by-value operational semantics. We elide the complete set of rules, since they are mostly standard. The operational semantics relates *configurations*, which are triples of the form $(\{a_1, \ldots, a_k\}, m, (e_1, \ldots, e_n))$. The first component is a set of nonces, which lists all the nonces a_i that have been generated during execution. It is only used to ensure that mknonce returns a fresh value. The second component is a *memory*, a finite map from locations to values. The third component is a *thread pool*, which contains all threads that are running in the system. To run Cryptis code, we must supply an appropriate definition for the send

$$\lambda x.e \triangleq \mathsf{rec}\ \_\ x.e \qquad \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \triangleq (\lambda x.e_1)\ e_2 \qquad e_1; e_2 \triangleq \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2 \qquad \mathsf{some}(e) \triangleq \mathsf{inr}(e)$$

$$\mathsf{none} \triangleq \mathsf{inl}() \qquad \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(x) \to e_1; \mathsf{inr}(y) \to e_2\ \mathsf{end} \triangleq \mathsf{match}(e, (\lambda x.e_1), (\lambda y.e_2))$$

$$\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 \triangleq \mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{inl}() \to \mathsf{none}; \mathsf{inr}(x) \to e_2\ \mathsf{end}$$

$$\mathsf{tenc}(c, e_1, e_2) \triangleq \mathsf{enc}(e_1, \mathsf{tag}(c, e_2)) \qquad \mathsf{tdec}(c, e_1, e_2) \triangleq \mathsf{untag}(c, \mathsf{dec}(e_1, e_2))$$

$$[e_1; ...; e_n] \triangleq \mathsf{some}(e_1, ..., \mathsf{some}(e_n, \mathsf{none}), ...).$$

Fig. 2. Derived forms

```
let initiator pkI skI pkR =           let responder pkR skR =
  let nI = mknonce () in                let [nI; pkI] = tdec "m1" skR (recv ()) in
  send (tenc "m1" pkR [nI; pkI]);       assert (is_enc_key pkI);
  let [=nI; nR; =pkR] =                 let nR = mknonce () in
    tdec "m2" skI (recv ()) in          send (tenc "m2" pkI [nI; nR; pkR]);
  send (tenc "m3" pkR nR);              assert (recv () == tenc "m3" pkR nR);
  [nI; nR]                              [pkI; nI; nR]
```

Fig. 3. Needham-Schroeder-Lowe protocol

and recv functions, which model the behavior of the attacker. As we'll see later (Section 3), Cryptis ships with a comprehensive type system for making it easier to write such functions.

2.1 The Cryptis Assertion Language

The Cryptis assertion language allows us to specify and verify the behavior of cryptographic protocols such as NSL. The language is a typed λ-calculus, whose syntax is presented in Figure 4, along with some basic proof rules. (For brevity, we only include propositional terms, of type *iProp*. Arguments of function type are indicated with λ binders.) The language features basic connectives of higher-order logic and separation logic, as well as the less standard connectives $\square$ and $\Rrightarrow$, which we'll explain shortly. The form $\mathsf{wp}(e, \varphi)$ is a *weakest precondition* assertion, which states that the expression e can safely execute in the current state and, if it terminates, will produce a result v such that $\varphi(v)$ holds. The form $\mu p.\varphi$ denotes a recursive predicate p; to be well-formed, every recursive occurrence of p in φ must be guarded by the *later modality* $\triangleright$ [42, 4, 16], which rules out inconsistent definitions.

Readers who are unfamiliar with the $\square$ and $\triangleright$ modalities can mostly ignore them. We just note that $\square$ is useful for defining a fragment of *persistent propositions*, which are those that satisfy the equivalence $\varphi \Leftrightarrow \square\varphi$. Intuitively, a proposition is persistent when it does not depend on separation logic resources, such as the points-to predicate $l \mapsto v$. Any proposition of the form $\square\varphi$ is persistent. For persistent propositions, separating conjunction $*$ is equivalent to its more conventional counterpart $\wedge$, and, in particular, they can be freely duplicated, as in intuitionistic logic. Most propositions in Cryptis are persistent, though we'll encounter some important exceptions later. For this reason, we'll often present proof principles in Cryptis using inference-rule notation. As usual, the antecedents of such rules are implicitly conjoined by $\wedge$.

Before we can verify NSL, let us walk through the main additions of the Cryptis logic. First, Cryptis features two predicates to describe cryptographic terms, $\mathsf{term}(v)$ and $\mathsf{pterm}(v)$. The first one simply states that all the nonces in the value v have been allocated at some point during execution. The second one says that v is public, as we've discussed in the Introduction. Cryptis

$$\begin{array}{lr}
\mathit{iProp} \ni \varphi, \psi := p \mid \top \mid \bot \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid \forall x.\varphi \mid \exists x.\varphi \mid a = b & \text{higher-order logic} \\
\quad \mid \varphi * \psi \mid \varphi -\!\!* \psi \mid l \mapsto v \mid \Box\varphi \mid \varphi \Rrightarrow \psi \mid \mathsf{wp}(e, \lambda x.\varphi) & \text{separation logic} \\
\quad \mid \mu x.\varphi \mid \triangleright \varphi & \text{recursion} \\
\quad \mid \mathsf{term}(v) \mid \mathsf{pterm}(v) & \text{terms} \\
\quad \mid \mathsf{enc\_pred}(c, \lambda kv.\varphi) \mid \cdots & \text{tag invariants} \\
\quad \mid \mathsf{meta}(v, c, v') \mid \mathsf{token}(v, C) \mid \cdots & \text{nonces}
\end{array}$$

$$\mathsf{pterm}(v) \Rightarrow \mathsf{term}(v) \qquad \mathsf{term}(v) \Leftrightarrow \Box\,\mathsf{term}(v) \qquad \mathsf{pterm}(v) \Leftrightarrow \Box\,\mathsf{pterm}(v)$$

$$\mathsf{enc\_pred}(s, \varphi) \Leftrightarrow \Box\,\mathsf{enc\_pred}(s, \varphi) \qquad \mathsf{meta}(v, c, v') \Leftrightarrow \Box\,\mathsf{meta}(v, c, v')$$

$$\mathsf{token}(v, C) \wedge c \in C \Rrightarrow \mathsf{meta}(v, c, v') \qquad \mathsf{meta}(v, c, v_1) \wedge \mathsf{meta}(v, c, v_2) \Rightarrow v_1 = v_2$$

$$\mathsf{token}(v, C) \wedge \mathsf{meta}(v, c, v') \wedge c \in C \Rightarrow \mathsf{false} \qquad \mathsf{token}(v, C_1 \uplus C_2) \ast\!\!\ast\ \mathsf{token}(v, C_1) * \mathsf{token}(v, C_2)$$

Fig. 4. Assertion language and rules (excerpt).

actually *defines* these predicates in terms of more basic notions, but for simplicity we'll present them as if they were primitive.

Each proof in Cryptis is parameterized by a series of *tag invariants*. A tag invariant describes under what conditions certain terms satisfy pterm. We focus on tag invariants for encryption, but the full logic contains invariants for key derivation and message authentication as well. Consider the following rule for encrypting tagged terms:

$$\textsc{EncHon}\ \frac{\mathsf{term}(k) \qquad \mathsf{term}(v) \qquad \Box(\mathsf{pterm}(\mathsf{dk}(k)) \Rightarrow \mathsf{pterm}(v)) \qquad \mathsf{enc\_pred}(c, \varphi) \qquad \triangleright\Box\varphi(k, v)}{\mathsf{pterm}(\mathsf{enc}(\mathsf{ek}(k), \mathsf{tag}(c, v)))}$$

The first two premises say that the seed k and the term v must have already been allocated. The third premise, $\Box(\mathsf{pterm}(\mathsf{dk}(k)) \Rightarrow \mathsf{pterm}(v))$, says that that v must be public if $\mathsf{dk}(k)$ is also public. Finally, the last premise says that k and v must satisfy the tag invariant associated with c.

Tag invariants can be expressed by arbitrary user-defined predicates; the only restriction is that we can attach at most one predicate to each tag. In the communication protocols of Section 1, tag predicates guarantee that certain encrypted terms were already public, so that they could be leaked to the network safely. As we will see, for NSL, we will require more sophisticated invariants, which allow the agents to agree on a session key.

In addition to the EncHon rule, which can be used by honest agents, Cryptis provides the following rule for encryption:

$$\textsc{EncPub}\ \frac{\mathsf{pterm}(\mathsf{ek}(k)) \qquad \mathsf{pterm}(v)}{\mathsf{pterm}(\mathsf{enc}(\mathsf{ek}(k), v))}$$

This rule says that it is always possible to encrypt a public term with a public encryption key, and the result of the encryption is always public. Note that this rule does not force the encrypted term to be tagged; intuitively, the attacker is always allowed to send ill-formed inputs to agents, and it is the job of the agents to handle these cases correctly.

$$\textsc{Pure}\ \frac{e \to^*_{\mathsf{pure}} v}{\{\mathsf{true}\}\ e\ \{v'.v' = v\}} \qquad \textsc{Bind}\ \frac{\{\varphi\}\ e\ \{v.\psi\} \qquad \forall v.\ \{\psi\}\ K[v]\ \{v'.\xi\}}{\{\varphi\}\ K[e]\ \{v'.\xi\}} \qquad \textsc{Recv}\ \frac{}{\{\mathsf{true}\}\ \mathsf{recv}()\ \{v.\,\mathsf{pterm}(v)\}}$$

$$\textsc{Send}\ \frac{}{\{\mathsf{pterm}(t)\}\ \mathsf{send}(t)\ \{v.v = ()\}}$$

$$\textsc{Nonce}\ \frac{}{\{\mathsf{true}\}\ \mathsf{mknonce}()\ \{v.\,\mathsf{term}(v) \land \Box(\mathsf{pterm}(v) \Leftrightarrow \triangleright\Box\varphi(v)) \land \mathsf{token}(v, \top)\}}$$

Fig. 5. Select program logic rules

These two rules are the only way of proving that an encrypted message is public, as shown in the following inversion principle:

$$\textsc{Dec}\ \frac{\mathsf{pterm}(\mathsf{enc}(\mathsf{ek}(k), \mathsf{tag}(c, v))) \qquad \mathsf{enc\_pred}(c, \varphi)}{\begin{array}{c}\mathsf{pterm}(\mathsf{ek}(k)) \land \mathsf{pterm}(v) \\ \lor\ \mathsf{term}(k) \land \mathsf{term}(v) \land \Box(\mathsf{pterm}(\mathsf{dk}(k)) \Rightarrow \mathsf{pterm}(v)) \land \triangleright\Box\varphi(k, v)\end{array}}$$

This rule is reminiscent of the use of union types in protocol verification [8, 9]. If we know that some encrypted tagged term v came from the network, and we know which invariant is associated to its tag c, this rule allows us to reason about how this term was encrypted. Intuitively, the first case of the disjunction tells us that the term might have been encrypted by the attacker, whereas the second case tells us that the term was encrypted by an honest agent, who ensured that the tag invariant φ holds.

To get a sense of how these rules work before we reach a more complex example, let us go back to the decryption oracle O' of Section 1:

$$O' \triangleq \mathsf{send}(\mathsf{tdec}(\texttt{"p2"}, \mathsf{dk}(k), \mathsf{recv}())).$$

We've argued that, to prove safety, right before calling send, we need to show that $\mathsf{pterm}(m)$ holds while assuming that

$$\mathsf{pterm}(\mathsf{enc}(\mathsf{ek}(k), \mathsf{tag}(\texttt{"p2"}, m))) \qquad \mathsf{enc\_pred}(\texttt{"p2"}, \lambda km.\,\mathsf{pterm}(m)).$$

We apply the Dec rule and have to consider two cases. In the first case, we learn directly that m is public. In the second case, we learn (modulo the modalities) that m is public via the tag invariant.

2.2 Program Logic

Figure 5 presents select rules of the Cryptis program logic, stated with Hoare triples of the form $\{\varphi\}\ \ e\ \ \{v.\psi\} \triangleq \Box(\varphi \twoheadrightarrow \mathsf{wp}(e, \lambda v.\psi))$. Most rules are inherited from conventional concurrent separation logic, and we focus on the less standard ones. Cryptographic operations usually have no side effects, and we can analyze their behavior simply by executing them: the premise in the Pure rule requires that the expression e reduces to the value v without performing any side effects. The Bind rule allows us to analyze the behavior of a compound expression $K[e]$, formed by plugging in an expression e into a call-by-value context K. The rule says it suffices to replace e by some value v that satisfies e's postcondition.

The only Cryptis-specific forms that have side effects are recv, send and mknonce. The rules for the network primitives recv and send say that terms coming from the network or sent to the network are public. The rule for creating a new nonce is more interesting. It says that mknonce()

$$
\begin{aligned}
I_1(k_R, m_1) &\triangleq \exists n_I, k_I. m_1 = [n_I; \mathsf{ek}(k_I)] \land (\mathsf{pterm}(n_I) \Leftrightarrow \triangleright \mathsf{corruption}(k_I, k_R)) \land \mathsf{pterm}(\mathsf{ek}(k_I)) \\
I_2(k_I, m_2) &\triangleq \exists n_I, n_R, k_R. m_2 = [n_I; n_R; \mathsf{ek}(k_R)] \\
&\qquad \land (\mathsf{pterm}(n_R) \Leftrightarrow \triangleright \mathsf{corruption}(k_I, k_R)) \land \mathsf{meta}(k_R, \texttt{"nsl"}, [k_I; k_R; n_I; n_R]) \\
I_3(k_R, n_R) &\triangleq \forall n_I, k_I.\, \mathsf{meta}(k_R, \texttt{"nsl"}, [k_I; k_R; n_I; n_R]) \Rightarrow \mathsf{meta}(k_I, \texttt{"nsl"}, [k_I; k_R; n_I; n_R]).
\end{aligned}
$$

Fig. 6. Invariants for NSL

returns some value v that is a valid secret term. When the rule is used, we can choose an arbitrary predicate φ to determine under what conditions v is public. For example, we can choose φ to be true to create a public nonce, or choose it to be false. Later (Section 4), we will see that, by varying the choice of φ, we can also express more interesting declassification conditions.

The last component in the rule for nonce generation is token, which is used to associate nonces with *metadata*. We write $\mathsf{meta}(v, c, v')$ to say that the metadata value v' has been permanently associated with the nonce v under the name c. This metadata might represent the protocol where v is being used, the peer that we're trying to contact, or something else. The predicate $\mathsf{token}(v, C)$ means that no metadata has been attached to v thus far under any name $c \in C$. In the rule for nonce generation, C is the set $\top$ of all strings, indicating that no metadata has been saved right after the nonce has been generated. To save a metadata value, as shown in Figure 4, we need to relinquish ownership of $\mathsf{token}(v, C)$ for $c \in C$. The $\varphi \Rrightarrow \psi$ connective that appears in that rule says that it is possible to obtain ψ by giving up on φ and performing an update to *ghost state*. This ghost state cannot be manipulated by the program directly, and is used to formulate the validity of Cryptis assertions. Among other things, it is used to record nonce metadata. Note that $\mathsf{token}(v, C)$ and $\mathsf{meta}(v, c, v')$ are inconsistent with each other when $c \in C$, and that only one metadata value can be associated with a given field at the same time.

2.3 Verifying the NSL Protocol

We now have all the ingredients needed to formulate the correctness theorem for the NSL protocol and verify it. Intuitively, the protocol provides two guarantees: the exchanged nonces are not public (*secrecy*), and the agents agree on their identities and which nonces they exchanged (*authentication*).

Naturally, these guarantees cannot hold if an attacker controls one of the private keys, either because it was leaked or because one of the participants was malicious to begin with. It would be possible to rule out this scenario like we did earlier, by assuming that the private keys are not public (Section 1). However, this would be too restrictive. Indeed, many protocols are designed to provide *forward secrecy guarantees*, which hold even if some keys are compromised by an attacker *after* the protocol is completed. We can model such scenarios because Cryptis is an intuitionistic logic: even if an assertion is not valid at some point in time, it might become valid later during program execution. Rather than assuming preconditions of the form $\neg\, \mathsf{pterm}(v)$, which prevent v from *ever* becoming public, we'll allow the postcondition to detect that a compromise has occurred, as is often done in analyses of forward secrecy [15, 13]. Note that, while NSL does not provide forward secrecy by itself, we will see later that such a specification can be used to provide this guarantee for a protocol that combines NSL with Diffie-Hellman key exchange (Section 4).

To state the theorem, we use the following abbreviations, where k_I and k_R denote the seeds used to generate the keys of the protocol participants:

$$\begin{aligned}\mathsf{corruption}(k_I, k_R) &\triangleq \mathsf{pterm}(\mathsf{dk}(k_I)) \lor \mathsf{pterm}(\mathsf{dk}(k_R))\\ \mathsf{session}(k_I, k_R, n_I, n_R) &\triangleq \mathsf{meta}(k_I, \texttt{"nsl"}, [k_I; k_R; n_I; n_R]) \land \mathsf{meta}(k_R, \texttt{"nsl"}, [k_I; k_R; n_I; n_R])\end{aligned}$$

The first definition expresses that one of the private keys of the agents was compromised during execution. The second definition says that the agents that have associated the nonces n_I and n_R as metadata with their keys. Intuitively, you can read $\mathsf{session}(k_I, k_R, n_I, n_R)$ as saying that a session has been initiated between k_I and k_R using n_I and n_R as nonces.

Theorem 2.1 (Security of NSL; simplified). *Let k_I and k_R be such that*

$$\mathsf{pterm}(\mathsf{ek}(k_I)), \qquad \mathsf{term}(\mathsf{dk}(k_I)), \qquad \mathsf{pterm}(\mathsf{ek}(k_R)), \qquad \mathsf{term}(\mathsf{dk}(k_R)),$$

$$\mathsf{token}(k_I, \{\texttt{"nsl"}\}) * \mathsf{token}(k_R, \{\texttt{"nsl"}\}).$$

Suppose that the invariants $\mathsf{enc\_pred}(\texttt{"m1"}, I_1)$, $\mathsf{enc\_pred}(\texttt{"m2"}, I_2)$ *and* $\mathsf{enc\_pred}(\texttt{"m3"}, I_3)$ *hold, where each I is defined in Figure 6. Then*

$$\{\mathsf{true}\}\ P_I \left\{ v.\ \begin{array}{l} v = \mathsf{none} \lor \exists n_I n_R. v = \mathsf{some}([n_I; n_R]) \\ \land\, \mathsf{pterm}(n_I) \Leftrightarrow \mathsf{pterm}(n_R) \Leftrightarrow \triangleright\, \mathsf{corruption}(k_I, k_R) \\ \land (\mathsf{corruption}(k_I, k_R) \lor \mathsf{session}(n_I, n_R, k_I, k_R)) \end{array} \right\}$$

$$\{\mathsf{true}\}\ P_R \left\{ v.\ \begin{array}{l} v = \mathsf{none} \lor \exists n_I n_R k_I'. v = \mathsf{some}([\mathsf{ek}(k_I'); n_I; n_R]) \\ \land\, \mathsf{pterm}(\mathsf{ek}(k_I')) \\ \land\, \mathsf{pterm}(n_I) \Leftrightarrow \mathsf{pterm}(n_R) \Leftrightarrow \triangleright\, \mathsf{corruption}(k_I', k_R) \\ \land (\mathsf{corruption}(k_I', k_R) \lor \mathsf{session}(n_I, n_R, k_I', k_R)), \end{array} \right\}$$

where

$$P_I \triangleq \mathsf{initiator}(\mathsf{ek}(k_I), \mathsf{dk}(k_I), \mathsf{ek}(k_R)) \qquad P_R \triangleq \mathsf{responder}(\mathsf{ek}(k_R), \mathsf{dk}(k_R))$$

Let us dissect this statement. The premises of the theorem say that the long-term keys of honest and malicious agents have been allocated, and that the encryption keys are public. We also require the tag invariants of Figure 6, which we will explain shortly. The first triple in the conclusion describes the behavior of the initiator. If the initiator terminates successfully with a value v, two things can happen. The value might be none, indicating that the handshake failed because one of the messages contained the wrong nonces or had the wrong format. Otherwise, the initiator returns a pair of nonces $[n_I; n_R]$, which are public if and only if a corruption has occurred. Moreover, if a corruption did *not* occur, the protocol guarantees that the two agents agree on the data that was involved in the handshake. Note that this statement does not allow us to reason about multiple runs of the protocol: once we associate a key with a given session, there is no way of creating a new association, because the rules for meta allow at most one metadata value for a given name (cf. Figure 4). We keep this formulation for simplicity, but our formalization contains a more general statement that allows us to run the protocol multiple times. (Indeed, we'll use this more general statement when analyzing unbounded security games in Section 4.4.) The specification for the responder is similar, but also involves the key of the initiator returned by the handshake.

To prove the correctness of the initiator, we first split the tokens, since we are only going to use $\mathsf{token}(k_I, \{\texttt{"nsl"}\})$ (the other half is used in the proof for the responder). We apply the Nonce rule (Figure 5) to allocate n_I so that $\mathsf{pterm}(n_I) \Leftrightarrow \triangleright\, \mathsf{corruption}(k_I, k_R)$. This means that

```
let game () =
  (* Generate fresh key pairs *)
  let (pkI, skI), (pkR, skR) = mkkey (), mkkey () in

  (* Reveal the public keys to the attacker *)
  send pkI; send pkR;

  (* The attacker decides who the initiator should contact *)
  let pkR' = recv () in

  (* Run the two agents in parallel *)
  let ([nI; nR], [pkI'; nI'; nR']) =
    initiator pkI skI pkR' ||| responder pkR skR
    (* ||| send (tdec "p2" skR (recv ())) *) (* composition *) in

  (* Check who won *)
  if pkR != pkR' && pkI != pkI' then true else
  let m = recv () in
  [pkR; pkI; nI; nR] == [pkR'; pkI'; nI'; nR'] && m != [nI; nR]
```

Fig. 7. Security game for the NSL protocol. The agents win the game if they agree on the handshake parameters and if the attacker cannot guess the session key.

I_1 holds of the initiator's first message $m_1 \triangleq [n_I; \mathsf{ek}(k_I)]$. Moreover, the rules for pterm imply that $\mathsf{pterm}(\mathsf{dk}(k_R)) \Rightarrow \mathsf{pterm}(n_I) \Rightarrow \mathsf{pterm}(m_1)$. Thus, m_1 can be safely encrypted with $\mathsf{ek}(k_R)$ under the tag "m1" and sent to the network.

Now, consider what happens when the initiator receives a response m_2. If any checks fail during execution, the initiator returns none, and the postcondition trivially holds. Otherwise, we reach the point where it is about to send n_R back. By Dec, we need to consider two cases to prove that the message can be safely sent. The first case is that the encrypted contents in m_2 are public. But this implies that n_I and n_R are public, since they are contained in m_2. By the definition of n_I, we learn that a corruption occurred. Moreover, n_R can be freely sent over the network by using the rule EncPub. We conclude by showing that the postcondition follows from the assumptions.

The second case is that the invariant I_2 holds. By definition, this implies that $\mathsf{pterm}(n_R)$ holds if and only if $\triangleright\, \mathsf{corruption}(k_I, k_R)$, and also that $\mathsf{meta}(k_R, \texttt{"nsl"}, [k_I; k_R; n_I; n_R])$ holds. We consume k_I's token to obtain $\mathsf{meta}(k_I, \texttt{"nsl"}, [k_I; k_R; n_I; n_R])$. The two metadata assertions can be combined to establish $\mathsf{session}(k_I, k_R, n_I, n_R)$ and to establish I_3. This, combined with the fact that $\mathsf{pterm}(\mathsf{dk}(k_R)) \Rightarrow \mathsf{pterm}(n_R)$, allows us to send n_R back and to prove the postcondition, thus concluding the proof. The reasoning for proving the case for the responder is analogous.

2.4 Adequacy and Game-based Security

The interest of the security statements that we have proven so far is that they inherit the composition principles of the Cryptis logic, allowing us to reuse them when proving more complex results. But Cryptis is a non-standard logic, which can make its formulations of secrecy and authentication hard to interpret. To help bridge this gap, we show Cryptis can be used to derive self-contained security results that mention only the operational semantics of the Cryptis language, which is much more conventional.

Let us consider the security game of Figure 7. (For now, ignore the commented line marked with "composition"; we'll come back to it in Section 2.5.) The game consists of an initiator and a responder running in parallel, using two long-term keys that have been freshly generated (internally, key generation is defined in terms of nonce generation). The two public keys pkI and pkR are leaked to the attacker, whereas the corresponding secret keys aren't. Moreover, the peer contacted by the initiator is chosen by the network. The attacker wins the game if the test at the end of the function fails; otherwise, the agents win. The test says that, if one of the contacted peers was an honest agent, then the participants agree on all the parameters established by the protocol—an *authentication guarantee.* Moreover, in this case, the attacker cannot guess the exchanged nonces.

There are many attacks that could be attempted on the game. For instance, Lowe [39] found a man-in-the-middle attack on the original Needham-Schroeder protocol where the initiator attempts to contact a malicious agent, and the agent forwards the request to another responder. By manipulating the messages between the initiator and responder, the attacker manages to learn the exchanged nonces and to trick the responder into thinking that it was contacted by the initiator. This could cause the agents to lose the game, either because their nonces appeared in clear text, or because $pk_I \neq pk'_I$. As Lowe found out, the root of the issue was that the second message of the original protocol did not mention the sender's identity, so it was not possible for the initiator to determine who they were talking to. Fortunately, we are going to show that Lowe's fix is enough to guarantee security.

The first step is to prove a specification of the form

$$\{I\}\ \mathsf{game}()\ \{v.v = \mathsf{true}\},$$

where I contains all the tag invariants required by the variants of NSL, as stated in Theorem 2.1, plus some other invariants that we elide here, which are used by mkkey to generate fresh keys. We prove this result by allocating the key pairs (pk_I, sk_I) and (pk_R, sk_R) so that the encryption keys are public and the decryption keys satisfy $\Box\neg\,\mathsf{pterm}(sk_I)$ and $\Box\neg\,\mathsf{pterm}(sk_R)$. (Formally, this relies on generating the keys with nonces that cannot become public, and on the invariants used by mkkey.) These properties mean that we do not allow key compromises in this game, unlike in Theorem 2.1, where we did not want to rule out this possibility. (We will see how to lift this restriction in Section 4.)

After allocating and distributing the keys, we analyze the execution of the agents by using the Par rule of separation logic, which is valid in Cryptis:

$$\textsc{Par}\quad \frac{\{\varphi_1\}\ e_1\ \{v_1.\psi_1\} \qquad \{\varphi_2\}\ e_2\ \{v_2.\psi_2\}}{\{\varphi_1 * \varphi_2\}\ e_1 \,|||\, e_2\ \{(v_1, v_2).\psi_1 * \psi_2\}}.$$

The Par rule says that we can execute two processes in parallel provided that their pre- and postconditions use disjoint resources, as expressed by the separating conjunction $*$. In this security game, these resources are the tokens $\mathsf{token}(k_I, \{\texttt{"nsl"}\})$ and $\mathsf{token}(k_R, \{\texttt{"nsl"}\})$ mentioned in the proof of Theorem 2.1.

Now, after both agents stop running, we arrive at the test $pk_I = pk'_I \vee pk_R = pk'_R$. If the test fails, the game returns true, and we are done. Otherwise, suppose that $pk_I = pk'_I$. By the correctness result for the initiator and the responder, we know that there hasn't been a key compromise (since we allocated sk_I and sk_R so that a compromise would be impossible). Therefore, we find that

$$\mathsf{session}(k_I, k'_R, n_I, n_R) \qquad\qquad \mathsf{session}(k_I, k_R, n'_I, n'_R).$$

By unfolding these predicates and using the rules of Figure 4, we find that $n_R = n'_R$, $k_I = k'_I$ and $k_R = k'_R$. Moreover, we know that $\mathsf{pterm}(m)$ holds (because it originated from the network), we cannot have $m = [n_I; n_R]$, since those terms are only public if a compromise has occurred (which, as we've already argued, is impossible). The other case, $pk_R = pk'_R$, is similar, and we elide its proof.

After proving this triple, the second step is to combine it with the *adequacy* theorem for the Cryptis logic. Roughly speaking, adequacy allows us to prove a metatheoretic property of the result of a program p if this metatheoretic property is implied by the validity of p's postcondition. Since this postcondition is a simple equality, we conclude that the same equality holds at the meta-level.

Theorem 2.2. *For any valid attacker, if* game() *terminates, it returns true.*

Proofs for more complex games would follow the same pattern. For instance, we might add a phase at the beginning of the game where the attacker is allowed to run some arbitrary (but valid) setup code (cf. Section 3). The validity of the attacker would imply that that its execution wouldn't interfere with the rest of the game, and we would arrive to the same conclusion.

2.5 Compositional Verification

To conclude this tour of Cryptis, let us consider how we can obtain guarantees for composite protocols. Suppose that we add the decryption oracle from Section 1 as another component running in parallel to the game of Figure 7:

```
...
let ([nI; nR], [pkI'; nI'; nR'], _) =
  initiator pkI skI pkR' ||| responder pkR skR
  ||| send (tdec "p2" skR (recv ())) (* composition *)
in
...
```

To prove security in this scenario, we first prove the following easy specification for O', which follows the informal outline presented in Section 1:

$$\{\mathsf{term}(sk_R) \wedge \mathsf{enc\_pred}(\texttt{"p2"}, \lambda km.\, \mathsf{pterm}(m))\}\ O'\ \{\mathsf{true}\},$$

where $O' \triangleq \mathsf{send}(\mathsf{tdec}(\texttt{"p2"}, sk_R, (\mathsf{recv}())))$. Then, we can easily adapt the proof of correctness of Section 2.4 by including this tag invariant in its precondition. Once again, the Par rule guarantees that the processes run without interference, and that the game is secure.

It might seem that this proof is hardly doing anything. But this is precisely the point: thanks to tag invariants, the rules of concurrent separation logic are all we need to determine when concurrent execution is safe in the Cryptis model.

3 MODELING THE ATTACKER

The statement of game security for NSL (Theorem 2.2) underscores that Cryptis can only provide protection against *valid* attackers, by which we mean any instantiation of the send and recv functions that validates the proof rules in Figure 5. Naturally, these guarantees would be useless if valid attackers were too weak compared to the threats that protocols might encounter in practice. This section aims to increase our confidence in Cryptis by demonstrating the scope of its attacker model.

We layer a simple type system for writing attacker code on top of the Cryptis programming language. The syntax, presented in Figure 8, includes types for public terms, encryption and decryption keys, products, sums, higher-order functions and references. Thanks to the expressiveness of Cryptis, it wouldn't be difficult to extend this system with richer features, such as polymorphism, recursive types and concurrency. However, we stick to the current formulation for simplicity.

Each type τ is interpreted as a value predicate $[\![\tau]\!]_v$. (Note that tag invariants are not used in this definition, since their sole purpose is to provide guarantees to honest agents.) We lift types to

$$\begin{aligned}
\mathsf{Type} \ni \tau, \sigma &:= \mathsf{Pub} \mid \mathsf{EK} \mid \mathsf{DK} \mid () \mid \tau \times \sigma \mid \tau + \sigma | \tau \to \sigma \mid \mathsf{Ref}\,\tau \mid \mathsf{List}\,\tau \mid \mathsf{string} \mid \mathbb{Z} \\
\mathbb{B} &\triangleq () + () \\
\mathsf{option}\,\tau &\triangleq () + \tau
\end{aligned}$$

$$\begin{aligned}
[\![-]\!]_v &: \mathsf{Type} \to V \to \mathit{iProp} \\
[\![-]\!]_e &: \mathsf{Type} \to E \to \mathit{iProp} \\
[\![\mathsf{Pub}]\!]_v(v) &\triangleq \mathsf{pterm}(v) \\
[\![\mathsf{EK}]\!]_v(v) &\triangleq \exists k. v = \mathsf{ek}(k) \land \mathsf{pterm}(v) \\
[\![\mathsf{DK}]\!]_v(v) &\triangleq \exists k. v = \mathsf{dk}(k) \land \mathsf{pterm}(v) \\
[\![()]\!]_v(v) &\triangleq (v = ()) \\
[\![\tau \times \sigma]\!]_v(v) &\triangleq \exists v_1, v_2. v = (v_1, v_2) \land [\![\tau]\!]_v(v_1) \land [\![\sigma]\!]_v(v_2) \\
[\![\tau + \sigma]\!]_v(v) &\triangleq (\exists v', v = \mathsf{inl}(v') \land [\![\tau]\!]_v(v')) \lor (\exists v', v = \mathsf{inr}(v') \land [\![\sigma]\!]_v(v')) \\
[\![\tau \to \sigma]\!]_v(v) &\triangleq \Box(\forall v' \in V, [\![\tau]\!]_v(v') \Rightarrow [\![\sigma]\!]_e(v\ v')) \\
[\![\mathsf{Ref}\,\tau]\!]_v(v) &\triangleq \exists l \in L. v = l \land \boxed{\exists v', l \mapsto v' \land [\![\tau]\!]_v(v')} \\
[\![\mathsf{List}\,\tau]\!]_v(v) &\triangleq \exists v_1, ..., v_n. v = [v_1; ...; v_n] \land [\![\tau]\!]_v(v_1) \land \cdots \land [\![\tau]\!]_v(v_n) \\
[\![\mathsf{string}]\!]_v(v) &\triangleq \exists s \in \mathsf{string}, v = s \\
[\![\mathbb{Z}]\!]_v(v) &\triangleq \exists n \in \mathbb{Z}, v = n \\
[\![\tau]\!]_e(e) &\triangleq \mathsf{wp}(e, v.[\![\tau]\!]_v(v))
\end{aligned}$$

$$\Gamma \vdash e : \tau \triangleq \Box(\forall \vec{v}, (\forall x \in \Gamma, [\![\Gamma(x)]\!]_v(\vec{v}(x))) \Rightarrow [\![\tau]\!]_e(e[\vec{v}])).$$

Fig. 8. Attacker type system

predicates over expressions by using weakest preconditions: $[\![\tau]\!]_e$ consists of expressions e that evaluate to a value in $[\![\tau]\!]_v$. Most clauses of the definition are straightforward. The interpretation of functions uses the $\Box$ modality to ensure that the semantics of types is persistent. The $\boxed{\cdots}$ form in the interpretation of reference types denotes an *invariant*, a property that is forced to hold at all moments during execution. Thus, the clause is simply saying that the contents of the reference must always be of the appropriate type. By unfolding definitions, we see that providing the send and recv functions is equivalent to providing a pair of functions of types Pub $\to$ () and () $\to$ Pub.

The rules of the type system is mostly standard. A judgment $\Gamma \vdash e : \tau$, defined in Figure 8, means that the expression e produces a result in τ when plugged in with values of types given in Γ. We omit most rules, as they are standard, but include the types of primitives for manipulating terms in Figure 9. Intuitively, these types say that all the term primitives of the Cryptis language can be restricted to public terms. The functions ek and dk are used to create encryption and decryption keys from a seed. The functions term_of_ek and term_of_dk behave as the identity, and are used for coercing keys into terms (alternatively, we could have provided the system with a notion of subtyping). Going the other direction, there are several functions for coercing cryptographic terms into other types, such as ek_of_term. These functions are not part of the primitives of Section 2, but can be easily defined with them. For instance,

$$\mathsf{ek\_of\_term} \triangleq \lambda x.\mathsf{if}\ \ \mathsf{is\_enc\_key}(x)\ \mathsf{then}\ \ \mathsf{some}(x)\ \mathsf{else}\ \ \mathsf{none}\,.$$

$$
\begin{aligned}
\mathsf{mknonce} &: () \to \mathsf{Pub} & \mathsf{term\_of\_list} &: \mathsf{List}(\mathsf{Pub}) \to \mathsf{Pub} \\
\mathsf{list\_of\_term} &: \mathsf{Pub} \to \mathsf{option}(\mathsf{List}(\mathsf{Pub})) & \mathsf{term\_of\_int} &: \mathbb{Z} \to \mathsf{Pub} \\
\mathsf{int\_of\_term} &: \mathsf{Pub} \to \mathsf{option}\,\mathbb{Z} & \mathsf{hash} &: \mathsf{Pub} \to \mathsf{Pub} \\
\mathsf{ek} &: \mathsf{Pub} \to \mathsf{EK} & \mathsf{dk} &: \mathsf{Pub} \to \mathsf{DK} \\
\mathsf{term\_of\_ek} &: \mathsf{EK} \to \mathsf{Pub} & \mathsf{term\_of\_dk} &: \mathsf{DK} \to \mathsf{Pub} \\
\mathsf{ek\_of\_term} &: \mathsf{Pub} \to \mathsf{option}\,\mathsf{EK} & \mathsf{dk\_of\_term} &: \mathsf{Pub} \to \mathsf{option}\,\mathsf{DK} \\
(=) &: \mathsf{Pub} \to \mathsf{Pub} \to \mathbb{B} & \mathsf{group} &: \mathsf{Pub} \to \mathsf{Pub} \\
(\widehat{\ }) &: \mathsf{Pub} \to \mathsf{Pub} \to \mathsf{Pub} & \mathsf{enc} &: \mathsf{EK} \to \mathsf{Pub} \to \mathsf{Pub} \\
\mathsf{dec} &: \mathsf{DK} \to \mathsf{Pub} \to \mathsf{Pub} & \mathsf{tag} &: \mathsf{string} \to \mathsf{Pub} \to \mathsf{Pub} \\
\mathsf{untag} &: \mathsf{string} \to \mathsf{Pub} \to \mathsf{option}\,\mathsf{Pub} & \mathsf{fork} &: (() \to ()) \to () \\
\mathsf{mkchan} &: () \to (() \to \mathsf{Pub}) \times (\mathsf{Pub} \to ()).
\end{aligned}
$$

Fig. 9. Term manipulation functions

For ease of reference, we also include some operations that were omitted from the syntax in Figure 1. The function hash denotes a non-invertible, collision-resistant hash function. The function group takes in a seed as an argument and returns a generator of some Diffie-Hellman group. The operation $\widehat{\ }$ denotes exponentiation in such a group, which satisfies the equation

$$\mathsf{group}(g)\,\widehat{\ }\,x\,\widehat{\ }\,y = \mathsf{group}(g)\,\widehat{\ }\,y\,\widehat{\ }\,x.$$

The fork function forks off a new thread. Finally, the function mkchan is used for creating a communication channel. Its result is a pair of functions for reading terms from the channel and sending them. This function is not a primitive of the language, but rather implemented using a lock and a reference to store a bag of sent terms.

To illustrate the type system, consider the code in Figure 10. The code shows an attacker function that attempts to exploit the decryption oracle discussed in Section 1. The attacker creates two channels to set up the network and returns them to the agents. In a forked-off thread, it attempts to forward Alice's message to the oracle and read the oracle's leaked message. However, the tags allow the oracle to detect and prevent the leak. It would also be possible to write more sophisticated attacks using the language as well, such as Lowe's attack on the Needham-Schroeder protocol [39].

4 CASE STUDIES

We have evaluated the use of Cryptis on four case studies (cf. Figure 11). The case studies demonstrate that the Cryptis logic is capable of handling various features that are commonly used in real-world protocols, such as digital signatures, Diffie-Hellman key exchange, branching control flow and loops. Our model of Diffie-Hellman terms illustrates that the logic can cover rich security properties such as forward secrecy. Moreover, thanks to its simple composition principles, we can reuse security proofs to analyze the behavior of multiple protocols running together, something that can be challenging for automated tools [13, 15, 10]. For example, while ProVerif can take several hours to check a combined model of TLS 1.2 and 1.3 [15], Cryptis can check the proof of our combined models running together in about a minute.

```
let attacker ()
  : (Pub -> () * () -> Pub) =
  (* Create channels *)
  (* Written by agents, read by attacker *)
  let c1 = mkchan () in
  (* Read by agents, written by attakcer *)
  let c2 = mkchan () in

  fork (fun () ->
    (* Receive Alice's message *)
    let m1 = fst c1 () in
    (* Relay message to Bob *)
    snd c2 m1;
    (* Receive Bob's decrypted message *)
    let s = fst c1 () in
    snd c2 s);

  (snd c1, fst c2)
```

```
let game () =
  let (send, recv) = attacker () in
  let s = mknonce () in
  let (pkB, skB) = mkkey () in
  let alice () = send (tenc "p1" pkB s) in
  let bob () = tdec "p1" skB (recv ()) in
  let oracle () =
    send (tdec "p2" skB (recv ())) in
  (alice () ||| bob () ||| oracle ());
  let guess = recv () in
  s != guess
```

Fig. 10. Attacker that tries to exploit the oracle from Section 1

Case study	Exercised Features	Reference
Challenge/Response	Authentication with public signatures	Section 4.1
NSL+DH	Diffie-Hellman key exchange and forward secrecy	Section 4.2
TLS 1.3	Multi-mode authentication; control flow	Section 4.3
Combined	Composition; loops	Section 4.4

Fig. 11. Case studies

4.1 Digital Signatures

As a warm up, we consider how to model digital signatures. Cryptis does not have a dedicated mechanism for expressing signatures; rather they are simply implemented using asymmetric cryptography, by swapping which keys are public and secret. The simplest scheme is to use encryption to sign a message and decryption to verify it (although variations are also possible, such as signing a hash of the message and sending the signature along with the message in clear text). This idea is illustrated in the following challenge/response protocol:

$$I \to R : n_I, \mathit{vk}_I \qquad R \to I : \{\texttt{"cr2"}, n_I, n_R, \mathit{vk}_I\}_{\mathit{sk}_R} \qquad I \to R : \{\texttt{"cr3"}, n_I, n_R, \mathit{vk}_R\}_{\mathit{sk}_I}.$$

The initiator starts out by sending a fresh nonce n_I in clear text, along their verification key vk_I. The responder then generates a fresh nonce of their own, signs the message, and sends it back to the initiator. The initiator acknowledges by signing the contents of the message and sending them back. The control flow is similar to the NSL protocol (Section 2). The main difference with respect to the earlier setting is in the last message: since the nonces are public, an attacker might forge the first message to the responder, and the initiator needs to confirm that they meant to initiate a connection with those parameters.

The Cryptis formalization of this protocol is similar to NSL, except that only the second and third messages carry invariants, since the first one is sent in clear text. Its specification is also similar to what we've seen earlier, but does not guarantee that the exchanged nonces are secret.

4.2 Diffie-Hellman Key Exchange and Forward Secrecy

One of the cornerstones of modern communication protocols is *Diffie-Hellman key exchange*. In its simplest form, this handshake can be depicted as follows.

$$I \rightarrow R : g^a, I, R \qquad\qquad R \rightarrow I : g^a, g^b, I, R.$$

In words, the initiator of the protocol generates a fresh nonce and uses it to compute a key share g^a. Here, g denotes some generator of a previously agreed upon group, whereas the superscript denotes exponentiation. The responder replies including another key share of its own. In the end, the two parties compute the session key $g^{ab} = (g^a)^b = (g^b)^a$.

This handshake is useful because it guarantees, under standard hardness assumptions, that the session key g^{ab} remains secret even if the attacker has access to the transcript of the communication between the two parties, thus making it harder for attackers to eavesdrop a conversation.

By itself, Diffie-Hellman key exchange does not provide any authenticity guarantees. However, it can be combined with other primitives to circumvent this limitation. One possibility is to require each party to sign their messages, using a variant of the protocol of Section 4.1. Another possibility, which we explore here, is to encapsulate the exchange inside of a handshake based on public-key encryption such as NSL. This process works as follows:

$$I \rightarrow R : \{\texttt{"m1"}, g^a, pk_I\}_{pk_R} \qquad R \rightarrow I : \{\texttt{"m2"}, g^a, g^b, pk_R\}_{pk_I} \qquad I \rightarrow R : \{\texttt{"m3"}, g^b\}_{pk_R}.$$

In the original NSL, revealing the long-term secret key of any agent is enough to ruin the confidentiality guarantees of the protocol, because the attacker can decrypt the exchanged messages and learn the session key. However, when NSL is combined with Diffie-Hellman key exchange, this is no longer possible: once the handshake completes, the exchanged messages are not useful to the attacker. In the literature, this guarantee is often known as *forward secrecy*. Internally, we can state forward secrecy by modifying the specification of NSL (Theorem 2.1). We provide here a simplified statement for the initiator of the protocol that focuses on secrecy, but the result carries over to the responder and is capable of providing authentication guarantees as well.

Theorem 4.1 (Correctness for NSL+DH; initiator). *Using similar notations and assumptions as in Theorem 2.1, we can show the following specification, where P_I^{DH} denotes the code of the initiator of Diffie-Hellman key exchange running on NSL.*

$$\{\mathsf{true}\}\ P_I^{DH} \left\{ v.\ \begin{array}{c} v = \mathsf{none} \vee \exists n_I n_R. v = \mathsf{some}(k) \wedge \mathsf{term}(k) \\ \wedge (\mathsf{corruption}(k_I, k_R) \vee \Box(\mathsf{pterm}(k) \Rightarrow \triangleright\, \mathsf{false})) \end{array} \right\}$$

The proof of this theorem is similar to the one of Theorem 2.1. (As a matter of fact, both results are corollaries of a similar theorem for a more general version of NSL, which is the one we proved in our Coq formalization.) The main difference is that the message invariants are slightly modified so that, when the handshake completes without compromise, we can guarantee that each participant owns key shares of the form g^a and g^b, where each share has been freshly generated by one agent. In this case, the term that is returned by P_I^{DH} is g^{ab} and, to conclude the proof, we just need to show $\Box(\mathsf{pterm}(g^{ab}) \Rightarrow \triangleright\, \mathsf{false})$.

To complete this step, let us see more closely how Diffie-Hellman terms are modeled in Cryptis (Figure 12). Values of the form $\mathsf{dh}(g, \{v_1; \ldots; v_n\})$ represent a Diffie-Hellman power $g^{v_1 \cdots v_n}$, with the exponents $v_1, \ldots, v_n$ being treated as a multiset, up to permutation. Note that dh is not part of the syntax of expressions, only of *values*: to construct a Diffie-Hellman term, programs can use $\mathsf{group}(g)$, which reduces to $\mathsf{dh}(g, \emptyset)$, or they can compute an exponential $v \mathbin{\hat{}} v'$, which reduces to $\mathsf{dh}(g, \{v'\} \cup \vec{v})$ if v is of the form $\mathsf{dh}(g, \vec{v})$; otherwise, it reduces (arbitrarily) to zero, to signal an invalid operation. As hinted earlier (Section 3), this means that $\mathsf{group}(g) \mathbin{\hat{}} a \mathbin{\hat{}} b$ and $\mathsf{group}(g) \mathbin{\hat{}} b \mathbin{\hat{}} a$

$$\begin{aligned} \oplus &\in \{\ldots, \hat{}\,, \ldots\} \\ V \ni v &\triangleq \cdots \mid \mathsf{dh}(v, \{v_1, \ldots, v_n\}) \mid \cdots \\ E \ni e &\triangleq \cdots \mid \mathsf{group}(e) \mid \cdots \\ \mathit{iProp} \ni \varphi &\triangleq \cdots \mid \mathsf{dh\_pred}(v, v') \mid \cdots \end{aligned}$$

$$\mathsf{dh\_pred}(v, \varphi) \Leftrightarrow \Box\, \mathsf{dh\_pred}(v, \varphi) \qquad \frac{}{\{\mathsf{true}\}\ \mathsf{mknonce}()\ \{v.\cdots \wedge \Box \forall v', \mathsf{dh\_pred}(v, v') \Leftrightarrow \psi(v')\}}\ \textsc{Nonce}$$

$$\frac{\mathsf{pterm}(g) \qquad \forall i \in \{1, \ldots, n\}, \mathsf{dh\_pred}(v_i, \mathsf{dh}(g, \{v_1, \ldots, v_n\}))}{\mathsf{pterm}(\mathsf{dh}(g, \{v_1, \ldots, v_n\}))}\ \text{DH-}\textsc{Release} \qquad \frac{\mathsf{pterm}(g)}{\mathsf{pterm}(\mathsf{dh}(g, \emptyset))}\ \text{DH-}\textsc{Group}$$

$$\frac{\mathsf{pterm}(\mathsf{dh}(g, \{v_1, \ldots, v_n\})) \qquad \mathsf{pterm}(v_{n+1})}{\mathsf{pterm}(\mathsf{dh}(g, \{v_1, \ldots, v_n, v_{n+1}\}))}\ \text{DH-}\textsc{Exp}$$

Fig. 12. Support for Diffie-Hellman terms in Cryptis

reduce both to the same value $\mathsf{dh}(g, \{a, b\})$, guaranteeing that the equational properties of Diffie-Hellman terms are modeled correctly.

As shown in Figure 12, Cryptis describes which Diffie-Hellman terms can be released by means of a predicate $\mathsf{dh\_pred}(v, v')$. In the extended Nonce rule, the parameter ψ controls which Diffie-Hellman terms that mention the nonce v can be made public. As shown in the DH-Release rule, such a term becomes public if it satisfies the predicates of all of its nonces. The DH-Group and DH-Exp rules say that the basic Diffie-Hellman operations are compatible with public terms, which allows us to ascribe the types shown in Figure 9. These are the only three rules that can be applied to show that such terms are secret.

To obtain forward secrecy, we allocate the nonces a and b in the initiator and the responder so that, for $n \in \{a, b\}$,

$$\mathsf{pterm}(n) \Leftrightarrow \triangleright\, \mathsf{false} \qquad \mathsf{dh\_pred}(n, \mathsf{dh}(g, \vec{v})) \Leftrightarrow \triangleright(|\vec{v}| = 1 \wedge \mathsf{corruption}(k_I, k_R)).$$

If we conclude the protocol without corruption, we know that $\mathsf{dh}(g, \{a\})$ and $\mathsf{dh}(g, \{b\})$ were generated by honest agents and were allocated as shown above. We show that $\mathsf{pterm}(\mathsf{dh}(g, \{a, b\}))$ is absurd by considering all the pterm rules for Diffie-Hellman terms. The rules DH-Release and DH-Group do not apply, since the term has two exponents. Therefore, the rule DH-Exp must have been used. But this would imply either $\mathsf{pterm}(a)$ and $\mathsf{pterm}(b)$, which contradicts how these terms have been allocated. In sum , $\mathsf{dh}(g, \{a, b\})$ cannot be public if the protocol completed successfully.

Alternatively, we could also have expressed forward secrecy externally, as a security game. The game, shown in Figure 13, is a variant of Figure 7 that reveals the secret keys of the participants after the handshake is over.

Theorem 4.2. *For any valid attacker, if the game of Figure 13 successfully terminates, it returns true.*

The proof of this statement is similar to the one of Theorem 2.2, but with a twist: since we are leaking the keys after the handshake, we cannot allocate them in such a way that prevents them from ever being made public. Instead, we establish the following equivalences:

$$\mathsf{pterm}(k_I) \Leftrightarrow \mathsf{pterm}(k_R) \Leftrightarrow \triangleright\, \mathsf{meta}(k_I, \texttt{"pub"}, ()),$$

```
let game () =
  let (pkI, skI), (pkR, skR) = mkkey (), mkkey () in
  send pkI; send pkR;
  let pkR' = recv () in
  let (gab, [pkI'; gab']) = dh_initiator pkI skI pkR' ||| dh_responder pkR skR  in
  if pkR != pkR' && pkI != pkI' then true else
  send skI; send skR; (* <-- leak secret keys after handshake *)
  let m = recv () in
  [pkR; pkI; gab] == [pkR'; pkI'; gab'] && m != gab
```

Fig. 13. Security game for the NSL protocol combined with DH key exchange

where k_I and k_R are the seeds used for the participants' keys. We split k_I's token to obtain $\mathsf{token}(k_I, \{\texttt{"nsl"}\})$ and $\mathsf{token}(k_I, \{\texttt{"pub"}\})$. The first one is used to analyze the parallel execution of the two agents. When the handshake completes, the second one rules out the possibility of early corruption: the secret keys can only be made public if their seeds are public. And thanks to how we allocated these terms, corruption is only possible when $\mathsf{meta}(k_I, \texttt{"pub"}, ())$ holds, which is impossible because we still own the resource $\mathsf{token}(k_I, \{\texttt{"pub"}\})$. Since no corruption occurred, we conclude that the secrecy of g^{ab} is guaranteed. At this point, we can trade in $\mathsf{token}(k_I, \{\texttt{"pub"}\})$ for $\mathsf{meta}(k_I, \texttt{"pub"}, ())$, which has the effect of making the decryption keys sk_I and sk_R public and safe to be sent over the network. We conclude by invoking the secrecy of g^{ab} and the authentication guarantees of NSL.

4.3 TLS 1.3

Many real-world protocols can be instantiated with a wide range of parameters. For example, when a client wants to connect to a server using the TLS protocol, they attempt to negotiate with the server a series of options such as the version of the protocol, which cryptographic algorithms will be used, etc. This sheer variety of options can be challenging to model, let alone verify, and has been a source of vulnerabilities in the past. For example, Logjam [2] is a man-in-the-middle attack where a client is tricked into using a weak Diffie-Hellman group to communicate with a server, even though both the server and the client support stronger groups. The attacker intercepts the client's request and forwards it to the server while downgrading the parameters to use a weak Diffie-Hellman group instead of a strong one. If the server accepts the connection, its reply does not contain sufficient information to let the client realize that the weak group is being used, thereby enabling the attack.

In this case study, we are interested in evaluating how Cryptis scales up to handle such complex designs, which feature various options and branching control flow. The case study models the handshake of TLS 1.3, which combines three authentication methods: one that uses on a pre-shared key, one that uses a Diffie-Hellman key exchange, and one that combines the previous two. The client chooses the method and sends it in their hello message, along with other connection parameters. If the server decides to accept the connection, it sends back a signed confirmation to the client along with other parameters that might be required by the method proposed by the client. The client finishes the handshake by sending back an acknowledgment to the server authenticated with the agreed session key. In all these exchanges, server and client ensure that they agree on the negotiated method and parameters. Besides nonces and key shares, the exchanged parameters play no major role in our model; we include them simply to demonstrate that the protocol allows the participants to agree on them. For simplicity, we do not model some of the more complex aspects of the handshake, such as letting the client send early data along with its hello message, or authenticate itself digital signatures, though we believe that the model could be extended to encompass those features as well.

The correctness statement for our model is similar to the previous ones, but the security guarantees depend on the authentication method chosen by the client. From the client's perspective, authentication can only fail if both the server's signing key and the pre-shared key are compromised. Moreover, if this is not the case, then the established session key satisfies forward secrecy if the client decided to use a Diffie-Hellman key exchange. The guarantees are almost the same from the perspective of the server, except that they compromising the pre-shared key is enough to break them (since the client does not authenticate via digital signatures).

This is by far the most complex of our case studies: the whole development takes about 2k lines of code. The most challenging part of the proof was correctly manipulating, checking, and reasoning about the different message formats used in each mode of the handshake, something that was not needed for the other protocols. We handled this complexity by decomposing the proof into several auxiliary lemmas, showing that each function in the protocol definition was a faithful implementation of a higher-level, pure mathematical function defined on structured data types. This decomposition allowed us to reason about the handshake invariants independently of the Cryptis program logic, thus simplifying the proof.

4.4 Putting Everything Together

In our last case study, we evaluate compositional reasoning in Cryptis by verifying a system that combines the previous three protocols. Our goal is to prove the security of the game shown in Figure 14. The game illustrates the security guarantees of the TLS handshake: we want to show that the session key produced by the handshake is secure even after running an arbitrary number of sessions of the protocol, and even after leaking the long term keys of the client and the server (but not the initial pre-shared key). The environment function spawns off four threads running the protocols of Sections 4.1 and 4.2 in a loop, and an additional thread that continuously runs the TLS server of Section 4.3. The tls_client function initiates a series of TLS handshakes with the server. When the client and the server successfully complete a handshake, they use they established session key as the pre-shared key for the next request, a behavior that mimics the so-called *resumption* feature of TLS. The client stops running when the network tells it to do so. When this happens, it returns *sk*, the key of its last session. The agents win the game if the attacker is unable to guess *sk*.

The proof of this protocol is follows the same strategy as the game-based proofs we've seen earlier, combined with some invariant reasoning to handle the looping threads. We use the proof of correctness for the TLS client (Section 4.3) to maintain an invariant saying that that the current session key is secret if we assume that the pre-shared key is also secret. As we've argued earlier, to compromise the TLS handshake, an attacker needs to compromise both the server's long-term key and its pre-shared key, which is why the game is secure even after leaking the long-term key. Reasoning about the other protocols is simple, since the game is not assessing their security guarantees. All that matters is that they can safely run using tag invariants that are disjoint from those used by TLS.

5 IMPLEMENTATION

We implemented the Cryptis logic as a library in the Coq proof assistant [49], using the Iris framework [35]. Iris allows defining expressive concurrent separation logics, with support for higher-order ghost state, invariants, resource algebras, prophecy variables [36], and more. Cryptis inherits those features from Iris, and since they are orthogonal to the reasoning patterns supported by Cryptis, it is possible to compose protocols with other concurrent programs and reason about their behavior without compromising the soundness of the logic. Though the model of Iris is quite

```
let game () =
  let (ekI, dkI), (ekR, dkR) = mkkey (), mkkey () in

  (* Leak all long-term keys to the attacker *)
  send ekI; send dkI; send ekR; send dkR;

  (* Generate a pre-shared key *)
  let psk = mkpsk () in

  (* Run all prior case studies in a loop in a separate thread *)
  environment ekI dkI ekR dkR psk;

  (* Run some number of client sessions and return the last session key *)
  let sk = tls_client psk in
  let m = recv () in
  m != sk
```

Fig. 14. Composite case study

complex, most of this complexity is shielded from the user; moreover, thanks to its generic *adequacy theorem*, it is possible to relate Iris proofs to results about the plain operational semantics of the language, as we have seen in Sections 2.4, 4.2 and 4.4. Moreover, Iris comes with an interactive proof mode [37], which greatly simplifies the verification of programs using the logic.

Rather than formalizing the Cryptis programming language from scratch, we implemented it as a library in HeapLang, the default programming language in Iris developments. We developed a small library of HeapLang programs to help manipulating lists and other structured data in protocol code. The resulting language differs in a few respects compared to our paper presentation. First, we formalized cryptographic terms as a separate type from HeapLang values, and rely on an explicit function to encode terms as values. Thanks to this encoding, we can ensure that Diffie-Hellman terms are normalized so that their intended notion of equality coincides with equality in the Coq logic, similar to some encodings of quotient types in type theory [23]. Moreover, this separation simplifies the definition of certain operations on terms that are awkward to express directly for HeapLang programs. For example, HeapLang does not provide any type tests on values, whereas our encoding marks each term constructor with a separate integer that can be inspected by programs, allowing us to perform this kind of test. We implemented nonces as heap locations, which allowed us to reuse much of the location infrastructure, such as the metadata predicates of Figure 4. Naturally, this encoding is well-suited for reasoning about protocols in the symbolic model, but it is not meant to be taken too literally—in particular, real cryptographic protocols need to send messages over the wire as bit strings, and it is not reasonable to expect that attackers that have access to the network at that level comply with the representation constraints that we impose.

On paper, Cryptis proofs are parameterized by a set of axioms mapping tags to invariants. To ensure soundness, we need to ensure that each tag is mapped to exactly *one* invariant. In our implementation, we guarantee this property by expressing this mapping in ghost state. Proofs that use the Iris program logic can simply assume that a certain tag is associated with some invariant as another hypothesis. To use these proofs in a self-contained result, the user needs to declare a ghost location that contains this mapping, and initialize the invariants one by one before invoking the Iris proof. To make this process more modular, we represent tags as Iris *namespaces*: if a protocol uses several tags, we can group them in a single namespace $\mathcal{N}$, so that they can be initialized together and independently of invariants attached to other tags.

Component	Definitions (loc)	Proofs (loc)	Time
Core Cryptis	2587	2106	3m10s
Digital Signatures	218	31	35s
NSL+DH	433	328	1m39s
TLS 1.3	839	1137	2m20s
Composite	120	174	58s

Fig. 15. Code statistics

To give an idea of the effort involved in Cryptis, Figure 15 shows the size of our development, split into lines of code in definitions and proofs. We also include the time spent to compile the code on Coq 8.12 running on an Ubuntu 20.04 laptop with an Intel i7-1185G7 3.00GHz and 15GiB of RAM. These statistics show that the proof effort required to use Cryptis is comparable to the state-of-the-art in semi-automated tools for reasoning about protocols [15].

6 RELATED WORK

There is an extensive literature on automated and semi-automated techniques for verifying cryptographic protocols; see Barbosa et al. [10] for a recent survey. These techniques can be roughly classified along three axes: (1) the model of cryptography (symbolic or computational); (2) the level of automation provided; (3) the verification target (either a model or an efficient, low-level implementation). This design space presents various trade-offs, and each point has certain strengths and weaknesses: the symbolic model is easier to reason about than the computational model, but provides weaker security guarantees; automated tools are easier to use, but can present expressiveness and/or performance issues; and a verified model can be secure against certain cryptographic threats, but still be incorrectly implemented, leading to memory-safety violations or other bugs that can be exploited by attackers. We briefly survey work in the area, focusing on systems that are closer to Cryptis; namely, expressive, semi-automated ones that target the symbolic model.

Protocol Logics. One line of work uses logics to reason about protocols. BAN [21] and related formalisms [28, 48] are early logics that have been applied to various case studies. These are epistemic logics that describe the belief that each agent has about the state of the system; e.g. "principal A believes that it has established a secure channel with B." Despite their intuitive appeal, one issue with BAN and related systems is that their guarantees were not grounded on a clear attacker model. This can be seen, for instance, in Lowe's attack on the original Needham-Schroeder protocol [43, 39], which could be carried out despite the fact that that protocol had been proven correct by Burrows, Abadi, and Needham [21]. Moreover, these works did not attempt to attack the problem of protocol composition.

Some of these issues were partially fixed in Protocol Composition Logic (PCL) [25]. Unlike BAN and related systems, PCL has a clear connection to a program semantics based on execution traces, which include events such as "principal A has decrypted the message M" and "the nonce N is fresh." The logic can reasoning about temporal properties involving these events, allowing one to formulate precise authentication properties that hold for protocols executing in quite arbitrary environments. Moreover, PCL was designed to enable proof composition using the Owicki-Gries method [44]. This works by identifying a set of invariants that each proof relies on, and then showing that these invariants are preserved by the execution of other components. Unfortunately, the method can require an additional number of verification steps that grows quadratically on the size of the components, and makes it impossible to treat components as black boxes (since they need to be reinspected whenever we compose them). Moreover, subsequent work revealed subtle flaws in published PCL proofs [24]. This might be partly a result of the fact that the logic was never implemented, and all these proofs were carried on paper.

Like PCL, Cryptis is grounded on a precise operational semantics, which allows for more robust security guarantees compared to the BAN family. Unlike PCL, however, these guarantees are not connected to an event trace, but simply to the input-output behavior of programs. Moreover, Cryptis is based on a general purpose programming language and logic, making it possible to reason about more complex programs that use cryptographic protocols as a component. Thanks to the treatment of invariants and ghost state in Iris, Cryptis makes it possible to compose protocols proofs with little effort, avoiding the pitfalls of the Owicki-Gries paradigm. Moreover, Cryptis has been implemented, increasing our confidence in its guarantees.

Type Systems. Many semi-automated techniques for the symbolic model are based on type systems. These systems allow reasoning about a variety of protocol properties and features, such as authentication [29], agent compromise [30], authorization [26], secrecy [1] and zero-knowledge proofs [7], among others. Type systems lie on a sweet spot between automation and expressiveness: they allow for better automation than logics like PCL [25] and Cryptis, but require more manual intervention than more automated checkers such as ProVerif [17] and Tamarin [41]. Like logics, type systems have the advantage of a more modular analysis, since type annotations can be used to decompose they verification process into smaller chunks that can be treated independently. Moreover, type checkers can exploit this decomposition for efficiency. By contrast, checkers often perform a whole program analysis and cannot benefit from this level of decomposition, often leading to higher verification times.

A common theme in this line of work is the use of types to distinguish public terms, which can be sent to and received from the network, from other kinds of terms, which might need to be kept hidden from the attacker, or that might convey extra invariants when inspected. This distinction is also found in the pterm and term predicates of Cryptis, although we do not explore the use of a custom type checking algorithm to verify honest agents: Cryptis relies on manual proofs to check the code of honest agents, and uses a type system solely for characterizing the capabilities of the attacker. On the other hand, Cryptis allows us to reason about the interaction of different protocols running in parallel even when they share encryption keys or other secrets, something that is not possible in these systems.

More recent work leverages advances in dependent type systems for verifying rich program properties. The state-of-the-art is DY* [15], a verification library for F*. DY* uses dependent types to model the global execution trace of a protocol. Like in PCL [25], this global trace determines which messages are known to the attacker; however, the framework provides a high-level labeling API to ensure that secret terms are not inadvertently leaked. Cryptis, on the other hand, does not model the event trace: the set of public terms is taken as the more fundamental notion, and we ensure that only messages that can be sent and received are those that have been made public beforehand. Authentication properties can be modularly described in terms of custom ghost state, rather than the event trace. Since these properties are connected to the execution of the underlying language via the adequacy of Iris, Cryptis avoids the pitfalls of other approaches [12], where such properties had to be explicitly stated as axioms that had to be justified separately, relying on the language's metatheory.

Like Cryptis and other type-system approaches, DY* requires honest agents to prove that encrypted and signed messages satisfy certain invariants laid out by the protocol. The specific invariant is determined by the message's *usage*—ghost metadata used to verify the protocol. By contrast, in Cryptis, the invariant is determined by the *tag* of the encrypted message, which can be inspected by protocol participants during execution. Moreover, rather than using an encryption predicate that is fixed for the verified protocol, Cryptis makes it natural to combine predicates associated to different keys used by multiple protocols.

Tagged Protocols. Many works have identified tags as a powerful tool for simplifying protocol composition [22, 6, 5, 40, 19, 20, 3]. In general, it is often possible to guarantee the security of a composite protocol by checking that its components use disjoint messages [3, 32]; tagging is simply a cheap discipline to achieve this goal [3]. These earlier works target protocols written in specialized type systems [40, 20, 20], process calculi [22, 5, 6] or similar formalisms [3], limiting the range of supported properties to help the analysis. Cryptis extends this line of work to a richer programming language, and provides a general purpose logic for writing specifications, while giving up on some automation. The logic is not yet expressive enough to capture some notions of secrecy that appear in the literature on tagging (e.g. [5]), which cover multiple executions, but we believe that this limitation can be addressed in the future.

Other Reasoning Techniques. Sumii and Pierce developed logical relations [46] and bisimulations [45] for reasoning about the secrecy and abstraction properties of *dynamic sealing*, a form of symmetric encryption for the symbolic model. If we ignore the imperative features of Cryptis, which are inherited from Iris and somewhat orthogonal to cryptographic reasoning, the two developments are quite similar. One important difference is that Cryptis imposes invariants on encrypted messages separately for each tag, making it easy to compose protocols even when they share keys. Sumii and Pierce also impose invariants on encrypted messages, but in a less structured way that makes them more difficult to compose: there is no simple way of checking when two arbitrary invariants used for proving different protocols can be combined soundly. Another difference is that Cryptis is currently restricted to reasoning about single executions, whereas their work is relational. On the other hand, the use of Iris allows Cryptis to model temporal properties such as authentication in a compositional way, an aspect that is not covered by those works.

7 CONCLUSION

We presented Cryptis, a mechanized framework that leverages recent advances in program logics to reason about composite systems that use cryptographic protocols. Cryptis uses tag invariants to express the assumptions made by different protocols, guaranteeing that they can be safely composed as long as they use disjoint message formats. Moreover, protocols can be integrated within more complex systems, making it possible to reason about the correctness of these systems by reusing the correctness of the protocols.

Future Work. Like related tooks [15], Cryptis' guarantees are currently limited to single executions. This can be restrictive for security, since many secrecy properties talk about pairs of executions (e.g. indistinguishability). We plan to lift this restriction in the future, drawing inspiration from Sumii and Pierce's work [45, 46], as well as recent work that extends Iris with relational reasoning [27]. Another avenue for strengthening the logic would be to extend it for reasoning about probabilistic properties and the computational model of cryptography. Recent work shows that probabilistic reasoning can benefit from separation logic [11], and we believe that these developments could be naturally incorporated to our setting.

By basing Cryptis on a general-purpose logic, we open the door for investigating the interaction of cryptographic protocols with other programming disciplines. A particularly intriguing possibility would be to combine authentication properties with session types, showing that authentication protocols can be used to safely create sessions in an adversarial setting. Actris [34, 33], a recent logic that adds support for session types in Iris, would be a natural starting point for this extension.

REFERENCES

[1] Martín Abadi. "Secrecy by Typing in Security Protocols". In: *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*. 1997, pp. 611–638. DOI: 10.1007/BFb0014571. URL: https://doi.org/10.1007/BFb0014571.

[2] David Adrian et al. "Imperfect forward secrecy: how Diffie-Hellman fails in practice". In: *Commun. ACM* 62.1 (2019), pp. 106–114. DOI: 10.1145/3292035. URL: https://doi.org/10.1145/3292035.

[3] Suzana Andova et al. "A framework for compositional verification of security protocols". In: *Inf. Comput.* 206.2-4 (2008), pp. 425–459. DOI: 10.1016/j.ic.2007.07.002. URL: https://doi.org/10.1016/j.ic.2007.07.002.

[4] Andrew W. Appel et al. "A very modal model of a modern, major, general type system". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007.* Ed. by Martin Hofmann and Matthias Felleisen. ACM, 2007, pp. 109–122. DOI: 10.1145/1190216.1190235. URL: https://doi.org/10.1145/1190216.1190235.

[5] Myrto Arapinis, Vincent Cheval, and Stéphanie Delaune. "Composing Security Protocols: From Confidentiality to Privacy". In: *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. Ed. by Riccardo Focardi and Andrew C. Myers. Vol. 9036. Lecture Notes in Computer Science. Springer, 2015, pp. 324–343. DOI: 10.1007/978-3-662-46666-7_17. URL: https://doi.org/10.1007/978-3-662-46666-7%5C_17.

[6] Myrto Arapinis, Vincent Cheval, and Stéphanie Delaune. "Verifying Privacy-Type Properties in a Modular Way". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012.* Ed. by Stephen Chong. IEEE Computer Society, 2012, pp. 95–109. DOI: 10.1109/CSF.2012.16. URL: https://doi.org/10.1109/CSF.2012.16.

[7] Michael Backes, Catalin Hritcu, and Matteo Maffei. "Type-checking zero-knowledge". In: *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008.* Ed. by Peng Ning, Paul F. Syverson, and Somesh Jha. ACM, 2008, pp. 357–370. DOI: 10.1145/1455770.1455816. URL: https://doi.org/10.1145/1455770.1455816.

[8] Michael Backes, Catalin Hritcu, and Matteo Maffei. "Union and Intersection Types for Secure Protocol Implementations". In: *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*. Ed. by Sebastian Mödersheim and Catuscia Palamidessi. Vol. 6993. Lecture Notes in Computer Science. Springer, 2011, pp. 1–28. DOI: 10.1007/978-3-642-27375-9_1. URL: https://doi.org/10.1007/978-3-642-27375-9%5C_1.

[9] Michael Backes, Catalin Hritcu, and Matteo Maffei. "Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations". In: *J. Comput. Secur.* 22.2 (2014), pp. 301–353. DOI: 10.3233/JCS-130493. URL: https://doi.org/10.3233/JCS-130493.

[10] Manuel Barbosa et al. "SoK: Computer-Aided Cryptography". In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 1393. URL: https://eprint.iacr.org/2019/1393.

[11] Gilles Barthe, Justin Hsu, and Kevin Liao. "A probabilistic separation logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 55:1–55:30. DOI: 10.1145/3371123. URL: https://doi.org/10.1145/3371123.

[12] Jesper Bengtson et al. “Refinement types for secure implementations”. In: *ACM Trans. Program. Lang. Syst.* 33.2 (2011), 8:1–8:45. DOI: 10.1145/1890028.1890031. URL: https://doi.org/10.1145/1890028.1890031.

[13] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. “Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate”. In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017.* IEEE Computer Society, 2017, pp. 483–502. DOI: 10.1109/SP.2017.26. URL: https://doi.org/10.1109/SP.2017.26.

[14] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. “Modular verification of security protocol code by typing”. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010.* Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 445–456. DOI: 10.1145/1706299.1706350. URL: https://doi.org/10.1145/1706299.1706350.

[15] Karthikeyan Bhargavan et al. “DY* : A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code”. In: *EuroS&P 2021 - 6th IEEE European Symposium on Security and Privacy.* Virtual, Austria, Sept. 2021. URL: https://hal.inria.fr/hal-03178425.

[16] Lars Birkedal et al. “First steps in synthetic guarded domain theory: step-indexing in the topos of trees”. In: *Log. Methods Comput. Sci.* 8.4 (2012). DOI: 10.2168/LMCS-8(4:1)2012. URL: https://doi.org/10.2168/LMCS-8(4:1)2012.

[17] Bruno Blanchet. “An Efficient Cryptographic Protocol Verifier Based on Prolog Rules”. In: *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada.* IEEE Computer Society, 2001, pp. 82–96. DOI: 10.1109/CSFW.2001.930138. URL: https://doi.org/10.1109/CSFW.2001.930138.

[18] Florian Böhl and Dominique Unruh. “Symbolic universal composability”. In: *J. Comput. Secur.* 24.1 (2016), pp. 1–38. DOI: 10.3233/JCS-140523. URL: https://doi.org/10.3233/JCS-140523.

[19] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. “Authenticity by tagging and typing”. In: *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering, FMSE 2004, Washington, DC, USA, October 29, 2004.* Ed. by Vijayalakshmi Atluri et al. ACM, 2004, pp. 1–12. DOI: 10.1145/1029133.1029135. URL: https://doi.org/10.1145/1029133.1029135.

[20] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. “Compositional Analysis of Authentication Protocols”. In: *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings.* Ed. by David A. Schmidt. Vol. 2986. Lecture Notes in Computer Science. Springer, 2004, pp. 140–154. DOI: 10.1007/978-3-540-24725-8_11. URL: https://doi.org/10.1007/978-3-540-24725-8%5C_11.

[21] Michael Burrows, Martín Abadi, and Roger M. Needham. “A Logic of Authentication”. In: *ACM Trans. Comput. Syst.* 8.1 (1990), pp. 18–36. DOI: 10.1145/77648.77649. URL: https://doi.org/10.1145/77648.77649.

[22] Ştefan Ciobâcă and Véronique Cortier. “Protocol Composition for Arbitrary Primitives”. In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010.* IEEE Computer Society, 2010, pp. 322–336. DOI: 10.1109/CSF.2010.29. URL: https://doi.org/10.1109/CSF.2010.29.

[23] Cyril Cohen. “Pragmatic Quotient Types in Coq”. In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings.* Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 213–228. DOI: 10.1007/978-3-642-39634-2_17. URL: https://doi.org/10.1007/978-3-642-39634-2%5C_17.

[24] Cas Cremers. “On the Protocol Composition Logic PCL”. In: *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security.* ASIACCS ’08. Tokyo,

Japan: Association for Computing Machinery, 2008, pp. 66–76. ISBN: 9781595939791. DOI: 10.1145/1368310.1368324. URL: https://doi.org/10.1145/1368310.1368324.

[25] Anupam Datta et al. "Protocol Composition Logic". In: *Formal Models and Techniques for Analyzing Security Protocols*. Ed. by Véronique Cortier and Steve Kremer. Vol. 5. Cryptology and Information Security Series. IOS Press, 2011, pp. 182–221. DOI: 10.3233/978-1-60750-714-7-182. URL: https://doi.org/10.3233/978-1-60750-714-7-182.

[26] Cédric Fournet, Andy Gordon, and Sergio Maffeis. "A Type Discipline for Authorization in Distributed Systems". In: *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*. IEEE Computer Society, 2007, pp. 31–48. DOI: 10.1109/CSF.2007.7. URL: https://doi.org/10.1109/CSF.2007.7.

[27] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 442–451. DOI: 10.1145/3209108.3209174. URL: https://doi.org/10.1145/3209108.3209174.

[28] Li Gong, Roger M. Needham, and Raphael Yahalom. "Reasoning about Belief in Cryptographic Protocols". In: *Proceedings of the 1990 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 7-9, 1990*. IEEE Computer Society, 1990, pp. 234–248. DOI: 10.1109/RISP.1990.63854. URL: https://doi.org/10.1109/RISP.1990.63854.

[29] Andrew D. Gordon and Alan Jeffrey. "Authenticity by Typing for Security Protocols". In: *Journal of Computer Security* 11.4 (2003), pp. 451–520. URL: http://content.iospress.com/articles/journal-of-computer-security/jcs189.

[30] Andrew D. Gordon and Alan Jeffrey. "Secrecy Despite Compromise: Types, Cryptography, and the Pi-Calculus". In: *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*. Ed. by Martín Abadi and Luca de Alfaro. Vol. 3653. Lecture Notes in Computer Science. Springer, 2005, pp. 186–201. DOI: 10.1007/11539452_17. URL: https://doi.org/10.1007/11539452%5C_17.

[31] Thomas Groß and Sebastian Mödersheim. "Vertical Protocol Composition". In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society, 2011, pp. 235–250. DOI: 10.1109/CSF.2011.23. URL: https://doi.org/10.1109/CSF.2011.23.

[32] Joshua D. Guttman and F. Javier Thayer. "Protocol Independence through Disjoint Encryption". In: *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*. IEEE Computer Society, 2000, pp. 24–34. DOI: 10.1109/CSFW.2000.856923. URL: https://doi.org/10.1109/CSFW.2000.856923.

[33] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. "Actris: session-type based reasoning in separation logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 6:1–6:30. DOI: 10.1145/3371074. URL: https://doi.org/10.1145/3371074.

[34] Jonas Kastberg Hinrichsen et al. "Machine-checked semantic session typing". In: *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. Ed. by Catalin Hritcu and Andrei Popescu. ACM, 2021, pp. 178–198. DOI: 10.1145/3437992.3439914. URL: https://doi.org/10.1145/3437992.3439914.

[35] Ralf Jung et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151. URL: https://doi.org/10.1017/S0956796818000151.

[36] Ralf Jung et al. "The future is ours: prophecy variables in separation logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 45:1–45:32. DOI: 10.1145/3371113. URL: https://doi.org/10.1145/3371113.

[37] Robbert Krebbers, Amin Timany, and Lars Birkedal. “Interactive proofs in higher-order concurrent separation logic”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 205–217. DOI: 10.1145/3009837.3009855. URL: https://doi.org/10.1145/3009837.3009855.

[38] Robbert Krebbers et al. “The Essence of Higher-Order Concurrent Separation Logic”. In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Hongseok Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26. URL: https://doi.org/10.1007/978-3-662-54434-1%5C_26.

[39] Gavin Lowe. “Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR”. In: *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS ’96, Passau, Germany, March 27-29, 1996, Proceedings*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 1055. Lecture Notes in Computer Science. Springer, 1996, pp. 147–166. DOI: 10.1007/3-540-61042-1_43. URL: https://doi.org/10.1007/3-540-61042-1%5C_43.

[40] Matteo Maffei. “Tags for Multi-Protocol Authentication”. In: *Electron. Notes Theor. Comput. Sci.* 128.5 (2005), pp. 55–63. DOI: 10.1016/j.entcs.2004.11.042. URL: https://doi.org/10.1016/j.entcs.2004.11.042.

[41] Simon Meier et al. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48. URL: https://doi.org/10.1007/978-3-642-39799-8%5C_48.

[42] Hiroshi Nakano. “A Modality for Recursion”. In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 2000, pp. 255–266. DOI: 10.1109/LICS.2000.855774. URL: https://doi.org/10.1109/LICS.2000.855774.

[43] Roger M. Needham and Michael D. Schroeder. “Using Encryption for Authentication in Large Networks of Computers”. In: *Commun. ACM* 21.12 (1978), pp. 993–999. DOI: 10.1145/359657.359659. URL: https://doi.org/10.1145/359657.359659.

[44] Susan S. Owicki and David Gries. “An Axiomatic Proof Technique for Parallel Programs I”. In: *Acta Informatica* 6 (1976), pp. 319–340. DOI: 10.1007/BF00268134. URL: https://doi.org/10.1007/BF00268134.

[45] Eijiro Sumii and Benjamin C. Pierce. “A bisimulation for dynamic sealing”. In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 169–192. DOI: 10.1016/j.tcs.2006.12.032. URL: https://doi.org/10.1016/j.tcs.2006.12.032.

[46] Eijiro Sumii and Benjamin C. Pierce. “Logical Relations for Encryption”. In: *J. Comput. Secur.* 11.4 (2003), pp. 521–554. URL: http://content.iospress.com/articles/journal-of-computer-security/jcs180.

[47] Nikhil Swamy et al. “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*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 256–270. DOI: 10.1145/2837614.2837655. URL: https://doi.org/10.1145/2837614.2837655.

[48] Paul F. Syverson and Paul C. van Oorschot. “On unifying some cryptographic protocol logics”. In: *1994 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA,*

USA, May 16-18, 1994. IEEE Computer Society, 1994, pp. 14–28. DOI: 10.1109/RISP.1994.296595. URL: https://doi.org/10.1109/RISP.1994.296595.

[49] The Coq Development Team. *The Coq Proof Assistant, version 8.12.0*. Version 8.12.0. July 2020. DOI: 10.5281/zenodo.4021912. URL: https://doi.org/10.5281/zenodo.4021912.