Cryptis: Cryptographic Reasoning in Separation Logic

ARTHUR AZEVEDO DE AMORIM, Rochester Institute of Technology, USA
AMAL AHMED, Northeastern University, USA
MARCO GABOARDI, Boston University, USA

We introduce *Cryptis*, an extension of the Iris separation logic for the symbolic model of cryptography. The combination of separation logic and cryptographic reasoning allows us to prove the correctness of a protocol and later reuse this result to verify larger systems that rely on the protocol. To make this integration possible, we propose novel specifications for authentication protocols that allow agents in a network to agree on the use of system resources. We evaluate our approach by verifying various authentication protocols and a key-value store server that uses these authentication protocols to connect to clients. Our results are formalized in Rocq.

CCS Concepts: • **Theory of computation** → **Separation logic**; • **Security and privacy** → **Logic and verification**.

Additional Key Words and Phrases: Separation Logic, Cryptographic Protocols, Authentication

ACM Reference Format:
Arthur Azevedo de Amorim, Amal Ahmed, and Marco Gaboardi. 2026. Cryptis: Cryptographic Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 10, POPL, Article 88 (January 2026), 31 pages. https://doi.org/10.1145/3776730

1 Introduction

Computer systems must manage various resources to behave correctly, in particular regarding security and privacy. However, doing so is nontrivial, especially when resources are shared by components that might interfere with each other. For example, if a networked system uses a cryptographic protocol and private keys are not shared with care, a security breach might result. A great tool for ruling out such resource conflicts is *separation logic* [Brookes 2007; O'Hearn 2007; Reynolds 2002]. Assertions denote the ownership of resources, and if a program meets a specification, it is guaranteed not to affect any resources disjoint from its pre- or postconditions. What constitutes a resource depends on the application. Originally, resources were data structures in memory, and being disjoint meant avoiding aliasing. This has since been generalized to other types of resources, such as the state of a concurrent protocol [Hinrichsen, Bengtson, et al. 2020] or sources of randomness [Bao et al. 2022; Barthe, Hsu, et al. 2020; Li et al. 2023].

By describing precisely what resources are used, and how they are used, separation logic brought a key advancement to program verification: *compositionality*. We can verify each component in isolation, without knowing exactly what resources are used elsewhere. Later, we can argue that the entire system is correct, provided that the resources used by each component are kept separate. This allows the logic to scale to large systems, including many that were challenging to handle with prior techniques, such as concurrent or distributed ones. And thanks to its rich specification language, the logic can be used to reason about a wide range of components with diverse purposes.

Authors' Contact Information: Arthur Azevedo de Amorim, Rochester Institute of Technology, Rochester, NY, USA, arthur.aa@gmail.com; Amal Ahmed, Northeastern University, Boston, MA, USA, amal@ccs.neu.edu; Marco Gaboardi, Boston University, Boston, MA, USA, gaboardi@bu.edu.

ACM 2475-1421/2026/1-ART88
https://doi.org/10.1145/3776730

Individual proofs of correctness can be composed in a unified formalism, thus ruling out bugs due to possible mismatches between the guarantees of one component and the requirements of another.

Due to the relative novelty of separation logic, however, this power remains underexplored in many domains. Among many examples, we can mention *cryptographic protocols*. To illustrate, suppose we want to verify a distributed application that serves multiple clients. Many frameworks have been introduced for tackling this task under increasingly realistic assumptions [Hinrichsen, Bengtson, et al. 2020; Krogh-Jespersen et al. 2020; Sergey et al. 2018]. For example, Gondelman et al. [2023] showed how to verify a remote procedure call (RPC) library and a key-value store in Aneris [Krogh-Jespersen et al. 2020], which assumes that messages can be dropped or duplicated, but not tampered with. It would be desirable to extend these results to a weaker model, where messages might be forged or tampered with, and where reliable communication must be enforced cryptographically. However, while there are several techniques for verifying protocols [Arquint et al. 2023; Bhargavan, Bichhawat, Do, et al. 2021b; Blanchet 2001; Datta et al. 2011; Meier et al. 2013; Vanspauwen and Jacobs 2015], they have never been applied to reason about application-level guarantees, such as proving that a client only receives correct responses from the server.

The goal of this paper is to connect these two lines of work. We introduce a new separation logic, *Cryptis*, which extends Iris [Jung, Krebbers, Jourdan, et al. 2018] with the Dolev-Yao symbolic model of cryptography [Dolev and Yao 1983]. Cryptis allows us to reuse proofs of protocol correctness to verify application-level specifications, which hold even in the presence of powerful adversaries that control the network. Proof reuse is enabled by several Iris features, including its support for concurrency, higher-order ghost state and invariants. These features are orthogonal to other parts of Cryptis, so it is possible to compose protocols with other concurrent programs and reason about their behavior without compromising soundness.

Core Features. To reason in the Dolev-Yao model, Cryptis follows prior work and uses a special predicate to overapproximate the set of messages known to the attacker [Arquint et al. 2023; Bhargavan, Bichhawat, Do, et al. 2021b; Vanspauwen and Jacobs 2015]. When a message is built using cryptographic primitives, such as encryption or digital signatures, we can define which properties must hold of the contents of the message on a per-protocol basis. Upon receipt, these properties allow us to prove that protocols meet their desired specifications.

To enable proof reuse, Cryptis associates terms such as nonces or cryptographic keys with *tokens*—resources that allow us to bind a term to metadata or other resources. For example, when clients and servers authenticate, they can use metadata associated with a session key to track how many messages have been exchanged through the connection, which allows them to transfer resources via messages using the escrow pattern [Kaiser et al. 2017; Turon et al. 2014], similar to what is done in Aneris [Gondelman et al. 2023; Krogh-Jespersen et al. 2020]. Because the connections are authenticated, a server can assume that the exchanged resources pertain to a specific client, which provides the capability to modify that client's data without interfering with other clients.

Evaluation. We evaluate Cryptis by verifying a key-value store that guarantees that clients always receive the correct response from the server. The store is built on top of several modules: RPC, reliable connections, and authentication, where each component is verified solely based on the specifications of the others. To our knowledge, this is the first correctness proof for such an application running on a Dolev-Yao network. Including protocols in the model of a system also allows us to analyze how its behavior is affected when honest agents can be compromised—a common concern in modern protocols [Cohn-Gordon et al. 2016]. For example, we prove that our store behaves correctly even when the long-term keys are leaked, provided that the client communicates with the server using a session key that was exchanged before the leak.

Game-Based Specifications. It is common to define the security of a cryptographic protocol via a *game*—a piece of code where honest agents aim to achieve some goal, such as exchanging an unguessable session key, even in the presence of an attacker. The protocol is secure, by definition, if the attacker cannot win the game and prevent the agents from achieving their goal. Games are one of the main paradigms of specification in the computational model of cryptography, where messages are bit vectors and adversaries are probabilistic algorithms. They provide an intuitive way to formulate concepts such as "secrecy" that would be otherwise difficult to define. Despite their appeal, reasoning about games in the computational model is notoriously difficult, because it requires intricate probabilistic arguments and often involves reductions ("given an adversary against this protocol P, I can build an adversary against some hard problem P'").

By contrast, Cryptis specifications are easier to prove than their computational analogues, but it might not be clear what protection they provide. To clarify this point, we advocate for a methodology based on *symbolic* security games. Like games for the computational model, symbolic games are simply code where honest agents interact with an attacker. Their proofs of security, however, are simpler than those in the computational model, since they can be carried within Cryptis. The adequacy of the logic allows us to translate such proofs to self-contained trace properties, which can be assessed independently of Cryptis. While symbolic games have appeared in prior works [Böhl and Unruh 2016], ours is the first to show how we can reason about them via a logic.

Trace-Based Specifications. Dolev-Yao tools often define correctness in terms of a trace of events—ghost data that describes the belief or the intent of agents at various points [Arquint et al. 2023; Bhargavan, Bichhawat, Do, et al. 2021b; Blanchet 2002; Lowe 1997; Meier et al. 2013]. For example, when an agent authenticates, they might emit an event to record the exchanged keys and who they believe the other participant is. We can rule out various bugs by forcing such events to match. To prevent a man-in-the-middle attack, we can verify that the event marking the end of a handshake is matched by an earlier event marking that an agent accepted the connection; to prevent replay attacks, we can also require that acceptance occur at most once for a given combination of keys.

Cryptis follows a different approach. Rather than relying on a baked-in trace, users can verify a protocol by plugging in their own ghost state—typically, using term metadata. An authentication protocol, as we will see, is simply a means for the agents to agree on a secret shared key and establish ghost resources to coordinate their actions. We could use ghost state to store an event trace, in which case it would be possible to adapt the classical notions of authentication into Cryptis. Nevertheless, we have not found a reason to do so: our specifications are capable of preventing bugs similar to those based on event traces, without the need for detailed temporal reasoning.

Secrecy as a Resource. Extending separation logic with symbolic cryptography provides novel idioms for reasoning about security, by treating secrecy as a resource secret k. While this resource is available, the term k is guaranteed to be secret, but the resource can be consumed at any point to make k available to the attacker. This enables a new model of dynamic compromise. Systems based on an event trace often feature an attacker API with functions for compromising agents or sessions. Operationally, this has the effect of creating a special event indicating when the compromise occurred and allowing the attacker to access private data. But because a compromise can occur anytime, it is difficult to reason about the behavior of a protocol under a specific compromise scenario (e.g., where a key is compromised only after a certain event takes place). Cryptis, on the other hand, allows us to model compromise in security games, by adding a command to leak a sensitive cryptographic term k in a specific step of the game. If we keep secret t until the compromise, we can argue that any actions that take place earlier are unaffected by it.

Contributions. In sum, our contributions are:

- *Cryptis*, a separation logic for symbolic cryptography with a trace-less semantics (Section 2).
- A new model of key compromise that treats the secrecy of a key as a separation-logic resource.
- Novel authentication specifications for coordinating agents via resources tied to cryptographic material (Sections 4 and 5).
- Connecting Cryptis proofs to security results phrased with *symbolic security games*.
- Case studies showing that protocol specifications can be reused in application-level proofs.
- A formalization of our results in Rocq [Team 2025] using Iris [Jung, Krebbers, Jourdan, et al. 2018] and the Iris proof mode [Krebbers et al. 2017].

Structure of the Paper. Section 2 gives a comprehensive overview of Cryptis. Section 3 presents the specification and the architecture of a modular key-value store cloud application, whose components we describe and verify in the rest of the paper. First, we show how to verify *authentication protocols*, which allow agents to exchange keys for encrypting sessions. We verify the classic Needham-Schroeder-Lowe protocol [Lowe 1996; Needham and Schroeder 1978] (Section 4), which uses asymmetric encryption, and the ISO protocol [Krawczyk 2003] (Section 5), which uses Diffie-Hellman key exchange and digital signatures. For the latter, we prove *forward secrecy*: session keys remain secret even after long-term keys are compromised. These protocols can be reused to verify *authenticated, reliable channels* (Section 6) which, in turn, can be used to implement an RPC mechanism (Section 7). This mechanism guarantees the security of the key-value store and allows us to prove its correctness (Section 8). In Section 9, we discuss details of our formalization and of the model of Cryptis. Section 10 discusses related work and Section 11 concludes.

Data-Availability Statement. The implementation and the case studies are included in the accompanying artifact [Azevedo de Amorim et al. 2025].

2 Core Cryptis

Cryptis is a logic for reasoning about networked programs in a typical functional imperative language. The logic and the language are summarized in Figure 1. Most features are inherited from Iris, so we focus on our extensions and refer readers to Jung, Krebbers, Jourdan, et al. [2018] for more background on other features. The $\triangleright$ symbol refers to the later modality, which enables recursive definitions while avoiding paradoxes. The assertion $\square\, P$ means that P holds persistently, without holding ephemeral resources. The assertion $P \Rrightarrow_{\mathcal{E}} Q$ means that we can make Q hold by consuming P, modifying ghost state and accessing invariants under any namespace $\mathcal{N} \in \mathcal{E}$.

Semantics and Networking. The operational semantics follows other Iris developments [Jung, Krebbers, Jourdan, et al. 2018]. A program configuration comprises a *heap*, a *thread pool*, the *network state* and a set of *generated nonces*. A *per-thread* reduction relation specifies how each thread can step and interact with its environment, possibly modifying objects on the heap, sending and receiving messages, and forking off new threads. There are two primitive networking functions, send and recv. These functions are restricted to *terms t*, a subset of values that excludes anything that cannot be meaningfully sent over the network, such as pointers or closures. The network state in Cryptis is modeled as a set of messages that can be observed by the attacker. In the protocol analysis literature, this set is sometimes known as the *attacker knowledge*. The attacker knowledge grows whenever an agent sends a message. To receive a message, the semantics chooses a message from the attacker knowledge nondeterministically and returns it to the calling agent. The message is not removed from the knowledge, so it could be received multiple times, even by different agents. As usual in the symbolic model, messages do not include their sender or recipient, since this information is unreliable. The per-thread reduction relation is lifted to whole configurations by choosing an

Key types $u := \mathsf{aenc} \mid \mathsf{adec} \mid \mathsf{sign} \mid \mathsf{verify} \mid \mathsf{senc}$

Functionalities $F := \mathsf{aenc} \mid \mathsf{sign} \mid \mathsf{senc}$

Terms $t, sk, pk, k := n \mid \mathcal{N} \mid (t_1, t_2) \mid \{t\}@k \mid \mathsf{key}_u\ t \mid t\ \hat{}\ (t_1 \cdots t_n) \mid \cdots$

Expressions $e := \mathsf{send}\ e \mid \mathsf{recv} \mid \{e\}@e \mid \mathsf{key}_u\ e \mid \mathsf{open}\ e_1\ e_2 \mid \mathsf{pkey}\ e \mid e_1\ \hat{}\ e_2 \mid \mathsf{mk\_nonce} \mid \cdots$

Assertions $P, Q := \boxed{F \mapsto_{\mathcal{N}} \varphi \mid t \mapsto_{\mathcal{N}} x \mid \mathsf{public}\ t} \mid \mathsf{token}\ F\ \mathcal{E} \mid \mathsf{token}\ t\ \mathcal{E} \mid \cdots$

Term equations	Private keys	Opening keys
$t\ \hat{}\ () = t$	$(\mathsf{key}_{\mathsf{aenc}}\ t)_{\mathsf{sec}} \triangleq \mathsf{key}_{\mathsf{adec}}\ t$	$(\mathsf{key}_{\mathsf{aenc}}\ t)_{\mathsf{open}} \triangleq \mathsf{key}_{\mathsf{adec}}\ t$
$t\ \hat{}\ (t_1 \cdots t_n)\ \hat{}\ (t_{n+1} \cdots t_m) = t\ \hat{}\ (t_1 \cdots t_m)$	$(\mathsf{key}_{\mathsf{sign}}\ t)_{\mathsf{sec}} \triangleq \mathsf{key}_{\mathsf{sign}}\ t$	$(\mathsf{key}_{\mathsf{sign}}\ t)_{\mathsf{open}} \triangleq \mathsf{key}_{\mathsf{verify}}\ t$
$t\ \hat{}\ (t_1 t_2) = t\ \hat{}\ (t_2 t_1)$	$(\mathsf{key}_{\mathsf{senc}}\ t)_{\mathsf{sec}} \triangleq \mathsf{key}_{\mathsf{senc}}\ t$	$(\mathsf{key}_{\mathsf{senc}}\ t)_{\mathsf{open}} \triangleq \mathsf{key}_{\mathsf{senc}}\ t$

Public terms

$$\begin{aligned}
\mathsf{public}\ n, \mathsf{public}\ \mathcal{N} &\iff \mathsf{True} \\
\mathsf{public}\ (t_1, t_2) &\iff \mathsf{public}\ t_1 \land \mathsf{public}\ t_2 \\
\mathsf{public}\ \{(\mathcal{N}, t)\}@k &\iff \mathsf{public}\ t \land \mathsf{public}\ k \lor \exists \varphi, F_u \mapsto_{\mathcal{N}} \varphi \land \square\, \varphi\ k_{\mathsf{sec}}\ t * \square(\mathsf{public}\ k_{\mathsf{open}} \mathrel{-\!\!*} \mathsf{public}\ t_1) \\
\mathsf{public}\ (\mathsf{key}_u\ t) &\iff \mathsf{public}\ t \lor \mathsf{public_key}\ u \\
\mathsf{public}\ (t\ \hat{}\ t') &\iff \mathsf{True} \qquad (\text{when } t \text{ does not begin with } \hat{}) \\
\mathsf{public}\ (t\ \hat{}\ (t_1 \cdots t_n)) &\iff \exists i, \mathsf{public}\ (t\ \hat{}\ (t_1 \cdots t_{i-1} t_{i+1} \cdots t_n)) \land \mathsf{public}\ t_i
\end{aligned}$$

Operational semantics

$$\mathsf{open}\ (\{t_1\}@k)\ k_{\mathsf{open}} \to \mathsf{Some}\ t_1$$
$$\mathsf{open}\ t\ k \to \mathsf{None} \quad (\text{in all other cases})$$

Public keys

$$\frac{u \in \{\mathsf{aenc}, \mathsf{verify}\}}{\mathsf{public\_key}\ u}$$

Program logic

$$\{\triangleright\, \mathsf{public}\ t\}\ \mathsf{send}\ t\ \{\mathsf{True}\} \qquad \{\mathsf{True}\}\ \mathsf{recv}\ ()\ \{t, \mathsf{public}\ t\}$$

$$\left\{\square \forall t\, t', t' \in T(t) \mathrel{-\!\!*} t \preceq t'\right\}\ \mathsf{mk\_nonce}\ ()\ \left\{t, \square(\mathsf{public}\ t \iff \triangleright \square\, \varphi\ t) * \mathop{\Large *}_{t' \in T(t)} \mathsf{token}\ t'\ \top\right\}$$

Metadata rules (for both terms and message predicates)

$$\alpha \mapsto_{\mathcal{N}} \beta_1 * \alpha \mapsto_{\mathcal{N}} \beta_2 \vdash \triangleright(\beta_1 = \beta_2) \qquad \alpha \mapsto_{\mathcal{N}} \beta * \mathsf{token}\ \alpha\ \mathcal{E} \vdash \uparrow\mathcal{N} \not\subseteq \mathcal{E}$$
$$\uparrow\mathcal{N} \subseteq \mathcal{E} * \mathsf{token}\ \alpha\ \mathcal{E} \Rrightarrow \alpha \mapsto_{\mathcal{N}} \beta \qquad \mathsf{token}\ \alpha\ (\mathcal{E}_1 \uplus \mathcal{E}_2) \vdash \mathsf{token}\ \alpha\ \mathcal{E}_1 * \mathsf{token}\ \alpha\ \mathcal{E}_2$$

Fig. 1. The Cryptis logic and programming language. The highlighted assertions are persistent.

active thread nondeterministically and allowing it to take a step. These steps are interleaved with *attacker actions*, which read some number of channel messages, combine these messages using a cryptographic operation (see "Attacker model" below), and add the result to the knowledge.

Cryptographic Operations. Terms can be manipulated with several cryptographic primitives: *sealing* ($\{t\}@k$), *Diffie-Hellman exponentiation* ($\hat{}$) and *nonce generation* (mk_nonce). Sealing is an umbrella primitive that encodes various encryption-like functionalities. In $\{t\}@k$, the term t is the sealed message, and k is the sealing key. We use k to range over key terms, and reserve *sk* for private keys and *pk* for public keys. Keys are terms of the form $\mathsf{key}_u\ t$, where t is the seeding material

used to generate it and u is its type. We distinguish between keys for asymmetric encryption ($u \in \{\mathsf{aenc}, \mathsf{adec}\}$), digital signatures ($u \in \{\mathsf{sign}, \mathsf{verify}\}$) and symmetric encryption ($u = \mathsf{senc}$). We can unseal a term by calling open, which succeeds only if the key used for sealing matches the one used for unsealing. The expression pkey sk computes the public key corresponding to some secret key sk. The (partial) operations k_{sec} and k_{open} map a key k to its corresponding private key and opening key. In a Diffie-Hellman term $t \mathbin{\hat{}} (t_1 \cdots t_n)$, the terms $t_1, \ldots, t_n$ represent the exponents. We quotient terms to validate useful properties of exponentiation; in particular, exponents can be freely permuted, and we have the familiar identity $t \mathbin{\hat{}} t_1 \mathbin{\hat{}} t_2 = t \mathbin{\hat{}} t_2 \mathbin{\hat{}} t_1$, which allows agents to compute a shared Diffie-Hellman secret based on their key shares $t \mathbin{\hat{}} t_1$ and $t \mathbin{\hat{}} t_2$. A call to mk_nonce nondeterministically chooses a nonce that does not occur in the set of generated nonces stored in the program configuration. It returns that value and adds it to the set of generated nonces.

Attacker Model. In Cryptis' Dolev-Yao model, cryptographic operations behave as black boxes. It is impossible to manipulate messages as bit strings, to guess nonces or keys out of thin air, or to extract the contents of an encrypted message without its key. On the other hand, we assume that the attacker can invoke *any* cryptographic operation on terms they know—encrypting or decrypting terms using known keys, generating nonces, extracting values from a tuple, etc. By running attacker actions nondeterministically, the semantics overapproximates any sequence of interactions with a Dolev-Yao attacker. Of course, real-life attackers might not abide by the Dolev-Yao restrictions, so Cryptis might miss some attacks. Nevertheless, the model is rich enough to rule out several critical, real attacks, such as Lowe's attack on the NS protocol (Section 4.3). (In our implementation, attacker actions are separate threads that must be explicitly initialized; cf. Section 9.)

Public Terms. To allow agents to communicate securely in the presence of such a powerful attacker, Cryptis forces every message traveling through the network to satisfy a special public predicate. Accordingly, the specification of the networking functions says that send takes in a public term, whereas recv is guaranteed to return a public term. The definition of public balances between two needs: capturing the capabilities of the attacker, on the one hand, and allowing honest agents to reason about their communication, on the other. To model the attacker, the definition ensures that public terms are preserved by all term operations (pairing two terms, sealing a term with a key, etc.), hence by all attacker actions. To make it possible to reason about communication, the public predicate allows us to impose predicates on sealed messages, as it is done in similar tools [Backes et al. 2011, 2014; Bhargavan, Bichhawat, Do, et al. 2021b]. Suppose we want to send a message of the form $m = \{(\mathcal{N}, t)\}@pk$, where $pk = \mathsf{key}_{\mathsf{aenc}}\ t'$ is an encryption key. In typical uses of encryption, the contents of the message, t, are not public. Nevertheless, the definition says that we can prove that m is public provided that t satisfies a certain predicate φ attached to the tag $\mathcal{N}$. The assertion $F \mapsto_{\mathcal{N}} \varphi$ means that the tag $\mathcal{N}$ is associated with the predicate φ for sealed messages under the functionality F. Each tag can be associated with at most one φ under a given F. The protocol verifier chooses which predicates to use by consuming a token $F\,\mathcal{E}$ resource, which states that no tags in $\mathcal{E}$ have had any predicates assigned to them. (Initially, $\mathcal{E}$ is set to $\top$, which contains every tag.) Cryptis focuses on tagged messages to make reasoning modular: if two protocols use different tags, their proofs can be automatically composed. We represent tags with Iris namespaces, which are similar to strings, but can be organized hierarchically. For example, if a protocol P has three messages, we might tag them using the namespaces $P.m1, $P.m2 and $P.m3 (we will use $ to distinguish namespaces from other identifiers). To set up their predicates, we can expose a lemma that consumes a token of the form token F (↑$P), where the set ↑$P contains any namespace that has $P as a prefix. Thus, a user of a protocol does not need to know exactly which tags it uses.

The clause for encrypted terms also includes $\square(\mathsf{public}\ k_{\mathsf{open}} \Rightarrow \mathsf{public}\ t)$, which allows the attacker to conclude that the contents of the message are public if they are ever able to decrypt it

with a public (compromised) decryption key. If $u = \mathsf{sign}$, then the antecedent of this implication is trivial, which means that we can only sign messages when the contents are public. Notice that both this implication and the message predicate φ are guarded by the persistence modality $\Box$: since a Dolev-Yao attacker can duplicate messages arbitrarily, public must be persistent.

Dually, when another agent receives a sealed message, the definition says that we must consider two cases: either the sealing key and the contents of the message are public, or the corresponding sealing predicate holds. The first case typically occurs when reasoning about communication with an attacker or compromised agent, which cannot be expected to enforce non-trivial properties, whereas the second one arises when communicating with another honest agent. This type of case analysis is reminiscent of the use of union types in protocol analysis [Backes et al. 2011, 2014].

Keys can be shared according to common usage patterns. For asymmetric encryption, the sealing key ($u = \mathsf{aenc}$) is always considered public, whereas the unsealing key ($u = \mathsf{adec}$) is public if and only the seed is. For signatures, it's the opposite. A symmetric key is public if and only if its seed is.

The definition of public for exponentials allows agents to freely exchange key shares $t \mathbin{\hat{}} t'$. When there is more than one exponent, the term is public if and only if it can be obtained by combining two smaller public terms via exponentiation. If t is not a DH term, then $\mathsf{public}\ (t \mathbin{\hat{}} t_1 t_2) \iff \mathsf{public}\ t_1 \vee \mathsf{public}\ t_2$. The specification for mk_nonce says, among other things, that the result t is such that $\mathsf{public}\ t$ can be any predicate $\varphi\ t$ chosen. The $\triangleright$ modality is required for soundness: if we had $\mathsf{public}\ t \iff \Box\, \varphi\ t$, we could get a contradiction by choosing $\varphi\ t \triangleq \neg\mathsf{public}\ t$. We can choose $\varphi\ t = \mathsf{True}$ to generate a public nonce, or $\varphi\ t = \mathsf{False}$ to generate a secret.

Term Metadata. The last feature that we need to cover is *term metadata*. The assertion $t \mapsto_{\mathcal{N}} x$ says that the term t has been permanently associated with the metadata item x under the namespace $\mathcal{N}$, where x ranges over elements from arbitrary countable types. Like sealing predicates, we can create metadata by consuming a resource token $t\ \mathcal{E}$. Each term is associated to at most one metadata item under a given $\mathcal{N}$. Tokens are created during nonce generation. As shown in the specification for mk_nonce, the post-condition contains tokens for any term $t' \in T(t)$, where $T(t)$ is a finite set such that t is a subterm of any $t' \in T(t)$ (written $t \preceq t'$). Intuitively, if $t \preceq t'$ and t is fresh, then t' cannot have been used by the program, which means that we are allowed to obtain a token for it.

Metadata in Cryptis serves multiple purposes. One use is to reason about term freshness: if a set of terms T is such that every term has metadata under some namespace $\mathcal{N}$, any term t that still has a token containing $\mathcal{N}$ must not belong to T. We can also use metadata to attach ghost state to a term. The assertion $\boxed{a}^{t}_{\mathcal{N}} \triangleq \exists \gamma, t \mapsto_{\mathcal{N}} \gamma * \boxed{a}^{\gamma}$ says that t is associated with an element a drawn from some resource algebra. This idiom is useful to track ghost state that is associated with an agent, when t is their public key, or with a session, when t is the corresponding session key. Most rules that apply to the Iris ghost ownership assertion carry over to this connective.

Adequacy. Cryptis satisfies an adequacy result that relates specifications to more elementary properties stated in terms of the operational semantics of its language. The formulation of adequacy is similar to the one of Iris, but it also provides tokens that are needed to carry out the proofs. To invoke the adequacy theorem, we must decide which predicates will be associated with each cryptographic functionality and each tag, and consume the tokens to set up these predicates, as shown in Figure 1. Any combination of predicates can be used, provided that they are associated with separate tags and functionalities.

Theorem 2.1. *Suppose that $P\ v$ is a meta-level proposition such that we have a Cryptis proof of $\{\mathsf{token}\ \mathsf{aenc}\ \top * \mathsf{token}\ \mathsf{senc}\ \top * \mathsf{token}\ \mathsf{sign}\ \top\}\ e\ \{v.P\ v\}$. If e terminates in a value v when running in the initial configuration, then $P\ v$ holds. Moreover, the initial configuration cannot reach a stuck state.*

$$\text{pending}\ \gamma * \square(\text{public}\ t \iff \triangleright\, \text{shot}\ \gamma\ 1) \twoheadrightarrow \text{secret}\ t \qquad \text{secret}\ t \Rrightarrow \text{public}\ t$$

$$\text{secret}\ t \Rrightarrow \square(\text{public}\ t \twoheadrightarrow \triangleright\, \text{False}) \qquad \text{secret}\ t * \text{public}\ t \twoheadrightarrow \triangleright\, \text{False}$$

$$\text{secret}\ t \triangleq (\text{public}\ t \twoheadrightarrow \triangleright\, \text{False}) \wedge (\text{True} \Rrightarrow \text{public}\ t) \wedge (\text{True} \Rrightarrow (\text{public}\ t \twoheadrightarrow \triangleright\, \text{False}))$$

Fig. 2. Secrecy resource. The shot $\gamma\ n$ proposition is persistent.

$$[\varphi]_{\text{aenc}}\ sk\ t \triangleq \text{public}\ t \vee \square\, \varphi\ sk\ t \wedge \square(\text{public}\ sk \Rightarrow \text{public}\ t) \qquad \text{aenc}\ pk\ \mathcal{N}\ t \triangleq \{(\mathcal{N}, t)\}@pk$$

$$\text{adec}\ sk\ \mathcal{N}\ t \triangleq \begin{cases} \text{Some}\ t' & \text{if open}\ sk\ t = \text{Some}\ (\mathcal{N}, t') \\ \text{None} & \text{otherwise} \end{cases} \qquad \text{mk\_aenc\_key}\ () \triangleq \text{key}_{\text{adec}}\ (\text{mk\_nonce}\ ())$$

$$\{\text{True}\}\ \text{mk\_aenc\_key}\ ()\ \{sk, \exists t, sk = \text{key}_{\text{adec}}\ t * \text{token}\ sk\ \top * \text{secret}\ sk\}$$

$$\{\text{aenc} \mapsto_{\mathcal{N}} \varphi * [\varphi]_{\text{aenc}}\ sk\ t\}\ \text{aenc}\ (\text{pkey}\ sk)\ \mathcal{N}\ t\ \{t', \text{public}\ t'\}$$

$$\{\text{aenc} \mapsto_{\mathcal{N}} \varphi * \text{public}\ t\}\ \text{adec}\ sk\ \mathcal{N}\ t\ \{v, v = \text{None} \vee \exists t', v = \text{Some}\ t' * [\varphi]_{\text{aenc}}\ sk\ t'\}$$

Fig. 3. Derived constructions for asymmetric encryption. We assume that *sk* ranges over decryption keys.

2.1 Derived Constructions

Cryptis includes several convenience features derived from the core elements presented above.

Secrecy Resources. Besides being public or private, the secrecy of nonce can behave as a resource. This pattern is useful to model dynamic compromise, when an attacker does not have access to some key at first, but eventually compromises it. Consider an ephemeral resource pending γ stating that γ has not been tied to any value yet, whereas the persistent resource shot $\gamma\ n$ means that γ is tied to the integer n and no other value. Such resources are commonplace and can be defined using various ghost state constructions [Timany et al. 2024]. If public t is equivalent to $\triangleright$ shot $\gamma\ n$, we can use pending γ to create a resource secret t (Figure 2), which means that t can become public or private at any point, by exchanging pending γ for shot $\gamma\ 1$ or shot $\gamma\ 0$. Moreover, because pending γ and shot $\gamma\ 1$ contradict each other, we can guarantee that t is secret as long as secret t is available.

Specialized Cryptographic Primitives. Though sealing comprises several functionalities, in practice, it is useful to expose separate functions for each one of them. Figure 3 shows the interface for programming with asymmetric encryption. The function mk_aenc_key is a wrapper around mk_nonce that uses the nonce to generate a decryption key. The functions aenc and adec are wrappers around sealing and opening. Because Cryptis works with tagged messages, it is more convenient for these functions to take the tag as a separate argument. We implement digital signatures and symmetric encryption in a similar way. Note that verify, the function that verifies a signed message, outputs the contents of the signed message, instead of a success bit. To prove the specification of mk_aenc_key, we allocate a resource pending γ and use the specification of mk_nonce to generate a nonce t so that public $t \iff \triangleright$ shot $\gamma\ 1$. Since public $(\text{key}_{\text{adec}}\ t) \iff$ public t, we can create a resource secret $(\text{key}_{\text{adec}}\ t)$, as shown in Figure 2. Moreover, since $t \preceq \text{key}_{\text{adec}}\ t$, we can use the mk_nonce rule to create a token for the key. The specifications for encryption and decryption mention the predicate $[\varphi]_{\text{aenc}}\ sk\ t$, which describes what holds of the contents of the message t if we know that aenc $\mapsto_{\mathcal{N}} \varphi$ holds.

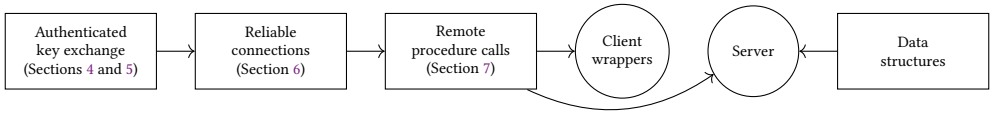

Fig. 4. Structure of key-value store. Arrows denote dependencies, circles denote internal components, and squares denote the key-value store itself.

3 Motivating Application: A Key-Value Store

In the rest of the paper, we will illustrate the expressiveness of Cryptis by verifying the correctness of a simple key-value store. This case study demonstrates how we can reason modularly about a high-level application that provides non-trivial integrity guarantees even in the presence of arbitrary Dolev-Yao attackers, and how these guarantees are affected when long-term keys are compromised. In this section, we content ourselves with an overview of the architecture of the key-value store and its specification. Later, we will dive into its individual components.

Figure 4 summarizes the structure of the application. A server stores client data in internal data structures, and clients performs API calls to retrieve and manipulate their data. The communication between clients and the server is implemented by a remote procedure call (RPC) component, which sits on top of a connection abstraction that preserves the ordering and contents of messages. To create a connection, a client must initiate an authentication handshake with the server, which allows the two parties to exchange a session key and confirm each other's identities. Since our focus is on how such distinct components can be developed and verified modularly, the functionality of each verified component will be rather minimal. For example, the store server provides sequential consistency, runs on a single machine, and stores the client data using an association list. Because of the modular design and the expressiveness of separation logic, it should be possible to make each component more realistic without changing fundamentally how they are connected.

Figure 5 shows the specification of the client API. To interact with the server, the client must first call the connect function. This function returns a connection object c, together with a resource DB.connected sk_C sk_S c that indicates that the client is connected. While the client is connected, it can perform database operations: load a value stored under a key (load), create a new key in the database (create), or store a new value under an existing key (store). The specifications are modeled after the specifications for memory operations in separation logic, with one minor twist: a load can return an incorrect value if the connection is compromised. A connection is compromised if either the server or the client was compromised when the connection was established (that is, if their private keys were known to the attacker).

We can rule out the possibility of a compromise by proving that the agents' private keys were still secret at any point after the connection c was established. Note that it is still possible that these private keys end up leaking at a later point without affecting the integrity of c. As we will see, this guarantee is a product of the post-compromise properties of the underlying key-exchange protocol. Logically, this is a consequence of the way persistent assertions work in Iris. The assertion $\triangleright \Box \neg$compromised c is persistent because it is guarded by the persistence modality $\Box$. This means that, if we are in a proof context where secret sk_C, secret sk_S and DB.connected sk_C sk_S c all hold, we can prove $\triangleright \Box \neg$compromised c without consuming these premises. Later, we can consume secret sk_C or secret sk_S to leak one of the private keys. (Note that our server only allows clients to have one active connection at a time. If multiple active connections were possible, ruling out a compromise would be more difficult, because the attacker would be able to initiate a new session using a compromised long term key and then corrupt client data.)

Notation	Abbreviation	Meaning
sk_C, sk_S, pk_S	—	Long term keys of client and server
k_c	—	Session key used in connection c
DB.disconnected sk_C sk_S	Δ	The client is disconnected
DB.connected sk_C sk_S c	Γ	The client is connected via the connection c
compromised c	—	The connection c is compromised
$T \mapsto_{\text{db}}^{sk_C, sk_S} \bot$	$T \mapsto_{\text{db}} \bot$	No term $t \in T$ is stored in the server
$t_1 \mapsto_{\text{db}}^{sk_C, sk_S} t_2$	$t_1 \mapsto_{\text{db}} t_2$	The value t_2 is stored under the key t_1

$$\{\Delta\}\ \text{DB.connect}\ sk_C\ pk_S\ \{c, \Gamma\} \qquad \{\Gamma\}\ \text{DB.close}\ c\ \{\Delta * \text{public}\ k_c\}$$

$$\{\Gamma * t_1 \mapsto_{\text{db}} t_2\}\ \text{DB.load}\ c\ t_1\ \left\{t_2', (\text{compromised}\ c \vee t_2' = t_2) * \Gamma * t_1 \mapsto_{\text{db}} t_2\right\}$$

$$\{\Gamma * t_1 \mapsto_{\text{db}} \bot\}\ \text{DB.create}\ c\ t_1\ t_2\ \{\Gamma * t_1 \mapsto_{\text{db}} t_2\} \qquad \left\{\Gamma * t_1 \mapsto_{\text{db}} t_2'\right\}\ \text{DB.store}\ c\ t_1\ t_2\ \{\Gamma * t_1 \mapsto_{\text{db}} t_2\}$$

$$t_1 \mapsto_{\text{db}} t_2 * t_1 \mapsto_{\text{db}} t_2' \vdash \text{False} \qquad \text{secret}\ sk_C * \text{secret}\ sk_S * \Gamma \vdash \triangleright \Box \neg\text{compromised}\ c$$

$$\text{token}\ sk_C\ (\uparrow \$\text{db.client}.sk_S) \Rrightarrow_{\top} \Delta * \top \mapsto_{\text{db}} \bot \qquad T_1 \uplus T_2 \mapsto_{\text{db}} \bot \vdash T_1 \mapsto_{\text{db}} \bot * T_2 \mapsto_{\text{db}} \bot$$

Fig. 5. Key-value store assertions and specifications for the client API. For readability, we abbreviate some of the resources and tacitly assume that the terms t_1, t_2 and t_2' are public.

At any moment, the client can choose to disconnect from the server by calling the close function. After disconnecting, k_c, the session key used to encrypt the connection, is no longer needed, so it can be made public and leaked to the attacker. Of course, if the key were leaked, the attacker would be able to read any messages that were encrypted with it, ruining any confidentiality guarantees. We allow the session key to be leaked after disconnection to highlight that this would not affect the *integrity* of the client's data: the attacker could try to send requests to the server using the compromised key, but those requests would be ignored.

To illustrate how these specifications can be used, let us assess the integrity of the store with a security game (Figure 6). The game sets up signature keys for the client and the server, sends the public keys to the attacker, and then runs the client and the server in parallel. The client uses the server to store a value chosen by the attacker and then tries to retrieve that value from the server. Our goal is to prove that the client's assertion succeeds; that is, the client reads back the same value that it stored originally. Moreover, this assertion succeeds even though various keys are leaked during the game. Though the game code is simple, its operational semantics is complex, because the agents run concurrently and their interaction is mediated by a Dolev-Yao attacker. Thus, checking the security of the game forces us to reason about concurrency, making it challenging to provide similar formulations in sequential systems, such as DY* [Bhargavan, Bichhawat, Do, et al. 2021b].

To prove that the game is secure, we use the specification for mk_sign_key, analogous to the one of mk_aenc_key (Figure 3), to obtain secrecy resources secret sk_C and secret sk_S. We also obtain metadata tokens for these keys, which we can use to initialize the ghost state required to run the server and the client. For the client, this means obtaining the assertions DB.disconnected sk_C sk_S and $\top \mapsto_{\text{db}}^{sk_C, sk_S} \bot$, which guarantees that the database is currently uninitialized. Note that the token needed for the initialization lemma mentions the namespace $\$\text{db.client}.sk_S$, whose last component is a secret key rather than a plain identifier. This allows the client's token to be used for setting up databases with multiple servers. (The lemmas used to initialize the server's ghost

```
let run_client leak_keys skC pkS =
  let c1 = DB.connect skC pkS in
  let key = recv () in
  let val = recv () in
  DB.create c1 key val;
  DB.close c1;
  send (session_key c1);
  let c2 = DB.connect skC pkS in
  leak_keys ();
  let val' = DB.load c2 key in
  assert (val = val')
```

```
let game () =
  let skC, skS =
    mk_sign_key (), mk_sign_key () in
  let pkC, pkS = pkey skC, pkey skS in
  send pkC; send pkS;

  let leak_keys () = send skC; send skS in

  fork (fun () -> DB.start_server skS);
  fork (fun () -> run_client leak_keys skC pkS)
```

Fig. 6. Security game for the key-value storage service. The client reads back the value they stored even if long-term keys are leaked after the connection.

state are omitted for brevity.) We pass all these resources to the `run_client` function. When we close the first connection c_1, we are allowed to leak its session key thanks to the specification of DB.close. When we establish the second connection c_2, we use secret sk_C and secret sk_S to prove that c_2 is not compromised, which allows us to prove that DB.load returns the expected value.

4 Authentication: The NSL Protocol

The first component of the key-value store we will analyze is its authentication protocol. For post-compromise security, the implementation uses a protocol based on Diffie-Hellman key exchange, which we will cover in Section 5. Before we do so, however, we will consider the Needham-Schroeder-Lowe protocol [Lowe 1996; Needham and Schroeder 1978] (*NSL*), a classic protocol based on public-key encryption. Though the protocol provides weaker security guarantees, it is often used as an introductory example in protocol-verification tools and, thus, serves as a good point of comparison for Cryptis.

There are two versions of the protocol: one that relies on a trusted server to distribute public keys, and one where the participants know each other's public keys from the start. For simplicity, we model the second one. A typical run can be summarized as follows:

$$I \to R : \mathsf{aenc}\ pk_R\ \$\mathtt{m1}\ [a; pk_I] \qquad R \to I : \mathsf{aenc}\ pk_I\ \$\mathtt{m2}\ [a; b; pk_R] \qquad I \to R : \mathsf{aenc}\ pk_R\ \$\mathtt{m3}\ b.$$

First, the initiator I generates a fresh nonce a and sends it to the responder R, encrypted under their public key pk_R. The responder replies with a together with a fresh nonce b. The initiator confirms the end of the handshake by returning b. If the protocol terminates successfully, and both agents are honest, they can conclude that their identities are correct—that is, they match the public keys sent in the messages—and that the nonces a and b are secret. In particular, they can use a and b to derive a secret session key to encrypt further communication. Figure 7 shows an implementation of the protocol in the Cryptis programming language. To keep examples short, we'll use a syntax inspired by the ProVerif protocol analyzer [Blanchet 2001]: `let` declarations can mention patterns of the form `=p`, which are only matched by `p` itself. Any errors that arise during execution, such as failed pattern matching, cause the code to safely return `None`. (Formally, these errors are managed using the option monad, and `let` in our code snippets should be read as monadic bind.)

4.1 Proving Security

A Cryptis proof is typically structured as follows. First, we formulate the expected security or correctness guarantees of a component using a series of specifications in the Cryptis logic or security games. Then, we determine which predicates we must associate with the encrypted or

```
let initiator skI pkR =
  let pkI = pkey skI in
  let a = mk_nonce () in
  let m1 = aenc pkR $m1 [a; pkI] in
  send m1;
  let m2 = recv () in
  let [=a; b; =pkR] = adec skI $m2 m2 in
  let m3 = aenc pkR $m3 b in
  send m3;
  let k = key senc [pkI; pkR; a; b] in
  k
```

```
let responder skR =
  let pkR = pkey skR in
  let m1 = recv () in
  let [a; pkI] = adec skR $m1 m1 in
  if not (is_aenc_key pkI) then fail ();
  let b = mk_nonce () in
  let m2 = aenc pkR $m2 [a; b; pkR] in
  send m2;
  let m3 = adec skR $m3 (recv ()) in
  if m3 != b then fail ();
  let k = key senc [pkI; pkR; a; b] in
  (pkI, k)
```

Fig. 7. Implementation of NSL. Variables beginning with sk and pk refer to secret and public keys.

signed messages exchanged in the protocol. We prove lemmas that allow us to initialize these message predicates by consuming corresponding resource tokens, using the rules of Figure 1. We proceed to prove that the desired specifications hold assuming that the required message predicates are available. Finally, to prove a closed result that is independent of the Cryptis logic (e.g., that an assertion in a game does not fail), we invoke the adequacy theorem (Theorem 2.1), using the tokens that it generates to initialize the message predicates required by each component of the program.

Let us see how this applies in the context of NSL. The handshake produces a session key k that is guaranteed to be secret, as long as both participants are honest. We formalize this claim with the following theorem, which, moreover, produces metadata tokens for the agents to coordinate their actions. We use the metavariable σ to range over *sessions*, which comprise the keys of each protocol participant, $\sigma.\mathsf{init}$ and $\sigma.\mathsf{resp}$, as well as their nonces, $\sigma.\mathsf{shares}_{\mathsf{init}}$ and $\sigma.\mathsf{shares}_{\mathsf{resp}}$. The session key is defined as $\sigma.\mathsf{key} \triangleq \mathsf{key}_{\mathsf{senc}}\ [\mathsf{pkey}\ \sigma.\mathsf{init}; \mathsf{pkey}\ \sigma.\mathsf{resp}; \sigma.\mathsf{shares}_{\mathsf{init}}; \sigma.\mathsf{shares}_{\mathsf{resp}}]$.

Theorem 4.1. *Define* $\mathsf{session}_{\mathsf{NSL}}\ sk_I\ sk_R\ \sigma$ *as* $sk_I = \sigma.\mathsf{init} * sk_R = \sigma.\mathsf{resp} * \Box(\mathsf{public}\ \sigma.\mathsf{key} \iff \triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R))$. *Assuming that the message predicates of Figure 8 are set up, the following triples hold:*

$\{\mathsf{True}\}\ \mathsf{initiator}\ sk_I\ (\mathsf{pkey}\ sk_R)$

$$\left\{ r,\ r = \mathsf{None} \vee \exists \sigma, r = \mathsf{Some}\ \sigma.\mathsf{key} * \mathsf{session}_{\mathsf{NSL}}\ sk_I\ sk_R\ \sigma * \mathsf{token}\ \sigma.\mathsf{init}\ \top \right\}$$

$\{\mathsf{True}\}\ \mathsf{responder}\ sk_R$

$$\left\{ r,\ r = \mathsf{None} \vee \exists sk_I\ \sigma, r = \mathsf{Some}\ (\mathsf{pkey}\ sk_I, \sigma.\mathsf{key}) * \mathsf{session}_{\mathsf{NSL}}\ sk_I\ sk_R\ \sigma * \mathsf{token}\ \sigma.\mathsf{resp}\ \top \right\}$$

Let us dissect this result. We focus on the initiator, since the responder is similar. If the protocol successfully terminates, the function returns the session key exchanged by the two agents. The predicate $\mathsf{session}_{\mathsf{NSL}}\ sk_I\ sk_R\ \sigma$ says that the session key $\sigma.\mathsf{key}$ is public if and only if one of the long-term secret keys is known by the attacker.

The proof of for the initiator is outlined in Figure 8, along with the required message predicates. (Note that the third predicate is trivial. We will come back to this point later.) We focus on the case where every operation succeeds since the specification does not impose any requirements when the function fails. We use the MkNonce rule to generate a fresh nonce a such that $\mathsf{public}\ a \iff \triangleright \Box(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R)$. This nonce comes with a token resource, which we will use in the postcondition. We encrypt the first message using the rule for aenc (Figure 3). To prove its precondition, assuming that the message invariants have been allocated appropriately, we must show that $[\varphi_{\$\mathtt{m1}}]_{\mathsf{aenc}}\ sk_R\ [a; pk_I]$ holds. Since a is not known to be public, we proceed by proving its second disjunct; that is: (1) proving $\varphi_{\$\mathtt{m1}}\ sk_R\ [a; pk_I]$ and (2) proving $\mathsf{public}\ sk_R \Rightarrow \mathsf{public}\ a * \mathsf{public}\ pk_I$. Both points follow from how a was generated, and because every key for asymmetric encryption is

$$\varphi_{\$\mathtt{m1}}\ sk_R\ m_1 \triangleq \exists\, a\, sk_I, m_1 = [a; \mathsf{pkey}\ sk_I] * (\mathsf{public}\ a \iff \triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R))$$
$$\varphi_{\$\mathtt{m2}}\ sk_I\ m_2 \triangleq \exists\, a\, b\, sk_R, m_2 = [a; b; \mathsf{pkey}\ sk_R] * (\mathsf{public}\ b \iff \triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R))$$
$$\varphi_{\$\mathtt{m3}}\ sk_R\ m_3 \triangleq \mathsf{True}$$
$$P \triangleq \square(\mathsf{public}\ a \iff \triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R)) * \mathsf{token}\, a\, \top$$
$$Q \triangleq P * (\mathsf{public}\ b \iff \triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R))$$

$\{\mathsf{True}\}$
let $a = \mathsf{mk\_nonce}\ ()$ in
$\{(\mathsf{public}\ a \iff \triangleright\square(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R)) * \mathsf{token}\, t\, \top\}$
$\Rightarrow \{P * [\varphi_{\$\mathtt{m1}}]_{\mathsf{aenc}}\ sk_R\ [a; pk_I]\}$
let $m_1 = \mathsf{aenc}\ pk_R\ \$\mathtt{m1}\ [a; pk_I]$ in
$\{P * \mathsf{public}\ m_1\}$
send m_1;
$\{P\}$
let $m_2 = \mathsf{recv}\ ()$ in
$\{P * \mathsf{public}\ m_2\}$
let $[=a; b; =pk_R] = \mathsf{adec}\ sk_I\ \$\mathtt{m2}\ m_2$ in
$\{P * [\varphi_{\$\mathtt{m2}}]_{\mathsf{aenc}}\ sk_I\ [a; b; pk_R]\}$
$\Rightarrow \{P * (\mathsf{public}\ b \iff \triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R))\}$
$\Rightarrow \{Q * [\varphi_{\$\mathtt{m3}}]_{\mathsf{aenc}}\ sk_R\ b\}$
let $m_3 = \mathsf{aenc}\ pk_R\ \$\mathtt{m3}\ b$ in
$\{Q\}$
let $k = \mathsf{key}_{\mathsf{senc}}\ [pk_I; pk_R; a; b]$ in
$\{\exists \sigma, k = \sigma.\mathsf{key} * \mathsf{session}_{\mathsf{NSL}}\ sk_I\ sk_R\ \sigma * \mathsf{token}\, \sigma.\mathsf{init}\, \top\}$

Fig. 8. Proof for the NSL initiator, message predicates and abbreviations.

public. After the message is encrypted, it is considered public, so it can be safely sent to the network. Now, consider what happens when the initiator receives m_2. Since the message is public, after decrypting and checking it, we prove $[\varphi_{\$\mathtt{m2}}]_{\mathsf{aenc}}\ sk_I\ [a; b; pk_R]$ holds (cf. the rule for adec in Figure 3). This entails $\mathsf{public}\ b \iff \triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R)$; indeed, we have two cases to consider. One possibility is that the message predicate holds. This implies the equivalence directly. The other possibility is that the body of the message (that is, the nonces a and b) is public, which could happen if m_2 was sent by an attacker. Because a is also public, it must be the case that $\triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R)$ holds. Since public b also holds, the equivalence holds as well. This equivalence proves that the last message can be safely encrypted and sent (intuitively, because b can be read by the responder). To conclude, we need to prove that the session key k has the desired secrecy. This follows trivially from the equivalence $\mathsf{public}\ a \iff \mathsf{public}\ b \iff \triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R)$.

4.2 What About the Third Message?

Our proof of NSL did not require any particular properties of the third message. In fact, whether the third message is needed or not depends on what the protocol is being used for. Its original purpose, as devised by Needham and Schroeder [1978], was to establish an authenticated interactive session, which also seems to be the goal of most authentication protocols. Assuming that the initiator sends the first message of the session, this means that the third handshake message is redundant, because it does not convey new information. All we need to know is that the session key is known only to the relevant parties and suitably fresh to prevent replay attacks. As we will see, metadata tokens provide a mechanism to argue about freshness: since we cannot own a token that overlaps with existing metadata (cf. Figure 1), we can guarantee that a fresh key k does not belong to a set of old keys K by ensuring that all keys in K are generated from nonces whose tokens have been used.

```
let check_key_secrecy session_key =
  let guess = recv () in
  assert (session_key != guess)

let rec do_init keysI skI pkR =
  fork (fun () -> do_init keysI skI pkR);
  (* Attacker chooses responder *)
  let pkR' = recv () in
  (* Run handshake *)
  let k = init skI pkR' in
  (* The session key should be fresh and *)
  assert (not (Set.mem keysI k));
  Set.add keysI k;
  (* if attacker chose honest responder,
     the key cannot be guessed. *)
  if pkR' == pkR then check_key_secrecy k
  else ()

let rec do_resp keysR skR pkI =
  (* Similar to initiator *)
  (* ... *)

let game () =
  (* Generate keys and leak public keys *)
  let skI, skR =
    mk_aenc_key (), mk_aenc_key () in
  let pkI, pkR = pkey skI, pkey skR in
  send pkI; send pkR;
  (* Generate sets of session keys *)
  let keysI = Set.new () in
  let keysR = Set.new () in
  (* Run agents *)
  fork (fun () -> do_init keysI skI pkR);
  fork (fun () -> do_resp keysR skR pkI)
```

Fig. 9. A security game where the attacker tries to learn the session keys or cause them to be reused.

However, we could consider alternative scenarios where the protocol authenticates a *single* request or message. For example, in a protocol for financial transactions, the client might not need to establish a whole interactive session with the server just to send one request. Suppose that the initiator embeds the request data d in the last handshake message, which would serve as a confirmation step to transfer funds to a vendor or carry out whatever other action is requested. We could modify the predicate of the third message to include an escrow [Kaiser et al. 2017; Turon et al. 2014] that would allow trading in one of b's tokens against a resource $P\ d$ associated with the client's data. Logically, this resource could provide the server with the necessary permissions to carry out the operation on the client's behalf. This idea would also make sense in hybrid scenarios, such as in the early data extension of TLS 1.3, where the handshake messages can be used to carry some application-level data before the regular message exchange begins. In Section 6, we will discuss in more detail how escrows in Cryptis enable the transfer of resources through messages.

The guarantees of the third message are also tied to another point discussed earlier: the temporal aspect of authentication. In most tools for protocol verification, the specification of an authentication protocol includes trace properties stating that various belief events logged by the agents occur in a certain order. If we were interested in adding such temporal guarantees to Theorem 4.1, we could strengthen the third predicate to communicate to let the responder know that the initiator confirmed the handshake. We chose not to follow this approach because it would complicate the specifications and because it was not needed to verify the applications we were interested in, just like specifications of imperative code rarely mention the precise order of memory operations it performs, focusing instead on the observable behavior that it produces.

4.3 Game Security for NSL

Because Theorem 4.1 does not involve the traditional temporal properties used in protocol verification, we might worry that it might be missing attacks. To increase our confidence in this result, we use a symbolic security game (Figure 9). We generate keys for two honest participants, an initiator and a responder, and let them run an arbitrary number of parallel sessions. In each iteration of `do_init`, the initiator attempts to contact an agent chosen by the attacker. If the handshake successfully terminates, the initiator adds the exchanged key to a set of keys `keysI`, while ensuring

$$I \to M : \mathsf{aenc}\ pk_M\ \$\mathtt{m1}\ [a; pk_I] \qquad R \to M : \mathsf{aenc}\ pk_I\ \$\mathtt{m2}\ [a; b] \qquad I \to M : \mathsf{aenc}\ pk_M\ \$\mathtt{m3}\ b$$
$$M \to R : \mathsf{aenc}\ pk_R\ \$\mathtt{m1}\ [a; pk_I] \qquad M \to I : \mathsf{aenc}\ pk_I\ \$\mathtt{m2}\ [a; b] \qquad M \to R : \mathsf{aenc}\ pk_R\ \$\mathtt{m3}\ b.$$

Fig. 10. Attack on the original Needham-Schroeder protocol [Lowe 1996].

that it is fresh. Moreover, if the initiator contacted the honest responder, the attacker tries to guess the session key. The logic in `do_resp` is similar. The agents win the game if no checks fail.

Providing this kind of guarantee can be elusive. The original version of the NSL protocol [Needham and Schroeder 1978] was vulnerable to a man-in-the-middle attack [Lowe 1996], even though it was thought to be secure for several years (and even verified [Burrows et al. 1990]). The issue was that the original version omitted the identity of the responder in m_2—that is, m_2 would have been $\mathsf{aenc}\ pk_I\ \$\mathtt{m2}\ [a; b]$ instead of $\mathsf{aenc}\ pk_I\ \$\mathtt{m2}\ [a; b; pk_R]$. This meant that the initiator had no way of telling if the responder was actually allowed to see the nonce b. Indeed, the second predicate ties the confidentiality of b to the secret key of the responder, and this property is required to prove that the third message can be sent. If the responder's identity were not explicitly mentioned, it would be impossible to know who can see b, so it would be impossible to prove that the third message is safe.

As seen in Figure 10, a malicious responder M can exploit this issue to lead an honest R into generating a nonce b for authenticating I, and then tricking I into leaking this nonce to M. In the end, M is able to construct the same session key that R believes is being used to talk to I—despite the fact that R believes that the handshake was performed between two agents that are, in fact, honest. The game shows that the attack cannot succeed—otherwise, `check_key_secrecy` would fail.

To show that the attacker cannot win the game, we proceed as follows. First, we prove specifications for the functions `do_init` and `do_resp` that guarantee that they are safe. We consume the secrecy resources of the agents' private keys to guarantee that they cannot become public. In the proof of `do_init`, we invoke the specification of `initiator` in Theorem 4.1. We maintain an invariant on `keysI` saying that every key $k' = \sigma.\mathsf{key}$ stored in the set satisfies $\sigma.\mathsf{shares}_{\mathsf{init}} \mapsto_{\$\mathsf{sess}} ()$. This means that the new session key k cannot be in the set, because its corresponding token has not been used yet. Thus, the first assertion cannot fail. We consume this token so that the key can be added to `keysI`. We then argue that the second assertion cannot fail because the attacker's guess is public, whereas the session key cannot be because the agents are honest. A symmetric reasoning shows that `do_resp` is safe as well. Finally, we prove that `game` is safe. We generate the keys of the honest participants by invoking the specifications in Section 2. Then, we allocate two empty sets of keys, which trivially satisfy the invariant that all keys have their metadata token set. We conclude by invoking the specifications of `do_init` and `do_resp` to show that the last line is safe.

5 Diffie-Hellman Key Exchange and Forward Secrecy

One limitation of a protocol like NSL is being vulnerable to *key compromise*. If a private key is leaked, an attacker can decrypt the handshake messages and learn its session key. By contrast, many modern protocols guarantee *forward secrecy*: if a handshake is successful, its session keys will remain secret even if long-term keys are leaked [Cohn-Gordon et al. 2016]. Our goal in this section is to demonstrate that Cryptis can scale up to such richer guarantees. Specifically, we will prove the correctness of the ISO protocol [Krawczyk 2003], which provides forward secrecy. Because of its stronger guarantees, it will be our protocol of choice to implement the communication components used in our key-value store. A typical run of the protocol proceeds as follows:

$$I \to R : [g^a; pk_I] \qquad R \to I : \mathsf{sign}\ sk_R\ \$\mathtt{m2}\ [g^a; g^b; pk_I] \qquad I \to R : \mathsf{sign}\ sk_I\ \$\mathtt{m3}\ [g^a; g^b; pk_R].$$

```
let initiator skI pkR =
  let pkI = pkey skI in
  let a = mk_nonce () in
  send [g^a; pkR];
  let [=g^a; gb; =pkI] =
    verify pkR $m2 (recv ()) in
  send (sign skI $m3 [g^a; gb; pkR]);
  let k =
    key senc [pkI; pkR; g^a; gb; gb^a] in
  k
```

```
let responder skR =
  let pkR = pkey skR in
  let [ga; pkI] = recv () in
  let b = mk_nonce () in
  send (sign skR $m2 [ga; g^b; pkI]);
  let [=ga; =g^b; =pkR] =
    verify pkI $m3 (recv ()) in
  let k =
    key senc [pkI; pkR; ga; g^b; ga^b] in
  (pkI, k)
```

Fig. 11. ISO authentication protocol based on Diffie-Hellman key exchange.

$$\varphi_{\$m2}\ sk_R\ m_2 \triangleq \exists s_a\ b\ pk_I, m_2 = [s_a; g^b; pk_I] * (\text{public } b \iff \triangleright \text{False})$$
$$\varphi_{\$m3}\ sk_I\ m_3 \triangleq \exists a\ s_b\ sk_R, m_3 = [g^a; s_b; \text{pkey } sk_R] * (\text{public } sk_I \vee \text{public } sk_R \vee (\text{public } (\text{key}_{\text{senc}}\ [\text{pkey } sk_I; \text{pkey } sk_R; g^a; s_b; s_b^a]) \mathrel{-\!\!*} \triangleright \text{False}))$$

Fig. 12. Message predicates for ISO protocol.

The flow is similar to the NSL protocol, except that (1) it uses digital signatures instead of asymmetric encryption; (2) the first message does not need to be signed or encrypted; (3) the keys used in the signed messages 2 and 3 are swapped; (4) the agents exchange the key shares g^a and g^b rather than the nonces a and b. At the end of the handshake, the participants can compute the shared secret $g^{ab} = (g^a)^b = (g^b)^a$ and use it to derive a session key. Figure 11 shows an implementation of ISO.

We proceed following the blueprint laid out in Section 4. We formulate a specification for the initiator and the responder, and use these specifications to prove the security of a game between the attacker and the agents. The main difference lies in the secrecy guarantees for the session key k: when the handshake terminates, if we can prove that the participants' long-term keys are not compromised *yet*, then k will remain secret forever, even if some long-term keys are leaked later. The ISO session σ now includes a component σ.secret, which corresponds to the shared Diffie-Hellman secret. We define σ.key as $\text{key}_{\text{senc}}\ [\sigma.\text{init}; \sigma.\text{resp}; \sigma.\text{shares}_{\text{init}}; \sigma.\text{shares}_{\text{resp}}; \sigma.\text{secret}]$.

Theorem 5.1. *Define* $\text{session}_{\text{ISO}}\ sk_I\ sk_R\ \sigma$ *as* $sk_I = \sigma.\text{init} * sk_R = \sigma.\text{resp} * (\text{public } sk_I \vee \text{public } sk_R \vee \square(\text{public } \sigma.\text{key} \iff \triangleright \text{False}))$. *The following triples hold:*

$\{\text{True}\}\ \text{initiator}\ sk_I\ (\text{pkey}\ sk_R)$

$$\left\{ \begin{array}{l} r,\ r = \text{None} \vee \exists\, \sigma, r = \text{Some } \sigma.\text{key} \\ \qquad * \text{session}_{\text{ISO}}\ sk_I\ sk_R\ \sigma * \text{token}\ \sigma.\text{init}\ \top \end{array} \right\}$$

$\{\text{True}\}\ \text{responder}\ sk_R$

$$\left\{ \begin{array}{l} r,\ r = \text{None} \vee \exists\, sk_I\ \sigma, r = \text{Some } (\text{pkey } sk_I, \sigma.\text{key}) \\ \qquad * \text{session}_{\text{ISO}}\ sk_I\ sk_R\ \sigma * \text{token}\ \sigma.\text{resp}\ \top \end{array} \right\}$$

We use the predicates of Figure 12. Each agent allocates their nonces n so that public $n \iff \triangleright$ False. When I checks the signature, either R is compromised, or they learn that R's key share is of the form g^b, with public $b \iff \triangleright$ False. Since public $a \iff \triangleright$ False, Figure 1 implies that public g^{ab} is equivalent to $\triangleright$ False. We modify the game of Figure 9 so that both signature keys are eventually leaked, and we only check a session key if it was exchanged before the compromise (Figure 13). To prove security, we proceed similarly to what we did earlier. The main difference is the management of long-term keys. After generating the sk_I and sk_R, we allocate an invariant J that says that either the compromise bit c is set to false, in which case secret sk_I * secret sk_R holds, or it is set to true, in which case both sk_I and sk_R are public. Then, we prove that the `check_key_secrecy`

```
let rec wait_for_compromise c =
  if not !c then wait_for_compromise c

let check_key_secrecy c k =
  if not !c then
    wait_for_compromise c;
    let guess = recv () in
    assert (k != guess)
  else ()

let compromise_keys c skI skR =
  c := true; send skI; send skR
```

```
let game () =
  (* ... *)
  let skI = mk_sign_key #() in
  let skR = mk_sign_key #() in
  let pkI = pkey skI in
  let pkR = pkey skR in
  let c = ref false in
  (* ... *)
  fork (fun () -> do_init keysI c skI pkR);
  fork (fun () -> do_resp keysR c skR pkI);
  fork (fun () -> compromise_keys c skI skR)
```

Fig. 13. Security game for the ISO protocol (excerpt).

function is safe provided that it is called on a session key k of the ISO protocol. If we run the "then" branch of that function, the invariant J, combined with the postcondition of the handshake, implies that $\square(\mathsf{public}\ k \iff \triangleright\,\mathsf{False})$ holds. This guarantees that the attacker cannot win.

5.1 Extensions

We now discuss several extensions of the base protocol and specification that make them easier to reuse in other settings. Figure 14 lists auxiliary predicates and rules that we will use.

Decomposing the Responder. In a typical client/server setting, it is useful to decompose R's logic into two steps. In the `ISO.listen` function, the responder waits for an incoming connection request, the first message of the ISO protocol. The server can use the initiator's identity to decide whether to accept the connection or not. If it decides to accept the connection, it can call the `ISO.confirm` function, which generates the responder's key share and runs the rest of the ISO handshake.

Session Compromise. The specification of ISO has a limitation: if the handshake completes successfully, it is impossible for us to model the compromise of the session key k, because the session key is secret forever. We can relax this limitation by modifying the secrecy predicates of the private DH keys a and b. Let $\mathsf{release\_token}\ t \triangleq \mathsf{token}\ t\ (\uparrow\texttt{\$ISO.released})$ and $\mathsf{released}\ t \triangleq t \mapsto_{\texttt{\$ISO.released}} ()$. We define $\mathsf{public}\ a$ and $\mathsf{public}\ b$ as $\triangleright(\mathsf{released}\ g^a * \mathsf{released}\ g^b)$. Intuitively, released marks whether that can be treated as compromised from the point of view of one of the parties. To create this resource, the agent must consume a matching release_token resource, which is generated once the key shares are created. Then, we can model a compromise of the session key by simply releasing the tokens of the initiator and the responder. While the agents still hold their release tokens, we can prove that the key is not yet compromised.

Early Compromise. Conversely, if we know that one of the agents is already compromised before the handshake, it is useful for the session key k to be public from the start. Then, if we use the key to encrypt something (cf. Section 6), we do not need to prove the corresponding message predicates. Our extended ISO specifications allow us to make k public in this scenario. We add a parameter $\rho \in \{\mathsf{init}, \mathsf{resp}\}$ to the session predicate, which tracks whether the agent of role ρ was able to compromise the session early. The predicate $\mathsf{compromised}\ \rho\ \sigma$ holds if the handshake was compromised from ρ's perspective—that is, that agent learned that one of the private keys was compromised before the end of the handshake.

Extended Specification. With all these extensions, we can strengthen Theorem 5.1 as follows. (We include only the specification for the initiator; the specification for the responder is similar.)

$$\text{compromised}\ \rho\ \sigma \vdash \text{public}\ \sigma.\text{key} \qquad \Sigma * \text{compromised}\ \rho\ \sigma \vdash \text{public}\ sk_C \vee \text{public}\ sk_S$$

$$\text{release\_token}\ t * \text{released}\ t \twoheadrightarrow \text{False} \qquad \Sigma * \text{secret}\ sk_C * \text{secret}\ sk_S \vdash \triangleright \Box \neg\text{compromised}\ \rho\ \sigma$$

$$\text{release\_token}\ t \Rrightarrow \text{released}\ t \qquad \Sigma * \triangleright \text{released}\ \sigma.\text{shares}_{\text{init}} * \triangleright \text{released}\ \sigma.\text{shares}_{\text{resp}} \vdash \text{public}\ \sigma.\text{key}$$

$$\Sigma * \text{release\_token}\ \sigma.\text{shares}_\rho * \text{public}\ \sigma.\text{key} \vdash \triangleright \text{compromised}\ \rho\ \sigma$$

Fig. 14. Properties of handshake compromise. We abbreviate $\text{session}_{\text{ISO}}\ sk_C\ sk_S\ \rho\ \sigma$ as Σ.

Theorem 5.2. *Let b be an arbitrary boolean. Assuming that the appropriate signature predicates are allocated, the triple* $\{b \Rightarrow \text{public}\ sk_I \vee \text{public}\ sk_R\}\ \text{initiator}\ sk_I\ (\text{pkey}\ sk_R)\ \{r, \varphi\ r\}$ *is valid, where* $\varphi\ r$ *is* $r = \text{None} \vee \exists\, \sigma, r = \text{Some}\ \sigma.\text{key} * \text{session}_{\text{ISO}}\ sk_I\ sk_R\ \text{init}\ \sigma * \text{release\_token}\ \sigma.\text{shares}_{\text{init}} * \Box(b \Rightarrow \text{compromised init}\ \sigma) * \text{token}\ \sigma.\text{init}\ (\top \setminus \uparrow\texttt{\$ISO})$.

6 Reliable Connections

Now that we have authentication, we can use it to implement authenticated, reliable connections. At the logic level, we follow prior work and model this functionality as the ability to reliably transfer arbitrary separation-logic resources [Gondelman et al. 2023; Hinrichsen, Bengtson, et al. 2020]. Operationally, the functionality guarantees that messages are received in the same order that they are sent and that their contents are not modified. To preserve their order, we include sequence numbers in every message sent; to preserve their contents, we encrypt them with a session key.

The functionality is described in Figure 15. By abuse of notation, we sometimes use a connection object c as if it were its underlying session σ. In particular, $c.\text{shares}_\rho$ refers to the key share of the agent of role ρ. There is no harm in doing that because the connection object tracks the session key, which fully determines all the session information. To connect to a server, a client uses the connect function, which initiates an ISO handshake and stores the resulting session key in the returned connection object, along with counters for tracking sequence numbers. The server behaves similarly, but runs the responder of the protocol. Once a connection is established, we can use the send and recv functions to communicate. These functions include and check sequence numbers to ensure that messages are received in the appropriate order. The recv function keeps polling the network until it receives a message with the expected tag and sequence number.

Let us analyze these specifications. As we mentioned earlier, it will be useful to let the connection functions create compromised connections if we know that one of the participants is also compromised. Accordingly, the precondition of the functions connect and confirm assumes that either the agents have been compromised or some resource P is available. When the connection is established, it will be marked as compromised if the first case holds; otherwise, the resource P will be available for use. This allows us to maintain an resource P across multiple connections, provided that the protocol participants are not compromised. Moreover, the postconditions provide release tokens to compromise session keys and a metadata token. Finally, the postconditions provide the resource $\text{Conn.connected}\ sk_C\ sk_S\ \rho\ c$, which says that the connection is ready. To send and receive messages, we must have assigned special $\text{conn\_pred}\ \rho\ \varphi$ predicates to their tags. For simplicity, we assume that each tag can be used to send messages for a single role ρ. Then, to send a message $\vec{t}$, we must prove that $\varphi\ sk_C\ sk_S\ c\ \vec{t}$ holds, unless the session key is compromised. Dually, we can assume that this predicate holds when receiving the message. Crucially, these predicates are *not* required to hold persistently, which allows us to transfer resources through a connection. To enable this transfer of resources, we use a variant of the *escrow pattern* [Gondelman et al. 2023; Kaiser et al.

```
let Conn.listen () = ISO.listen ()

let Conn.confirm skR request =
  let k = ISO.confirm skR request in
  {session_key = k;
   sent = 0; received = 0}

let Conn.send conn s m =
  let ciphertext =
    senc conn.session_key s
      [conn.sent; m] in
  conn.sent++;
  send ciphertext
```

```
let Conn.connect skI pkR =
  let k = ISO.initiator skI pkR in
  {session_key = k; sent = 0; received = 0}

let Conn.recv conn s =
  let rec loop () =
    let m = recv () in
    let [n; payload] =
      sdec c.session_key s m in
    if n == c.received then
      c.received++; payload
    else loop ()
  in loop ()
```

ConnConnect

$$\frac{\mathcal{E} = \top \setminus \uparrow\$\mathtt{ISO} \setminus \uparrow\$\mathtt{Conn}}{\{\text{public } sk_C \vee \text{public } sk_S \vee P\}\ \text{Conn.connect } sk_C\ (\text{pkey } sk_S)\ \left\{ c,\ \begin{array}{l} \text{Conn.connected } sk_C\ sk_S\ \text{init}\ c \\ * (\text{compromised init } c \vee P) \\ * \text{release\_token } c.\text{shares}_{\text{init}} \\ * \text{token } c.\text{init}\ \mathcal{E} \end{array} \right\}}$$

ConnConfirm

$$\frac{\mathcal{E} = \top \setminus \uparrow\$\mathtt{ISO} \setminus \uparrow\$\mathtt{Conn}}{\left\{ \begin{array}{l} \text{public } ga \\ * (\text{public } sk_C \vee \text{public } sk_S \vee P) \end{array} \right\}\ \text{Conn.confirm } sk_S\ (ga, \text{pkey } sk_C)\ \left\{ c,\ \begin{array}{l} \text{Conn.connected } sk_C\ sk_S\ \text{resp}\ c \\ * (\text{compromised resp } c \vee P) \\ * \text{release\_token } c.\text{shares}_{\text{resp}} \\ * \text{token } c.\text{resp}\ \mathcal{E} \end{array} \right\}}$$

ConnSend

$$\frac{\text{senc} \mapsto_{\mathcal{N}} \text{conn\_pred } \rho\ \varphi}{\left\{ \begin{array}{l} \text{Conn.connected } sk_C\ sk_S\ \rho\ c \\ * \text{public } \vec{t} * (\text{public } c.\text{key} \vee \varphi\ sk_C\ sk_S\ c\ \vec{t}) \end{array} \right\}\ \text{Conn.send } c\ \mathcal{N}\ \vec{t}\ \{\text{Conn.connected } sk_C\ sk_S\ \rho\ c\}}$$

ConnRecv

$$\frac{\text{senc} \mapsto_{\mathcal{N}} \text{conn\_pred } \rho^{-1}\ \varphi}{\{\text{Conn.connected } sk_C\ sk_S\ \rho\ c\}\ \text{Conn.recv } c\ \mathcal{N}\ \left\{ \vec{t},\ \begin{array}{l} \text{Conn.connected } sk_C\ sk_S\ \rho\ c \\ * \text{public } \vec{t} * (\text{public } c.\text{key} \vee \varphi\ sk_C\ sk_S\ c\ \vec{t}) \end{array} \right\}}$$

Fig. 15. Implementation and specification of reliable communication. The variable $\rho \in \{\text{init}, \text{resp}\}$ denotes the role of an agent, and ρ^{-1} denotes the opposite role.

2017; Turon et al. 2014]. The idea is to use an invariant J to allow an agent to extract a resource R by exchanging it against a guard G. Because invariants are persistent, they can be used in the proofs of message predicates.

Figure 16 presents the definitions of the predicates used in the specifications. The resource Conn.connected $sk_C\ sk_S\ \rho\ c$ contains an assertion $\boxed{\bullet m}^{c.\text{shares}_\rho}_{\$\text{conn.recv}}$ which tracks how many messages that agent has received. This uses a monotonic counter resource algebra, which combines the authoritative resource algebra [Jung, Krebbers, Jourdan, et al. 2018] with the monoid $(\mathbb{N}, \max, 0)$. An element of the form $\bullet m$ represents ownership of a monotonic counter that is set to m, whereas

$$\begin{aligned}
\text{Conn.connected } sk_C\ sk_S\ \rho\ c &\triangleq \text{session}_{\text{ISO}}\ sk_C\ sk_S\ \rho\ c \\
&\quad * \exists n\, m, c \mapsto \{key = k; sent = n; received = m\} * \boxed{\bullet m}^{c.\text{shares}_\rho}_{\$\text{conn.recv}} \\
\text{conn_pred } \rho\ \varphi\ k\ t &\triangleq \exists \sigma\, n\, \vec{t}, k = \sigma.\text{key} * t = (n :: \vec{t}) * \text{public } \vec{t} \\
&\quad * (\boxed{\bullet n}^{\sigma.\text{shares}_{\rho^{-1}}}_{\$\text{conn.recv}} \Rrightarrow_\top \triangleright(\varphi\ \sigma.\text{init}\ \sigma.\text{resp}\ \sigma\ \vec{t} * \boxed{\bullet n+1}^{\sigma.\text{shares}_{\rho^{-1}}}_{\$\text{conn.recv}}))
\end{aligned}$$

Fig. 16. Predicates for reliable connections

$$\begin{aligned}
\text{resp_pred_token}_q\ \sigma\ \varphi &\triangleq \boxed{\text{Agree}_q\ \varphi}^{\sigma.\text{shares}_{\text{init}}}_{\$\text{rpc.resp_pred}} \\
\text{rpc_pred } \varphi\ \psi &\triangleq \text{conn_pred init} \left(\begin{array}{l} \lambda sk_C\ sk_S\ \sigma\ \vec{t}, \text{resp_pred_token}_{1/2}\ \sigma\ (\psi\ sk_C\ sk_S\ \sigma\ \vec{t}) \\ * \varphi\ sk_C\ sk_S\ \sigma\ \vec{t} \end{array} \right) \\
\text{resp_pred } sk_C\ sk_S\ \sigma\ \vec{t} &\triangleq \exists \psi, \text{resp_pred_token}_{1/2}\ \sigma\ \psi * \psi \vec{t} \\
\text{RPC.connected } sk_C\ sk_S\ c &\triangleq \text{Conn.connected } sk_C\ sk_S\ \text{init}\ c * \text{release_token}\ c.\text{shares}_{\text{init}} \\
&\quad * (\text{compromised init}\ c \vee \text{resp_pred_token}_1\ \sigma\ (\lambda_, \text{False}))
\end{aligned}$$

$$\textsc{RpcCall}\ \frac{\begin{array}{c}\text{senc} \mapsto_{\mathcal{N}} \text{rpc\_pred}\ \varphi\ \psi \\ \text{senc} \mapsto_{\$\text{rpc.resp}} \text{conn\_pred init resp\_pred}\end{array}}{\left\{\begin{array}{l}\text{public } \vec{t} * \text{RPC.connected } sk_C\ sk_S\ c \\ * (\text{compromised init}\ c \vee \varphi\ sk_C\ sk_S\ c\ \vec{t})\end{array}\right\} \quad \text{RPC.call } c\ \mathcal{N}\ \vec{t} \quad \left\{\begin{array}{l}\vec{t}', \text{public } \vec{t}' * \text{RPC.connected } sk_C\ sk_S\ c \\ * (\text{compromised init}\ c \vee \psi\ sk_C\ sk_S\ c\ \vec{t}\ \vec{t}')\end{array}\right\}}$$

$$\textsc{RpcClose}\ \frac{}{\{\text{RPC.connected } sk_C\ sk_S\ c\} \quad \text{RPC.close } c \quad \{\text{public } c.\text{key}\}}$$

Fig. 17. Remote procedure calls

$\circ n$ means that the counter's value is at least n. The message predicate conn_pred $\rho\ \varphi$ contains an escrow that allows us to trade in that guard against resources attached to the message payload, provided that the guard's counter matches the sequence number of the message. Note that this escrow also returns an updated guard, signaling the fact that another message was received.

To prove the specification of send, we must be able to prove that conn_pred $\rho\ \varphi$ holds of c.key and $n :: \vec{t}$ under the preconditions of that function. Let us assume that $\varphi\ sk_C\ sk_S\ c\ \vec{t}$ holds; otherwise, the session key is public and the encrypted message is trivially public. We consume that resource to create an invariant $J \triangleq \boxed{\boxed{\circ n+1}^{c.\text{shares}_\rho}_{\$\text{conn.recv}} \vee \varphi\ c.\text{init}\ c.\text{resp}\ c\ \vec{t}}$. This invariant allows us to prove the implication $\boxed{\bullet n}^{c.\text{shares}_{\rho^{-1}}}_{\$\text{conn.recv}} \Rrightarrow_\top \triangleright(\varphi\ sk_C\ sk_S\ c\ \vec{t} * \boxed{\bullet n+1}^{c.\text{shares}_{\rho^{-1}}}_{\$\text{conn.recv}})$ persistently. Suppose that we are given $\boxed{\bullet n}^{c.\text{shares}_{\rho^{-1}}}_{\$\text{conn.recv}}$. If we open J, the first disjunct is contradictory, because it states that the counter has already passed the value $n+1$. This allows us to extract the second disjunct to prove the conclusion. Finally, we can reestablish J by bumping the counter and proving its first disjunct.

7 Remote Procedure Calls

The last internal component we need for our key-value store is the RPC mechanism. The component is a thin layer on top of reliable connections (cf. Figure 15). The connection stage is mostly unchanged. When the server accepts a connection from a client C, it enters a loop that continuously receives requests from C. To perform a call (RPC.call), the client sends a message with the appropriate

tag (which identifies the server operation) and arguments and then waits for the corresponding response. The server invokes a handler upon receiving this request based on the operation and sends whatever the handler returns back to the client.

Each RPC operation comes with two predicates: a predicate φ that should hold of the arguments of the operation, and a predicate ψ for its return values. It is convenient for ψ to be able to refer to the arguments in addition to the results. Here, the ability to transfer resources with reliable connections comes in handy. Using a fractional agreement algebra and term metadata, we define a resource $\mathsf{resp\_pred\_token}_q\ \sigma\ \psi'$, which keeps track of which property ψ' should hold of the results. The RPC client allocates this resource when the connection is established. Before performing a call, the client updates ψ' to $\psi\ sk_C\ sk_C\ \sigma\ \vec{t}$, where $\vec{t}$ is the list containing the arguments of the operation. Then, it sends one half of this token to the server. The message predicate for the server's response says that the results of the operation should satisfy exactly this predicate. The RPC functionality also includes a close call for the client to close the connection. The client consumes its release token and informs the server that the token has been released. The server releases its token as well, at which point the session key becomes public. This allows the server to reply to the client without proving any particular message predicates. When the client receives the server's acknowledgment, they conclude that the session key has been made public.

8 Implementing and Verifying the Key-Value Store

With all the communication primitives and data structures in place, implementing the key-value store is straightforward. The server sits in a loop waiting for incoming connection requests from the RPC module. When a request arrives, the server queries a directory to check if that client has an account. If the account doesn't exist yet, a new one is created. If it does, the server acquires a lock to the account and forks off a separate thread that handles that connection. Several clients can be served simultaneously, but each client can have at most one active connection, and the account lock is used to guarantee mutual exclusion. Each API call corresponds to a server handler that performs the corresponding operation on the client's database. Once the client closes the connection, the server releases the lock and kills the connection thread. The server uses a map data structure to store the account directory and the client databases. For simplicity, our implementation uses a purely functional association list stored in a location, but we could easily swap that out for a more efficient implementation.

To verify the specifications of Figure 5, we use some custom resources and RPC predicates described in Figure 18. (Once again, most of these resources are parameterized by the keys of the client and the server, but we elide most of these parameters for readability.) We distinguish between two types of databases: the *logical* database, which the client believes ought to be stored in the server, and the *physical* database, which is what is actually stored in the server. The logical database is ghost state that is owned by the client, and the physical database is tracked by a resource that says that the client's database is correctly represented as an association list.

The logical database consists of a series of resources stored in term metadata, which the client and the server can initialize by consuming the appropriate tokens (cf. DbMainAlloc, DbCopyAlloc and DbStateAlloc). The resource $\mathsf{db\_state}\ \sigma$ means that the current logical state is exactly σ. This predicate is defined with a function resource algebra, similarly to how the heap is modeled in Iris [Jung, Krebbers, Jourdan, et al. 2018]. As shown in Figure 18, it can be combined with the points-to assertion $t_1 \mapsto_{\mathsf{db}} t_2$ to update the logical state (DbStateUpdate) or find out which values are stored under it (DbStateAgree). The remaining database resources are used to transfer updates from the client to the server, and are implemented with a fractional agreement algebra. The resource $\mathsf{db\_main}\ \sigma$ tracks the client's view on the logical database, and $\mathsf{db\_copy}\ \sigma$ tracks the server's view

$$\textsc{DbStateAlloc} \quad \text{token } sk_C\ (\uparrow \texttt{\$db.client.}sk_S\texttt{.state}) \Rrightarrow \text{db\_state } \emptyset * \top \mapsto_{\text{db}} \bot$$

$$\textsc{DbStateAgree} \quad \text{db\_state } \delta * t_1 \mapsto_{\text{db}} ot_2 \twoheadrightarrow \delta\ t_1 = ot_2$$

$$\textsc{DbStateUpdate} \quad \text{db\_state } \delta * t_1 \mapsto_{\text{db}} ot_2 \Rrightarrow \text{db\_state } (\delta[t_1 \mapsto t_2']) * t_1 \mapsto_{\text{db}} t_2'$$

$$\textsc{DbMainAlloc} \quad \text{token } sk_C\ (\uparrow \texttt{\$db.client.}sk_S\texttt{.replica}) \Rrightarrow \text{db\_main } \emptyset * \text{db\_sync } \emptyset$$

$$\textsc{DbCopyAlloc} \quad \text{token } sk_S\ (\uparrow \texttt{\$db.server.}sk_C) \Rrightarrow \text{db\_copy } \emptyset$$

$$\textsc{DbMainUpdate} \quad \text{db\_main } \delta * \text{db\_sync } \delta \Rrightarrow \text{db\_main } \delta' * \text{db\_update } \delta\ \delta'$$

$$\textsc{DbCopyUpdate} \quad \text{db\_copy } \delta_1 * \text{db\_update } \delta_2\ \delta' \Rrightarrow \delta_1 = \delta_2 * \text{db\_copy } \delta' * \text{db\_sync } \delta'$$

$$\textsc{DbMainSync} \quad \text{db\_main } \delta_1 * \text{db\_sync } \delta_2 \twoheadrightarrow \delta_1 = \delta_2$$

$$\text{DB.connected } sk_C\ sk_S\ c \triangleq \exists\, \delta, \text{RPC.connected } sk_C\ sk_S\ c * \text{db\_state } \delta * (\text{compromised init } c \vee \text{db\_main } \delta * \text{db\_sync } \delta)$$

$$\text{DB.disconnected } sk_C\ sk_S \triangleq \exists\, \delta, \text{db\_state } \delta * (\text{public } sk_C \vee \text{public } sk_S \vee \text{db\_main } \delta * \text{db\_sync } \delta)$$

$$\varphi_{\texttt{\$store}}\ sk_C\ sk_S\ c\ \vec{t} \triangleq \exists t_1\ t_2\ \sigma, \vec{t} = [t_1, t_2] * \text{db\_update } \sigma\ \sigma[t_1 \mapsto t_2]$$

$$\varphi_{\texttt{\$ack\_store}}\ sk_C\ sk_S\ c\ \vec{t}\ \vec{t}' \triangleq \exists \sigma, \text{db\_sync } \sigma$$

$$\varphi_{\texttt{\$load}}\ sk_C\ sk_S\ c\ \vec{t} \triangleq \exists t_1\ t_2\ \sigma, \vec{t} = [t_1] * \sigma\ t_1 = \text{Some } t_2 * \text{db\_update } \sigma\ \sigma$$

$$\varphi_{\texttt{\$ack\_load}}\ sk_C\ sk_S\ c\ \vec{t}\ \vec{t}' \triangleq \exists t_1\ t_2\ \sigma, \vec{t} = [t_1] * \vec{t}' = [t_2] * \sigma\ t_1 = \text{Some } t_2 * \text{db\_sync } \sigma.$$

Fig. 18. Key-value store: Auxiliary assertions, rules and call predicates.

of the physical database. The resource db_sync σ indicates that these two views are in sync. When the client wants to update the logical database to σ', they consume this resource to create a new resource db_update $\sigma\ \sigma'$ with the DbMainUpdate rule. The server can use this resource to update their own view to the new state (DbCopyUpdate). The connection and disconnection predicates for the client ensure that the db_state is consistent with the update resources. In the case of a compromise, the client and the server can have inconsistent views of the logical database, in which case we do not require the update resources to be present, thus allowing their states to diverge.

Finally, to prove the specifications of Figure 5, we use RPC predicates to inform the server about which operations are performed on the logical state. We leverage the fact that the RPC abstraction can be used to transfer resources, which allows the client to send db_update resources to keep the server synchronized. When the server receives these messages, it synchronizes its copy of the logical state and applies the corresponding operations to maintain its invariant. For example, the message predicates for storing or loading a value are shown in Figure 18. In particular, the response predicate for loading a value guarantees that the value t_2 in the response is the correct value associated with the key t_1 sent in the request. Here, we make use of the fact that the predicate for the response of the load request can mention the queried key t_1; otherwise, the client wouldn't be able to tell that the value t_2 corresponds to t_1, since the key does not appear in the response.

9 Implementation and Model

Rather than formalizing the Cryptis language from scratch, we implemented it as a library in HeapLang, the main language used in Iris. We developed a small library to manipulate lists and other data structures. We formalized cryptographic terms as a separate type from HeapLang values, and rely on an explicit function to encode terms as values. Namespaces are included in terms by converting them to integers. We ensure that Diffie-Hellman terms are normalized so that their intended notion of equality coincides with equality in Rocq, similar to some encodings of quotient

Table 1. Code statistics.

Component	Impl. (loc)	Proofs (loc)	Game (loc)	Total (loc)	Wall-clock time (s)
Cryptis Core	—	—	—	8270	113
NSL (Section 4)	54	230	255	539	43
ISO (Section 5)	59	783	345	1212	54
Connections (Section 6)	79	521	—	613	27
RPC (Section 7)	52	492	—	554	25
Store (Section 8)	154	1383	161	1706	79

types in type theory [Cohen 2013]. We implemented nonces as heap locations, which allowed us to reuse much of the location infrastructure. To allocate a nonce, we simply allocate a new location, which is guaranteed to be fresh. Encoding nonces in terms of heap locations is well-suited for reasoning about protocols in the symbolic model, but it is not meant to be taken too literally—in particular, because heap allocation does not produce bit patterns with enough entropy to withstand real attackers.

To give an idea of the effort involved in Cryptis, Table 1 shows the size of our development and case studies. The "Cryptis Core" row encompasses the logic, the HeapLang libraries for manipulating terms and their specifications. The figures reported for case studies are broken down in lines of code for the HeapLang implementation, lines of code for proofs of the Cryptis specifications (aggregated with specifications and auxiliary definitions), and lines of code for defining and proving the security of games. We also include the time spent to compile the code with parallel compilation on Rocq 9.0 running on an Ubuntu 24.04 laptop with an Intel i7-1185G7 3.00GHz with eight cores and 15GiB of RAM. These statistics show that the effort required by Cryptis is comparable to other advanced tools for modular protocol verification, such as DY* [Bhargavan, Bichhawat, Do, et al. 2021b].

Differences with Respect to the Paper. We have assumed that term variables always range over terms that have been previously generated. Rocq cannot impose this restriction, so instead we have a separate minted predicate that ensures that every nonce that appears in a term has been previously allocated. Concretely, minted t says that every nonce $t' \preceq t$ satisfies $l \mapsto_{\texttt{\$minted}} ()$, where l is the nonce's underlying location, and the notation refers to the location metadata predicate. This standard Iris predicate allows us to attach metadata to individual locations in HeapLang and satisfies laws similar to those of our term metadata predicate. In fact, our term metadata was inspired by location metadata, as we discuss below.

Another difference lies in the treatment of the network. In Section 2, we modeled the network as a separate state component that is manipulated concurrently by the agents and attacker actions. In our implementation, the network is a modeled by a *channel object*, which is just a concurrent linked list stored in the heap. To access the channel, the program must acquire its lock and release it after it is done. We model the attacker as a separate set of threads that run the attacker actions described in Section 2. We maintain an invariant that the channel only contains public messages. This invariant is preserved by the honest agents because the specification of send requires a public message. It is also preserved by attacker actions because public terms are preserved by all cryptographic operations. One drawback of this encoding is that HeapLang does not have good support for global objects. Therefore, in Cryptis, every function that manipulates the channel must take the channel as a parameter. To run a Cryptis program, we must run a special `init_network` function, which allocates the channel and initializes the attacker threads.

Model of Cryptis Assertions. The main Cryptis predicates are obtained by a combination of Iris invariants and ghost state. The adequacy theorem (Theorem 2.1) is responsible for allocating these resources and setting up the required invariants. The public predicate is defined as a Rocq recursive function using the size of the term as fuel. The definition is stated to validate all the equivalences in Figure 1 directly. The only clause that is not covered by that definition is the clause for nonces. It is defined as public (nonce l) $\triangleq \exists \gamma\, \varphi, l \mapsto_{\$\mathsf{nonce}} \gamma \wedge \boxed{\varphi}^{\gamma} \wedge \triangleright \Box\, \varphi$ (nonce l), where φ ranges over predicates of type *term* → *iProp*. This is an example of higher-order ghost state [Jung, Krebbers, Birkedal, et al. 2016]. When we allocate a nonce, we also allocate a new ghost location γ to store its secrecy predicate φ, and then tie γ to the nonce by using the metadata of its underlying location l.

The definitions of message-predicate assertions and term metadata rely on the following construction. We define assertions $\mathsf{token}\ \gamma\ \mathcal{E}$, $\gamma \mapsto_{\mathcal{N}} x$ and $\boxed{a}^{\gamma}_{\mathcal{N}}$ that behave like the analogous propositions for term metadata but are indexed by ghost locations rather than terms. We can allocate a new resource $\mathsf{token}\ \gamma\ \top$ for a fresh γ at any point. (Internally, these assertions are defined using the reservation map resource algebra of Iris and adapted from the location metadata feature mentioned above.) This allows us to define message predicates and term metadata with one level of indirection:

$$\mathsf{token}\ F\ \mathcal{E} \triangleq \mathsf{token}\ \gamma_F\ \mathcal{E} \qquad F \mapsto_{\mathcal{N}} \varphi \triangleq \boxed{\varphi}^{\gamma_F}_{\mathcal{N}}$$

$$\mathsf{token}\ t\ \mathcal{E} \triangleq \exists \gamma, \mathsf{term\_name}\ t\ \gamma * \mathsf{token}\ \gamma\ \mathcal{E} \qquad t \mapsto_{\mathcal{N}} x \triangleq \exists \gamma, \mathsf{term\_name}\ t\ \gamma * \gamma \mapsto_{\mathcal{N}} x$$

In this definition, γ_F refers to a ghost name that is uniquely associated with the functionality F, whereas the assertion $\mathsf{term\_name}\ t\ \gamma$, which we will soon dive into, means that t is uniquely associated with the name γ. Message predicates are another instance of higher-order ghost state [Jung, Krebbers, Birkedal, et al. 2016]: we use a resource algebra of predicates that guarantees agreement to uniquely associate φ to γ_F and $\mathcal{N}$. One important technical point is that, because of its impredicative definition, the uniqueness of the predicate is only guaranteed under a $\triangleright$ (cf. Figure 1).

Regarding the term_name predicate, we keep a map μ stored under a ghost name γ_{term} that associates each term t to a name γ. We use an authoritative algebra to keep two copies of this map: an authoritative copy $\bullet\mu$, which records the exact state of the map, and a fragment $\circ\mu$, which can be split to track the name of each term. We define $\mathsf{term\_name}\ t\ \gamma$ as $\boxed{\circ[t \mapsto \mathsf{Agree}\ \gamma]}^{\gamma_{\mathsf{term}}}$, which guarantees that t corresponds to exactly one γ. The authoritative copy of the map is stored in an invariant that guarantees that every term in its domain is minted. In the proof of mk_nonce, when we allocate a fresh nonce t, but before t is minted, we can open this invariant to extend the term-name map with bindings for other terms $t' \succeq t$. Such terms are not minted, so we can prove that they are not in the map, which allow us to create fresh token resources for them.

10 Related Work

Verification of Message-Passing or Distributed Applications. Recent years have seen the introduction of several tools for reasoning about distributed systems and message-passing concurrency, such as Disel [Sergey et al. 2018], Actris [Hinrichsen, Bengtson, et al. 2020; Hinrichsen, Louwrink, et al. 2021], or Aneris [Gondelman et al. 2023; Krogh-Jespersen et al. 2020]. One common limitation of these tools is that they assume a non-adversarial communication model. For example, Actris assumes that messages cannot be dropped, duplicated or tampered with, whereas Aneris assumes that messages cannot be tampered with. By contrast, Cryptis allows us to reason about programs running over an adversarial network. On the other hand, some of these tools have been designed to reason about more challenging idioms of message-passing programming than what we currently handle. For example, Actris uses session types to reason about the communication between agents. In future work, we would like to bring together these two lines of research, by extending Cryptis to

integrate the reasoning principles identified by these and other tools for reasoning about distributed systems (e.g., integrate session types with our communication abstraction).

Some works in this space intersect with cryptography. For example, Hawblitzel et al. [2014] propose a methodology to verify applications running over an adversarial network. The specifications guarantee that the responses that a client receives from the network are the same regardless of whether what is sitting on the other end is the real application or a functionally equivalent abstract machine that serves as its specification. However, unlike Cryptis, the authors do not show that the responses received by the client satisfy any meaningful integrity guarantees. Drawing an analogy in the context of our key-value store case study, this would mean that the client would not have any guarantees that the value loaded from the database is indeed the last one that was stored.

Tools for Symbolic Cryptography Verification. There is a vast literature on techniques for verifying protocols; see Barbosa et al. [2021] for a comprehensive survey. One line of work in this landscape focuses on verifying the absence of memory-safety violations or other low-level bugs in implementations [Erbsen et al. 2019; Polubelova et al. 2020]. Ruling out such bugs is crucial for security, but does not suffice to establish all required integrity and confidentiality guarantees, which are usually analyzed with specialized tools. These tools strike a balance between many requirements, such as expressiveness, convenience, and scalability. In one corner of the design space, we have automated solvers such as ProVerif [Blanchet 2001], Tamarin [Meier et al. 2013] and CPSA [Doghmi et al. 2007], which favor convenience over expressiveness and scalability, but which are nonetheless powerful enough to analyze several real-world protocols. Cryptis explores a different set of trade-offs: limited support for push-button automation in return for more scalability and expressiveness, which enables the reuse of protocol proofs within larger systems.

Many other tools settle for similar trade-offs. The work that is the closest to ours is DY* [Bhargavan, Bichhawat, Do, et al. 2021b]. DY* is a state-of-the-art F* library for protocol verification that has been used to verify various protocols, such as Signal or ACME [Bhargavan, Bichhawat, Do, et al. 2021a]. There are several similarities between the two tools. Like Cryptis, DY* is based on symbolic cryptography and emphasizes expressiveness, allowing users to state and verify complex properties. DY* also maintains a predicate on the set of messages that travel through the network, akin to our public predicate: we can only send a message that satisfies the predicate, and every message received from the network is guaranteed to satisfy the predicate. Both tools are designed to reason about weak, or syntactic secrecy, and do not currently support relational indistinguishability properties. In DY*, secrecy is formulated using a system of confidentiality labels, akin to those used in information-flow control systems [Denning 1976]: each message tracks which principals or sessions are allowed to read it. Superficially, the public predicate of Cryptis is coarser, in that we only distinguish public and secret information. Nevertheless, the two models seem to be able to express similar policies. For example, Theorem 4.1 says that the NSL nonces can be read by the attacker if and only if $\triangleright(\mathsf{public}\ sk_I \vee \mathsf{public}\ sk_R)$, which we can interpret as saying that either the initiator or the responder are compromised. The DY* model of NSL offers similar guarantees.

Nevertheless, there are several differences between the two tools. Verification in DY* is carried out semi-automatically, by leveraging the F* type system and SMT solvers. DY* was designed to enable the extraction of executable code, a feature that would be crucial for making protocol implementations more reliable, but that Cryptis currently lacks. Regarding our focus, the reuse of protocol proofs to verify larger systems, DY* is not based on separation logic, so it does not support the several verification idioms that rely on it. Moreover, DY* has a restricted state model: agents can keep serialized long-term state associated with individual sessions, but do not have access to common primitives for manipulating heap data structures. Finally, DY* is sequential, whereas Cryptis has a nondeterministic scheduler. Since cryptographic protocols are concurrent systems,

they must be modeled in DY* by following a rigid coding discipline: we decompose each protocol into a series of actions, where each action is a separate function that does not rely on scheduling nondeterminism. If several actions are inadvertently combined into one function, its specification might hold only for a restricted choice of interleavings, which might miss some attacks.

These differences suggest that it would be difficult replicate in DY* the same type of proof reuse that Cryptis supports. Consider a system such as our key-value store. In DY*, it would be impossible to formulate succinct, self-contained specifications for the client wrappers, such as those of Figure 5. On the one hand, we would need to decompose each wrapper into several atomic actions to factor out local and network-wide scheduling nondeterminism. On the other hand, our specifications make crucial use of connectives that have no analogue outside of separation logic (e.g., a points-to connective to model the state of the store).

Another difference between the two tools, orthogonal to the goal of end-to-end verification, lies in the support for compositionality. DY* enables compositionality through a *layered approach* [Bhargavan, Bichhawat, Hosseyni, et al. 2023]: a protocol can be defined as a composition of several layers, where each layer specifies disjointness conditions that should be respected by other components, as well as predicates that need to be proved by its clients when using a cryptographic primitive. For example, if a component C uses an encryption key that is shared with other components, we must specify all encrypted messages that C is allowed to manipulate, and the other components cannot manipulate such messages in ways that conflict with what C expects. The message predicates of Cryptis play a similar role, but sacrifice some generality in return for ease of use: protocols can be composed automatically if they rely on disjoint message tags, a phenomenon that has been observed several times in the literature [Andova et al. 2008; Arapinis et al. 2015, 2012; Bugliesi et al. 2004a,b; Ciobâcă and Cortier 2010; Maffei 2005]. Tag disjointness only needs to be checked once, when declaring message predicates; by contrast, disjointness conditions in DY* need to be checked on every call to a cryptographic primitive.

On a parallel line of work, several authors have proposed ways of integrating symbolic cryptography within automated program analyzers for separation logic [Arquint et al. 2023; Vanspauwen and Jacobs 2015]. These proposals aim to verify that protocol implementations are free of memory safety violations while also conforming to their expected confidentiality and integrity guarantees. The work of Arquint et al. [2023] was the first to demonstrate that separation-logic resources are useful to reason about protocol security beyond just memory-related bugs, by using special freshness resources to prove that protocols satisfy *injective agreement* (the absence of replay attacks). Our metadata tokens enables similar, but more general, reasoning patterns. In particular, they can be used to prevent replay attacks at the application level, by guaranteeing that reliable connections deliver each message only once. Besides relying on a larger trusted computing base, one important difference with respect to Cryptis is that these works do not attempt to reuse proofs of protocol correctness to reason about larger systems. It is not obvious how these proposals could be leveraged to support this kind of reasoning. Our reliable connection abstraction, for instance, uses the term metadata feature of Cryptis to enable the transfer of resources through an authenticated connection, a feature that plays a crucial role in the verification of our key-value store.

Looking beyond symbolic cryptography, several works have been developed to reason about protocols in the computational model [Abate et al. 2021; Barthe, Dupressoir, et al. 2013; Gancher et al. 2023; Stoughton et al. 2022]. The computational model is more realistic than Cryptis' symbolic model, since it assumes that attackers have the power to manipulate messages as raw bitstrings, without being confined to a limited API of operations. On the other hand, dealing with such attackers requires more detailed reasoning, which means that such tools have difficulty scaling beyond individual cryptographic primitives or simple protocols. Looking beyond protocol verification, some works have proposed to reason about the use of cryptographic components in other types of

systems. For example, FastVer2 leverages a trusted execution environment (TEE) to allow clients to verify the correctness of key-value store operations. The authors prove that, if the FastVer2 monitor succeeds, then all the key-value store logs are sequentially consistent, unless a hash collision occurred. While Cryptis has similar goals, our focus is showing how proofs of cryptographic communication protocols can be reused to achieve application-level goals.

Specification of Authentication. Most verification works view a protocol as a means for agents to agree on their identities, parameters, session keys, or the order of events during execution [Arquint et al. 2023; Bhargavan, Bichhawat, Hosseyni, et al. 2023; Blanchet 2001; Datta et al. 2011; Gordon and Jeffrey 2003; Lowe 1997; Meier et al. 2013]. For example, if an initiator I authenticates with a responder R, we might want to guarantee that R was indeed running at some point in the past, that it was running and accepted to connect with I specifically, or that it accepted to start a unique session with I that corresponds to the session key that they exchanged [Lowe 1997]. Cryptis shows that agreeing on identities and on the contents of messages is crucial when reusing a protocol. For example, when a key-value store receives a database operation, it must know which agent sent this request to apply the operation to the correct database; when the client receives the response, it must track which key was queried to know which value will be returned. This aspect of authentication is implicit in Cryptis specifications, which allow us to determine the identity of participants based on the exchanged session key. On the other hand, we have not found an instance where the exact ordering of events in an authentication handshake could be leveraged to reason about a larger system that uses a protocol. This allowed us to define the Cryptis logic without the event traces that are used in related tools [Arquint et al. 2023; Bhargavan, Bichhawat, Do, et al. 2021b].

11 Conclusion and Future Work

We presented Cryptis, an Iris extension for symbolic cryptographic reasoning. As we demonstrated throughout the paper, Cryptis allows us to reduce the correctness of distributed systems verified in separation logic to elementary assumptions embodied by the symbolic model of cryptography, without the need for baking in a stronger (and less realistic) communication model. The integration of cryptographic reasoning allows us to evaluate how the correctness of a system is affected by compromising cryptographic material such as a long-term private key, going beyond what standard specifications in separation logic provide. Thanks to the adequacy of the Iris logic, which Cryptis inherits, these correctness results can be understood in rather concrete terms, via security games that rely only on the operational semantics of the underlying programming language.

Like other tools [Bhargavan, Bichhawat, Do, et al. 2021b], Cryptis is limited to single executions. This can be restrictive for security, since many specifications talk about pairs of executions (e.g. indistinguishability). We plan to lift this restriction drawing inspiration from prior work on reasoning about sealing via logical relations [Sumii and Pierce 2007, 2003] and relational reasoning in Iris [Frumin et al. 2018]. Another avenue for strengthening the logic would be to incorporate probabilistic properties and the computational model of cryptography. Prior work shows that probabilistic reasoning can benefit from separation logic [Barthe, Hsu, et al. 2020], and we believe that these developments could be naturally incorporated to our setting. Finally, we plan to extend the tool to encompass more protocols by adding more cryptographic primitives (e.g. group inverses would allow us to analyze the recent OPAQUE protocol [Jarecki et al. 2018]).

Acknowledgments

The authors would like to thank the anonymous reviewers for their thoughtful comments. This material is based upon work supported by the National Science Foundation under Grant No. 2314323 and 2314324.

References

Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 2021. "SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq." In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–15. DOI: 10.1109/CSF51468.2021.00048.

Suzana Andova, Cas J. F. Cremers, Kristian Gjøsteen, Sjouke Mauw, Stig Fr. Mjølsnes, and Sasa Radomirovic. 2008. "A framework for compositional verification of security protocols." *Inf. Comput.*, 206, 2-4, 425–459. DOI: 10.1016/j.ic.2007.07.002.

Myrto Arapinis, Vincent Cheval, and Stéphanie Delaune. 2015. "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* (Lecture Notes in Computer Science). Ed. by Riccardo Focardi and Andrew C. Myers. Vol. 9036. Springer, 324–343. DOI: 10.1007/978-3-662-46666-7_17.

Myrto Arapinis, Vincent Cheval, and Stéphanie Delaune. 2012. "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, 95–109. DOI: 10.1109/CSF.2012.16.

Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. 2023. "A Generic Methodology for the Modular Verification of Security Protocol Implementations." In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. ACM, 1377–1391. DOI: 10.1145/3576915.3623105.

[SW] Arthur Azevedo de Amorim, Amal Ahmed, and Marco Gaboardi, *Cryptis: Cryptographic Reasoning in Separation Logic* Nov. 2025. DOI: 10.5281/zenodo.17342914, URL: https://doi.org/10.5281/zenodo.17342914.

Michael Backes, Catalin Hritcu, and Matteo Maffei. 2011. "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* (Lecture Notes in Computer Science). Ed. by Sebastian Mödersheim and Catuscia Palamidessi. Vol. 6993. Springer, 1–28. DOI: 10.1007/978-3-642-27375-9_1.

Michael Backes, Catalin Hritcu, and Matteo Maffei. 2014. "Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations." *J. Comput. Secur.*, 22, 2, 301–353. DOI: 10.3233/JCS-130493.

Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. 2022. "A separation logic for negative dependence." *Proc. ACM Program. Lang.*, 6, POPL, 1–29. DOI: 10.1145/3498719.

Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. "SoK: Computer-Aided Cryptography." In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 777–795. DOI: 10.1109/SP40001.2021.00008.

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. "EasyCrypt: A Tutorial." In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures* (Lecture Notes in Computer Science). Ed. by Alessandro Aldini, Javier López, and Fabio Martinelli. Vol. 8604. Springer, 146–166. DOI: 10.1007/978-3-319-10082-1_6.

Gilles Barthe, Justin Hsu, and Kevin Liao. 2020. "A probabilistic separation logic." *Proc. ACM Program. Lang.*, 4, POPL, 55:1–55:30. DOI: 10.1145/3371123.

Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021a. "An In-Depth Symbolic Security Analysis of the ACME Standard." In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2601–2617. DOI: 10.1145/3460120.3484588.

Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021b. "DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code." In: *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 523–542. DOI: 10.1109/EUROSP51992.2021.00042.

Karthikeyan Bhargavan, Abhishek Bichhawat, Pedram Hosseyni, Ralf Küsters, Klaas Pruiksma, Guido Schmitz, Clara Waldmann, and Tim Würtele. 2023. "Layered Symbolic Security Analysis in DY★." In: *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part III* (Lecture Notes in Computer Science). Ed. by Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis. Vol. 14346. Springer, 3–21. DOI: 10.1007/978-3-031-51479-1_1.

Bruno Blanchet. 2001. "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, 82–96. DOI: 10.1109/CSFW.2001.930138.

Bruno Blanchet. 2002. "From Secrecy to Authenticity in Security Protocols." In: *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings* (Lecture Notes in Computer Science). Ed. by Manuel V. Hermenegildo and Germán Puebla. Vol. 2477. Springer, 342–359. DOI: 10.1007/3-540-45789-5_25.

Florian Böhl and Dominique Unruh. 2016. "Symbolic universal composability." *J. Comput. Secur.*, 24, 1, 1–38. DOI: 10.3233/JCS-140523.

Stephen Brookes. 2007. "A semantics for concurrent separation logic." *Theor. Comput. Sci.*, 375, 1-3, 227–270. DOI: 10.1016/j.tcs.2006.12.034.

Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. 2004a. "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, Michael Backes, David A. Basin, and Michael Waidner. ACM, 1–12. DOI: 10.1145/1029133.1029135.

Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. 2004b. "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* (Lecture Notes in Computer Science). Ed. by David A. Schmidt. Vol. 2986. Springer, 140–154. DOI: 10.1007/978-3-540-24725-8_11.

Michael Burrows, Martín Abadi, and Roger M. Needham. 1990. "A Logic of Authentication." *ACM Trans. Comput. Syst.*, 8, 1, 18–36. DOI: 10.1145/77648.77649.

Ştefan Ciobâcă and Véronique Cortier. 2010. "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, 322–336. DOI: 10.1109/CSF.2010.29.

Cyril Cohen. 2013. "Pragmatic Quotient Types in Coq." In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings* (Lecture Notes in Computer Science). Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Springer, 213–228. DOI: 10.1007/978-3-642-39634-2_17.

Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. 2016. "On Post-compromise Security." In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 164–178. DOI: 10.1109/CSF.2016.19.

Anupam Datta, John C. Mitchell, Arnab Roy, and Stephan Hyeonjun Stiller. 2011. "Protocol Composition Logic." In: *Formal Models and Techniques for Analyzing Security Protocols*. Cryptology and Information Security Series. Vol. 5. Ed. by Véronique Cortier and Steve Kremer. IOS Press, 182–221. DOI: 10.3233/978-1-60750-714-7-182.

Dorothy E. Denning. 1976. "A Lattice Model of Secure Information Flow." *Commun. ACM*, 19, 5, 236–243. DOI: 10.1145/360051.360056.

Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. 2007. "Searching for Shapes in Cryptographic Protocols." In: *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings* (Lecture Notes in Computer Science). Ed. by Orna Grumberg and Michael Huth. Vol. 4424. Springer, 523–537. ISBN: 978-3-540-71208-4. DOI: 10.1007/978-3-540-71209-1_41.

Danny Dolev and Andrew Chi-Chih Yao. 1983. "On the security of public key protocols." *IEEE Trans. Inf. Theory*, 29, 2, 198–207. DOI: 10.1109/TIT.1983.1056650.

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. "Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises." In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1202–1219. DOI: 10.1109/SP.2019.00005.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. "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, 442–451. DOI: 10.1145/3209108.3209174.

Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. 2023. "Owl: Compositional Verification of Security Protocols via an Information-Flow Type System." In: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1130–1147. DOI: 10.1109/SP46215.2023.10179477.

Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. "Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols." *Proc. ACM Program. Lang.*, 7, ICFP, 847–877. DOI: 10.1145/3607859.

Andrew D. Gordon and Alan Jeffrey. 2003. "Authenticity by Typing for Security Protocols." *Journal of Computer Security*, 11, 4, 451–520. DOI: 10.3233/JCS-2003-11402.

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. "Ironclad Apps: End-to-End Security via Automated Full-System Verification." In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. Ed. by Jason Flinn and Hank Levy. USENIX Association, 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel.

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. "Actris: session-type based reasoning in separation logic." *Proc. ACM Program. Lang.*, 4, POPL, 6:1–6:30. DOI: 10.1145/3371074.

Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. "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, 178–198. DOI: 10.1145/3437992.3439914.

Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. 2018. "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks." In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III* (Lecture Notes in Computer Science). Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. Springer, 456–486. DOI: 10.1007/978-3-319-78372-7_15.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. "Higher-order ghost state." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 256–269. DOI: 10.1145/2951913.2951943.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic." *J. Funct. Program.*, 28, e20. DOI: 10.1017/S0956796818000151.

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris." In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain* (LIPIcs). Ed. by Peter Müller. Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPICS.ECOOP.2017.17.

Hugo Krawczyk. 2003. "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols." In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings* (Lecture Notes in Computer Science). Ed. by Dan Boneh. Vol. 2729. Springer, 400–425. DOI: 10.1007/978-3-540-45146-4_24.

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. "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* (Lecture Notes in Computer Science). Ed. by Hongseok Yang. Vol. 10201. Springer, 696–723. DOI: 10.1007/978-3-662-54434-1_26.

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. "Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems." In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings* (Lecture Notes in Computer Science). Ed. by Peter Müller. Vol. 12075. Springer, 336–365. DOI: 10.1007/978-3-030-44914-8_13.

John M. Li, Amal Ahmed, and Steven Holtzen. 2023. "Lilac: A Modal Separation Logic for Conditional Probability." *Proc. ACM Program. Lang.*, 7, PLDI, 148–171. DOI: 10.1145/3591226.

Gavin Lowe. 1997. "A Hierarchy of Authentication Specification." In: *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*. IEEE Computer Society, 31–44. DOI: 10.1109/CSFW.1997.596782.

Gavin Lowe. 1996. "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* (Lecture Notes in Computer Science). Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 1055. Springer, 147–166. DOI: 10.1007/3-540-61042-1_43.

Matteo Maffei. 2005. "Tags for Multi-Protocol Authentication." *Electron. Notes Theor. Comput. Sci.*, 128, 5, 55–63. DOI: 10.1016/j.entcs.2004.11.042.

Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. "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* (Lecture Notes in Computer Science). Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Springer, 696–701. DOI: 10.1007/978-3-642-39799-8_48.

Roger M. Needham and Michael D. Schroeder. 1978. "Using Encryption for Authentication in Large Networks of Computers." *Commun. ACM*, 21, 12, 993–999. DOI: 10.1145/359657.359659.

Peter W. O'Hearn. 2007. "Resources, concurrency, and local reasoning." *Theor. Comput. Sci.*, 375, 1-3, 271–307. DOI: 10.1016/j.tcs.2006.12.035.

Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella Béguelin. 2020. "HACLxN: Verified Generic SIMD Crypto (for all your favourite platforms)." In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM, 899–918. DOI: 10.1145/3372297.3423352.

John C. Reynolds. 2002. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. DOI: 10.1109/LICS.2002.1029817.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. "Programming and proving with distributed protocols." *Proc. ACM Program. Lang.*, 2, POPL, 28:1–28:30. DOI: 10.1145/3158116.

Alley Stoughton, Carol Chen, Marco Gaboardi, and Weihao Qu. 2022. "Formalizing Algorithmic Bounds in the Query Model in EasyCrypt." In: *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel* (LIPIcs). Ed. by June Andronick and Leonardo de Moura. Vol. 237. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:21. DOI: 10.4230/LIPIcs.ITP.2022.30.

Eijiro Sumii and Benjamin C. Pierce. 2007. "A bisimulation for dynamic sealing." *Theor. Comput. Sci.*, 375, 1-3, 169–192. DOI: 10.1016/j.tcs.2006.12.032.

Eijiro Sumii and Benjamin C. Pierce. 2003. "Logical Relations for Encryption." *J. Comput. Secur.*, 11, 4, 521–554. DOI: 10.3233/JCS-2003-11403.

[SW] The Rocq Development Team, *The Rocq Prover* version 9.0, Apr. 2025. DOI: 10.5281/zenodo.15149629, URL: https://doi.org/10.5281/zenodo.15149629.

Amin Timany, Armaël Guéneau, and Lars Birkedal. 2024. "The Logical Essence of Well-Bracketed Control Flow." *Proc. ACM Program. Lang.*, 8, POPL, 575–603. DOI: 10.1145/3632862.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. "GPS: navigating weak memory with ghosts, protocols, and separation." In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black and Todd D. Millstein. ACM, 691–707. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660243.

Gijs Vanspauwen and Bart Jacobs. 2015. "Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications." In: *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings* (Lecture Notes in Computer Science). Ed. by Radu Calinescu and Bernhard Rumpe. Vol. 9276. Springer, 53–68. ISBN: 978-3-319-22968-3. DOI: 10.1007/978-3-319-22969-0_4.

Received 2025-07-10; accepted 2025-11-06