DY* Unchained: Now with Composable Security Proofs and Precise Compromise Scenarios

Théophile Wallez
CISPA Helmholtz Center for Information Security

***Abstract*—Cryptographic protocols are the cornerstone of Internet security, and any flaw in their design would have drastic effects. We can formally prove the absence of such flaws using a variety of automated or semi-automated tools. However, some features of real-world protocols are notoriously hard to analyze using these tools, including unbounded loops, unbounded data structures, and unbounded and dynamic number of protocol participants. The DY* protocol verification framework recently emerged as a tool designed to address these challenges, and it was successfully used to analyze protocols such as Signal, ACME and TreeSync.**

However, we note that DY* suffers from two deep limitations: first, security proofs of protocol subcomponents cannot be composed, which hinders the analysis of large protocols; second, the security proofs depend on a simple language to describe compromises, which overly restricts the set of compromise scenarios DY* can reason about.

In this paper, we present a major overhaul of DY* that addresses these limitations. We enable composing security proofs in DY* by developing a framework to define trace invariants modularly, and we improve the precision of compromise scenarios that DY* can prove by fully generalizing the notion of security labels. These improvements are essential to enable the analysis of large protocols. In particular, our new version of DY* was already used by and crucial to the security proofs of the TreeKEM protocol (IEEE S&P 2025).

1. Introduction

Communications over the Internet crucially rely on cryptographic protocols to protect sensitive data: Web browsers rely on Transport Layer Security (TLS) [39] to secure connections to Web servers, and messaging apps use the Signal Protocol [30] or Messaging Layer Security (MLS) [8] to secure the content of discussions.

These protocols are often composed of several building blocks. For example, TLS relies on a Public-Key Infrastructure (PKI), which provides authentication for the Handshake Protocol, which establishes a shared key used by the Record Protocol, which finally secures the transmitted data. The Signal Protocol uses the X3DH [31] (or PQXDH [28]) protocol, which feeds an initial key into the Double Ratchet protocol [37], and, as the name suggests, the Double Ratchet protocol is itself composed of two ratchet mechanisms; furthermore, the Signal Protocol is only one component of the Signal app, which comprises various other components such as Sealed Sender [29], the Private Group System [17], and Sesame [32]. As a last example, MLS [8] is the composition of three subcomponents [42], TreeSync, TreeKEM, and TreeDEM, with TreeKEM itself comprising multiple subcomponents [44], such as its tree, its keyschedule, and its welcoming procedure.

Such cryptographic protocols form the cornerstone of Internet security, and any flaw in their design would have dramatic effects. We can guarantee the absence of such flaws by performing *formal analysis* using tools developed by the research community.

Mechanized Cryptographic Protocol Analysis. A broad range of tools can mechanically prove the absence of some type of flaws in cryptographic protocols. Each of these tools has different trade-offs (see [7] for a detailed survey), but they broadly fall into two categories.

The first category consists of tools that prove probabilistic security guarantees of protocols using computational assumptions of their underlying primitives; these tools work in the *computational model*. Although these tools prove strong security guarantees, creating such computational proofs requires significant manual labor and are difficult to maintain, especially as the cryptographic protocol being analyzed grows in size. This makes it difficult to analyze every feature and component of large cryptographic protocols, which in turn may lead the analysis to miss attacks that exploit subtle interactions between these features.

To perform a more comprehensive analysis that covers many features of a cryptographic protocol, we can turn to tools of the second category, which work in the *symbolic model* and prove the absence of *logical attacks*. Although these tools prove weaker security properties, they benefit from better automation and scale better to large protocols; as a result, they are complementary to computational tools. Leading examples of such tools are ProVerif [14] and Tamarin [33], which have proved to be effective in performing comprehensive analysis of complex protocols, such as TLS 1.3 during its standardization [13], [21]. The symbolic model is the focus of this paper.

Composed Protocol Analysis. We mentioned earlier how cryptographic protocols are often composed of several subprotocols. Unfortunately, ProVerif and Tamarin do not benefit from this composite nature, as they can only perform whole-protocol analysis. This results in computationally-

intensive analysis: for example, the analysis of TLS 1.3 in Tamarin requires 100GB of memory and more than a day of computations [21].

Modular verification using DY*. To work around this issue, another line of research relies on dependent types to allow for a modular verification of cryptographic protocols, starting with RCF [10]. The leading example of this line of research is DY* [11], which is based on the program verifier F* [40] and has been used to successfully analyze the Signal Protocol [11] and the TreeSync sub-protocol of MLS [42]. However, despite this progress made toward the analysis of large protocols, we note that DY* still suffers from three deep limitations that prevent reaching this goal.

Limitation 1: Composing Security Proofs. DY* promises *modular verification* of cryptographic protocols, in the sense that DY* verifies each protocol step independently. However, it does not allow for *composing security proofs*, in the sense that if DY* has independently proved the security of two protocols, this does not mean that DY* can easily prove the security of their composition. To understand why, we now give a high-level explanation on how DY* proves security properties. DY* aims to prove security theorems of the shape "for all reachable traces, this security property holds". To do so, DY* requires users to provide a single *trace invariant*, then (1) prove that the trace invariant implies the security properties, and (2) prove that every reachable trace satisfy the trace invariant by proving that the trace invariant is preserved by each protocol step (we will dive more deeply into this methodology in §2). As a result, the proof is modular, because each protocol step is proved to preserve the trace invariant independently of other steps. However, it does not allow for composing proofs: this approach works because we use a single trace invariant for our proof, whereas the proofs of two distinct protocols likely rely on different trace invariants.

In an attempt to solve the problem of composing security proofs in DY*, Bhargavan et al. [12] proposed an approach for protocols that can be described in *layers*: for example, HTTPS depends on the TLS Record Protocol, which depends on the TLS Handshake Protocol, which depends on the PKI, where the guarantees of each protocol are proved using the guarantees provided by the protocols below it. Their approach is to write a trace invariant for the lowest layer (e.g. PKI), which is parametrized by invariants for higher layers (e.g. Handshake Protocol), so that the whole proof depends on a single trace invariant (e.g. the PKI trace invariant instantiated with the Handshake Protocol invariants). However, we note that this approach works only when the dependency graph between subcomponents can be ordered linearly in a vertical fashion. In particular, it does not handle when a protocol depends on two other protocols, and it does not handle *horizontal composition*, where several protocols (which may share secret keys) are run in parallel. Ultimately, this limitation makes it infeasible for DY* to reason on large protocols that combine many subcomponents, such as MLS.

Limitation 2: Compromise Scenarios. We further note that DY* suffers from a crucial limitation in its ability to reason with compromises when composing several sub-protocols. For example, to express the security properties of a key exchange sub-protocol (e.g. the TLS Handshake Protocol), DY* can prove that the exchanged key is secure unless some specific compromises have happened, and the compromise scenarios can be expressed using the full expressivity of F*. However, when this key is later used in another sub-protocol (e.g. the TLS Record Protocol), DY* requires encoding the compromise scenarios in a *security label* (described more thoroughly in §2 and §4), which is an internal tool in DY* to reason about the compromises that are required for the attacker to know a given bytestring. We therefore see that DY* offers two different languages to describe compromise scenarios: F* (in final security theorems) and security labels (in internal lemmas). As the language for security labels is less expressive than the language of F*, when DY* users need to reason on the security label of an exchanged key in a sub-protocol that uses this key, they may be forced to encode an over-approximation of the compromise scenarios in the security label. Ultimately, this means that the compromise scenarios in the security theorem of composed protocols will be less precise than they could be. This limitation makes it infeasible for DY* to express precise compromise scenarios for complex protocols such as MLS.

Limitation 3: Specification Auditability. When we analyze a cryptographic protocol using a tool that is proven sound, we defer the checking of the proof to a computer; humans only need to audit the specification of the cryptographic protocol and the security theorems. In DY*, the security proofs are performed at the same time as the protocol is specified, in an interleaved fashion, which hinders the readability of specifications in DY*. Although the authors of the original DY* paper [11] advocate for auditability through *executable specifications*, which allows for interoperability testing, this approach should be complementary to the auditing of specifications by humans instead of replacing it.

Contributions. In this paper, we present a new version of DY* that addresses the three aforementioned limitations. Our improvements are essential to enable the analysis of large protocols: in particular, our new version was already used by and crucial to the security proofs of the TreeKEM protocol [44]. Our two main contributions are the following. First, we present a generic approach toward composable security proofs in DY*, by developing a framework that allows users to define a single global trace invariant modularly from multiple local invariants, such that the invariant preservation proof of a protocol step works with any global trace invariant, as long as it is composed of some local invariants related to this step. Our framework supports a broad range of composition techniques, including vertical composition (where a component relies on the guarantees provided by another component) and different flavors of horizontal composition (when two components are safe to run in parallel, for example when they use distinct keys, or

when they reuse the same key but with proper domain separation). Second, we improve the precision of compromise scenarios that DY* can prove by allowing security labels to use the full expressivity of F*, thereby merging the two languages DY* uses to express compromise scenarios. As a third (and more minor) contribution, we also improve the auditability of specifications in DY* by allowing users to do proofs separately from specifications, while still allowing the execution of specifications for interoperability testing.

The code of our new version of DY* is open source: it is publicly developed on GitHub [1] and is also documented and available as supplementary material [2].

Overview. We first give a gentle introduction to the DY* proof methodology (§2). We then explain how we compose security proofs (§3), how we generalize the security label language (§4) and how we improve the auditability of DY* specifications (§5). We finally discuss related work (§6) and conclude (§7).

2. Background on DY*

In this section we give the background on DY* that is necessary to fully understand our contributions. We first explain how protocols are specified (§2.1), then how to write security properties (§2.2). We finally dive into the specificities of security proofs in DY* (§2.3).

2.1. Specifying Cryptographic Protocols in DY*

In order to prove that a cryptographic protocol achieves given security properties, one needs to write a *specification* of this protocol. We now describe how this is done in DY* and how DY* models the attacker.

Protocol participants. In a cryptographic protocol, one can model participants as state machines. Such participants can perform pure computations, whose results do not depend on what happens in the outside world and conversely have no effect on the outside world. Examples for pure computations are serializing messages or computing cryptographic functions. Participants can also perform impure actions, which may depend on or be affected by the outside world, such as sending messages on the network, generating fresh randomness, or transitioning state. DY* models these impure actions by logging them in a *trace*, which is a global, shared, append-only log of every impure action that happened during a protocol execution amongst all participants (including the attacker). In DY*, users specify protocol participants by implementing their state machine via a set of effectful functions that perform the transition from one state to another. The effect of these functions is handled by appending new entries and reading past entries of the trace. For example, protocol participants can send messages on the network or store their state by appending corresponding entries in the trace; conversely, they can read messages from the network or retrieve their state by obtaining them from the trace.

Compromise. In the real world, secret state stored by a participant may leak to the attacker (e.g. when a phone is seized by the police or a laptop is hacked using a zero-day exploit). To capture this, the attacker may compromise any state of protocol participants at any time. Therefore, all security properties will be expressed "modulo compromise". For example, DY* will not say that the attacker never knows a secret value; instead DY* will say this value is secret from the attacker unless the attacker has performed some specific compromises. When the attacker compromises the state of a participant, DY* models this as extending the trace with an entry corresponding to this compromise.

The DY* attacker. In DY*, the attacker is the standard Dolev-Yao (or symbolic) attacker [22], who controls the network (hence can drop, delay, replay, and modify messages on the fly), and can compute any cryptographic function on inputs they know. The attacker is modeled as an F* function that can compute any cryptographic function and can call the various participants' effectful functions defined by the protocol specification. Any symbolic attack can be represented by such an F* program. A trace T is defined to be reachable by the attacker when there exists an F* function (representing the attacker) that, when starting with the empty trace, results in the trace T.

2.2. Writing security properties in DY*

The analysis of cryptographic protocols seeks to establish security properties. We now explain how these are described in DY*.

Reachability properties. DY* seeks to establish *reachability properties*, that is, security statements of the form "for all traces reachable by the attacker, this property holds on the trace". To express properties on reachable traces, DY* allows the use of the full expressivity of F*: in addition to all the logic operators (such as conjunction, disjunction or implication), DY* allows for expressing the presence of some entries in the trace, which may represent facts such as "Alice has finished the handshake with Bob and established the key k", "the attacker has compromised Charlie". Furthermore, DY* also allows for expressing the fact that "the attacker knows the bytestring b".

Attacker knowledge. Given a trace T, DY* defines the set of bytestrings known by the attacker (with respect to trace T) as the smallest set of bytestrings that

- contains bytestrings that were sent on the network (i.e. there exist a corresponding entry in T),
- contains bytestrings that correspond to compromised states (i.e. there exist corresponding entries in T),
- is stable by computing cryptographic functions.

Example properties. DY* can express confidentiality properties, for example via a reachability property like "for all reachable traces, if Alice has finished the handshake with Bob and established the key k, and the attacker knows the key k, then it must be that the attacker has compromised either Alice or Bob". DY* can also express authenticity

properties, for example via a reachability property like "for all reachable traces, if Alice has finished the handshake with Bob and established the key k, then it must be that either Bob has finished the handshake with Alice and established the same key k, or the attacker has compromised Bob's identity keys".

2.3. Proving trace properties with DY*

Given a protocol specification and its desired security properties (expressed as reachability properties), DY* provides tools to prove that every reachable trace obeys these security properties.

Trace invariant. DY* requires users to strengthen the desired security properties to obtain a *trace invariant* and then prove that every reachable trace satisfies this trace invariant. This is proved by requiring users to prove that every effectful protocol function preserves the trace invariant. As every function used by the attacker preserves the trace invariant, the attacker (which is modeled as an F* function) has no choice but to also preserve the trace invariant. This concludes that the attacker can only reach traces that satisfy the security properties. This approach allows users to conduct proofs *modularly*, because the proofs that each protocol step preserves the trace invariant are independent of each other. For better usability, DY* does not actually require users to provide a full-fledged trace invariant; instead, DY* provides a basic blueprint to create trace invariants and allows users to tweak parts of the trace invariant that are protocol-specific. This also allows DY* to prove once and for all that every action performed by the attacker (e.g. receiving messages or compromising states, computing cryptographic functions and sending the result on the network) must preserve the trace invariant.

Bytes invariant. A key ingredient of the trace invariant is the *bytes invariant*, which effectively restricts honest participants to a "safe" usage of cryptography regarding the desired security properties (recall that the invariants are strengthenings of these properties). For example, the bytes invariant imposes honest participants to only sign bytestrings that verify a property called the *signature predicate* (which is specific to each protocol and captures the meaning of the signature), so that when another participant successfully verifies the signature, they know that either the signature was computed honestly, hence the signed message must obey the signature predicate, or the signature was computed by the attacker, hence the attacker must know the corresponding private signature key. As another example, honest participants may only encrypt messages to keys that are "more secret" than the message, to ensure the attacker cannot use the ciphertext and the key they know to learn something they are not allowed to.

The bytes invariant is an invariant on all the bytes that appear in the protocol execution, whether computed by honest protocol participants or by the attacker. The bytes invariant relies on lower-level tools described below, such as security labels or key usages.

Security labels. Sharing ideas with Information Flow, security labels capture what it means for a bytestring to be "more secret" than another. Every bytestring is associated with a label that encodes an over-approximation of the compromise scenarios that may lead the attacker to know this bytestring. Two labels can be compared to determine whether one is more secret than the other. The order between labels depends on the compromise events in the trace: when an attacker compromises the state of a participant, the label corresponding to this state effectively becomes equivalent to the public label. Then the Attacker Knowledge Theorem (described below) will guarantee that any bytestring known by the attacker has a label equivalent to the public label, or equivalently that one of the compromise scenarios encoded by the label must have happened.

Key usage. Security proofs (whether symbolic or computational) often rely (sometimes implicitly) on the fact that some keys are distinct from each other. For example, in a proof of a security theorem on TLS, the proof implicitly assumes that the long-term keys of TLS are not reused in SSH: if TLS and SSH happened to share long-term keys, this would require a combined security proof to ensure that there is no cross-protocol attack. Another example is that computational security proofs rely on the fact that signature keys are not used to encrypt: doing so would prevent applying the standard EUF-CMA security assumption. In more generality, it is good hygiene to make sure a key is used with only one cryptographic primitive. In DY*, cryptographic keys are associated with a *usage*, and ensures that two keys with different purposes must have different usage, hence are distinct.

Attacker knowledge theorem. A key ingredient to prove confidentiality theorems is the *Attacker Knowledge Theorem*, which states that the attacker only knows bytestrings (1) that satisfy the bytes invariant and (2) whose label is equivalent to the public label. This allows to prove confidentiality properties as follows: if the attacker knows some secret key (of which the label is known, e.g. only Alice is supposed to know that key), then by the Attacker Knowledge Theorem, the label of that key is equivalent to the public label, which can only happen if some compromise scenario has happened (e.g. the attacker has compromised Alice).

Soundness. We note that the trace invariant is only a proof technique in DY* and that the exact definition of the trace invariant is untrusted: it suffices to exhibit any trace invariant that (1) is preserved by every protocol step (i.e. actions that can be performed by the attacker) and (2) implies the desired security properties. Both facts imply that every trace reachable by the attacker must obey the desired security properties. Furthermore, both facts are mechanically checked in F*. This guarantees the soundness of the DY* approach, both for the original DY* and for our new version presented in this paper.

3. Composing Security Proofs

The DY* proof methodology relies on the user to create a trace invariant and then prove two facts: that every protocol step preserves this trace invariant and that this trace invariant implies some security property. These two facts imply that the attacker cannot break the protocol security: the attacker can only reach traces that satisfy the trace invariant (first fact), which must obey the protocol security properties (second fact).

Defining the trace invariant. Although the trace invariant must be specific for each protocol security proof, DY* provides a generic trace invariant that is parametrized for parts that are protocol-specific. Therefore, DY* does not require users to write a full-fledged trace invariant for each protocol analysis. Instead, users only need to provide parts of the trace invariant, which we call *sub-invariants*. An example of such a sub-invariant is the *signature sub-invariant*, which is a predicate on signed messages: the generic DY* trace invariant will ensure that for any signature appearing in the protocol, either the signature is honest, and in that case the user-provided predicate for the signature sub-invariant holds on the signed message, or the signature was computed by the attacker, in which case the attacker knows the private key corresponding to this signature. Similarly, the generic trace invariant requires users to provide predicates to define the sub-invariants for Authenticated Encryption with Associated Data (AEAD), Message Authentication Code (MAC), and also for state storage or protocol event logging. User-provided sub-invariants are not always predicates: for example, the sub-invariants for Key Derivation Functions (KDF) include a function that computes the security label of the derived key.

Composing security proofs. With this approach, the protocol verification is modular because each protocol step is proved to preserve the trace invariant independently. However, in the process of writing the proof for a protocol step, it may happen that the DY* user needs to tweak (say) the signature sub-invariant. When doing so, the trace invariant will change and every protocol step needs to be re-checked with respect to this new trace invariant, even the protocol steps that do not deal with signatures.

Because of this limitation, DY* cannot compose protocol security proofs, for example, to prove the security of a protocol (e.g. TLS 1.3) by proving the security of (smaller) sub-protocols (e.g. TLS' Handshake Protocol and Record Protocol). Both protocol security proofs likely use different sub-invariants; however, DY* requires users to provide a single set of sub-invariants so that every protocol step is proved to preserve the same trace invariant. Hence, to combine both security proofs, the user would need to combine the two sets of sub-invariants in a single one and re-prove that each protocol steps preserves the new trace invariant.

This is a known limitation of DY*: in an attempt to address it, Bhargavan et al. [12] propose an approach for a restricted type of protocol composition in DY* and implement a generic "communication layer" to be used by protocols that rely on an authenticated or confidential communication channel. Their approach handles vertical composition (when a protocol relies on another protocol), but it only handles a linear dependency graph (e.g. it does not work when a protocol relies on two other protocols), and it does not handle horizontal composition (when two protocols that may share keys run in parallel).

Contributions. In this section, we present a framework to create sub-invariants, which allows for a broad range of protocol compositions, including vertical and horizontal composition, and is essential to enable the analysis of large protocols in DY*. In particular, our sub-invariant framework was already used by and crucial to the security proofs of the TreeKEM protocol [44].

Soundness. As previously explained in §2.3, the trace invariant is only a proof technique in DY* and the soundness of security proofs in DY* does not depend on the exact definition of the trace invariant. Because our framework is only a means to help DY* users to exhibit a trace invariant that is preserved by every protocol step and implies the desired security properties, our framework has no impact on the soundness of DY*; the soundness of our new version of DY* is still mechanically checked in F*.

Outline. We present our invariant framework in four steps:

- First, we discuss the general shape of sub-invariants our framework can create.
- Second, we show how to prove that a protocol step preserves the trace invariant by relying on a notion of *local sub-invariant*.
- Third, we show how to create a single global sub-invariant from several local sub-invariants.
- Finally, we demonstrate our sub-invariant framework on two case studies: TreeKEM and a model of HPKE.

3.1. Shaping DY* sub-invariants as decision trees

Secure cryptographic protocols may lose their security guarantees when composed with other secure cryptographic protocols, for example when they share long-term keys, which may lead to cross-protocol attacks.

For this reason, it is impossible to combine arbitrary sub-invariants, as this would be unsound: it would prove the security of the corresponding composition of protocols, although the composition of two secure protocols is not always secure. Instead, our framework enforces a specific shape of user-provided sub-invariants so as to allow for their composition. We now go through several criteria that intuitively describe under which conditions protocols can be safely composed. We then turn these intuitive criteria into a formal shape of user-provided sub-invariants that are composable. Finally, we discuss why this does not affect the expressive power of DY*.

Distinct keys. Two cryptographic protocols may be safely composed when they use distinct key material (for example, software developers routinely run TLS and SSH in parallel). When this is the case, it is possible to combine

```
let sign_pred vk msg =
  match get_usage vk with
  | SignKey "TLS" →
    tls_sign_pred vk msg
  | SignKey "SSH" →
    ssh_sign_pred vk msg
```

Figure 1: Example of a global sub-invariant (predicate for signed messages) that dispatches to local sub-invariants using key usage.

```
let tls_sign_pred vk msg =
  match get_domain_sep msg with
  | "TLS␣1.2" →
    tls12_sign_pred vk msg
  | "TLS␣1.3" →
    tls13_sign_pred vk msg
```

Figure 2: Example of a global sub-invariant (predicate for signed messages) that dispatches to local sub-invariants using a domain separator. Note that TLS does not have such a straightforward domain separator, this code only serves as an example.

the corresponding user-provided sub-invariants of the two protocols, by using a new sub-invariant of the form "if the key belongs to protocol one, then dispatch to the sub-invariant of protocol one, and if the key belongs to protocol two, then dispatch to the sub-invariant of protocol two", as shown in Figure 1. The notion of a key belonging to a protocol is formalized in DY* via the *key usage* (see §2.3).

Domain separation. Two cryptographic protocols may use the same key but use that key with good domain separation, for example by tagging the input of their cryptographic primitive. When this is the case, it is possible to combine the user-provided sub-invariants of the two protocols by using a DY* sub-invariant of the form "if the input tag corresponds to protocol one, then dispatch to the sub-invariant of protocol one, and if the input tag corresponds to protocol two, then dispatch to the sub-invariant of protocol two", as shown in Figure 2.

Shaping sub-invariants as decision trees. The two approaches we have seen can be combined: first, dispatch on the key usage, then, dispatch on the domain separator, and so on. Ultimately, the user-provided sub-invariants are shaped like a decision tree, where the branch to take at each step depends on a tag (e.g. the key usage or a domain separator).

Expressivity. We claim that imposing such shape on the user-provided sub-invariants does not affect the expressivity of DY*. Wallez et al. [43] give a sufficient set of criteria protocols should obey to avoid cross-protocol attacks, through the lens of message formats, in turn, protocols that obey these criteria can have their sub-invariants shaped as decision trees. For example, on the criterion for signatures, Wallez et al. say that "each signature key must be used to sign messages with the same self-contained, non-ambiguous, representation-unique message format". With the lens of DY*, for protocols that obey this criterion, the decision tree can first branch on the signature key usage (as in Figure 1), then for each usage corresponds a single message format, and use the corresponding parsing function to implement get_domain_sep (as in Figure 2). Conversely, should a protocol not obey this criterion, we would be unable to write the signature sub-invariant in DY* (regardless of our framework), which would hint at a cross-protocol attack.

Our sub-invariant framework. We do not deal with full-blown decision trees directly. Instead, our sub-invariant framework only deals with one node of the decision tree. It can then be applied recursively to obtain a complete decision tree. When we consider one node of the decision tree, we will say that the sub-invariant at this node is the *global sub-invariant*, and sub-invariants it dispatches to (i.e. branches of the decision tree) are *local sub-invariants*. This notion of global or local sub-invariants therefore depends on the context: for example, tls_sign_pred is the global sub-invariant in Figure 2, but is a local sub-invariant in Figure 1. In §3.2, we will see how to express and use the fact that a global sub-invariant "contains" a local sub-invariant, then in §3.3 we will see how to construct a global sub-invariant from a list of local sub-invariants, such that the global sub-invariant "contains" each local sub-invariant.

3.2. Relating local and global sub-invariants

When proving that a protocol step preserves the generic DY* trace invariant that is parametrized by user-provided sub-invariants, our key insight is that DY* users do not need to know the complete definition of all sub-invariants: they only need to know how some sub-invariants behave on a particular branch of a decision tree. For example, when proving that each protocol step of TLS 1.3 preserves the trace invariant, DY* users only need to know that the user-provided predicate for the signature sub-invariant sign_pred contains the branch tls_sign_pred (Figure 1) and that tls_sign_pred contains the branch tls13_sign_pred (Figure 2). In this section, we formally define what it means for a global sub-invariant to contain a local sub-invariant.

One node at a time. We only deal with one node of the decision tree, and apply our methodology recursively to obtain a complete decision tree. Hence, the notion of "global" and "local" sub-invariant is relative to the node of the decision we are currently considering: for example, in Figure 1, tls_sign_pred is a local sub-invariant, but in Figure 2 it is the global sub-invariant.

Setup provided by the user. Our framework relies on the user to provide some types and functions:

- $\mathbb{D}_G$, a type for global sub-invariant input data (in the example of signature sub-invariant, $\mathbb{D}_G$ would be a tuple containing verification key and signed message)
- $\mathbb{D}_L$ a type for local sub-invariant input data (in the simplest case it might be equal to $\mathbb{D}_G$)

- $\mathbb{T}$, a type for tags (in the example of Figure 1, $\mathbb{T}$ is the set of key usages; in the example of Figure 2, $\mathbb{T}$ is the set of domain separators, i.e. strings)
- $\mathbb{O}$, a type for the sub-invariant output (for predicates, $\mathbb{O}$ is the type of propositions)
- a function $\mathit{decode} : \mathbb{D}_G \to \mathbb{T} \times \mathbb{D}_L \cup \{\bot\}$ that computes a tag and a local input data given a global input data.

Relating local and global sub-invariants. With this setup, a local sub-invariant is a function $\mathbb{D}_L \to \mathbb{O}$, and a global sub-invariant is a function $\mathbb{D}_G \to \mathbb{O}$. Given a tag t, a local sub-invariant l, and a global sub-invariant g, we say that $(t, l) \subset g$ (or g "contains" l) when

$$\forall x_L x_G. \mathit{decode}(x_G) = (t, x_L) \implies g(x_G) = l(x_L)$$

In other words, when $(t, l) \subset g$, we know how g behaves on inputs with tag t. For example, in Figure 1, the property

$$(\texttt{"TLS"}, \mathsf{tls_sign_pred}) \subset \mathsf{sign_pred}$$

would be equivalent to

$$\forall \mathsf{vk}, \mathsf{msg}.\ \mathsf{get_usage\ vk} = \mathsf{SignKey}\texttt{"TLS"} \implies \mathsf{sign_pred\ vk\ msg} = \mathsf{tls_sign_pred\ vk\ msg}$$

This means that when we know that vk is a key that belongs to the TLS protocol, we deduce that sign_pred vk msg behaves like tls_sign_pred vk msg, which allows us to prove a protocol step of TLS with respect to the global signature sub-invariant sign_pred, without knowing its complete definition.

Reasoning on global sub-invariants using local sub-invariants. DY* users must deal with the various (global) sub-invariants when proving that a protocol step preserves the trace invariant. For example, they must prove that the (global) predicate for the signature sub-invariant holds when computing a signature, or deduce that the (global) predicate for the signature sub-invariant holds when a verification succeeds.

In the original DY* [11], the proof methodology was to define top-level global sub-invariants and to prove every protocol step with respect to these particular global sub-invariants. As a result, if the global sub-invariants were to change (e.g. by merging the sub-invariants of two protocols or tweaking parts of the sub-invariants), every proof showing that the protocol steps preserve the trace invariant would need to be re-checked and possibly be updated, even when they are not related to the change in question.

With our methodology, DY* users can now write proofs showing that protocol steps preserve the trace invariant with respect to *any* global sub-invariants, as long as they contain local sub-invariants specific to this protocol step. The proof is done once and for all, and it is unaffected by changing the global sub-invariants on unrelated local sub-invariants. Using the relationship $(t, l) \subset g$, DY* users can do proofs on the (abstract) global sub-invariant by relying on our knowledge about the (concrete) local sub-invariant, for every input that corresponds to the tag t.

```
let L = [
    ("TLS", tls_sign_pred);
    ("SSH", ssh_sign_pred);
]

// Need to prove the following:
assert (pairwise_distinct (map fst L))
// This is trivially true because
// map fst L == ["TLS"; "SSH"]
```

Figure 3: Example of local sub-invariants list for Figure 1.

3.3. Creating a global sub-invariant from local sub-invariants

With our methodology, each proof that a protocol step preserves the trace invariant now requires that the global sub-invariants contain a bunch of local sub-invariants, or in mathematical notation, $(t_i, l_i) \subset g$ for $i \in \{1, \ldots, n\}$.

For example, if someone were to prove the security of the composition of TLS and SSH using sign_pred in Figure 1, they would need to prove that:

- $(\texttt{"TLS"}, \mathsf{tls_sign_pred}) \subset \mathsf{sign_pred}$ (required by the TLS proofs)
- $(\texttt{"SSH"}, \mathsf{ssh_sign_pred}) \subset \mathsf{sign_pred}$ (required by the SSH proofs)

We now show how to create a global sub-invariant from a list of local sub-invariants and prove that this global sub-invariant contains every local sub-invariant.

Requirements from the user. Our framework requires the user to gather the set of tags and local sub-invariants $L = \{(t_1, l_1), \ldots, (t_n, l_n)\}$ used throughout every protocol step proof. Then, our framework requires users to prove tags are all distinct, namely $\forall i, j. t_i = t_j \implies i = j$. In practice, this can easily be proved by normalization, because tags are concrete values. We give an example of such an L for the signature sub-invariant of Figure 1 in Figure 3.

Building the global sub-invariant. Given the list of tags and local sub-invariants L, our framework provides a global sub-invariant $G(L)$. If L obeys the distinctness conditions, our framework proves that $\forall i. (t_i, l_i) \subset G(L)$. Therefore, this global sub-invariant $G(L)$ satisfies the requirements of every protocol step proof.

Internals of G. The definition of G is as follows: given an input x_G, compute $\mathit{decode}(x_G) = (t, x_L)$. If $t = t_i$ for some i, then $G(L)(x_G)$ is defined to be $l_i(x_L)$. If no such i exists, or if $\mathit{decode}(x_G) = \bot$, then $G(L)(x_G)$ is defined to be equal to a default value (supplied by the user).

When tags are not distinct. Our framework enforces the requirement that tags must be distinct. What happens if this is not the case? We argue that when tags are not distinct, this sheds light on a protocol weakness: in the example of Figure 1, this would give evidence that a given key is used for too many purposes, in the example of Figure 2, this would give evidence on a lack of domain separation.

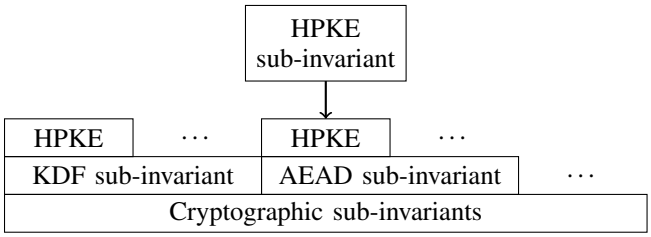

Figure 4: Graphical depiction of the cryptographic sub-invariants decision tree required by HPKE. HPKE relies on a local KDF sub-invariant and on a local AEAD sub-invariant that is obtained by compiling the global HPKE sub-invariant.

3.4. Case study: Hybrid Public-Key Encryption (HPKE)

We demonstrate the expressivity of our modular sub-invariant technique with a case study on Hybrid Public Key Encryption (HPKE) [9].

Modeling HPKE. We specify and prove a simplified model of HPKE [9], more specifically its Single-Shot API in the Base mode, which is used in MLS [8]. Internally, HPKE relies on three cryptographic primitives: Key Encapsulation Mechanism (KEM), Key Derivation Function (KDF), and Authenticated Encryption With Associated Data (AEAD).

Properties of HPKE. We want DY* users to be able to use HPKE as a Public-Key Encryption (PKE) algorithm. Therefore, we do not directly prove generic security properties for HPKE: instead, we prove lemmas that are similar to DY*'s native PKE algorithm. That is, we state and prove under what conditions HPKE functions preserve the bytes invariant. In DY*, using an authenticated encryption function typically preserves the bytes invariant under two conditions:

- Confidentiality: the plaintext is less secret than the private key used to encrypt (otherwise, the attacker could use the resulting ciphertext and the private key to learn something more secret than what they are allowed to know).
- Authenticity: a user-provided global predicate (the encryption sub-invariant) holds on the plaintext (for honest encryptions), or the plaintext security label is corrupt (for encryptions performed by the attacker).

We prove similar properties for HPKE, hence rely on a (global, user-provided) sub-invariant that is a predicate on HPKE plaintexts.

Proofs. Because HPKE internally relies on KEM, KDF and AEAD, its proof ultimately depends on the sub-invariants of these primitives. More precisely, as depicted in Figure 4, our proof works with any sub-invariants, as long as the underlying KDF sub-invariant contains a local sub-invariant for HPKE, and as long as the underlying AEAD sub-invariant contains a local sub-invariant for HPKE. Because the authenticity properties of HPKE ultimately boil down to the authenticity guarantees of the underlying AEAD, and because in DY* the authenticity properties of HPKE are captured by the HPKE sub-invariant, in our proof, the *local* AEAD sub-invariant for HPKE is compiled from the *global* HPKE sub-invariant.

Modular HPKE sub-invariant. DY* users may compose multiple protocols that internally rely on HPKE. For example, when these protocols use distinct keys, the users may write a global HPKE sub-invariant using again our modular sub-invariants technique and dispatch to local HPKE sub-invariants depending on the HPKE keypair usage (similarly to Figure 1).

Discussion. This case study demonstrates the versatility of our framework: it allows for horizontal composition (our proof can be composed with other protocols that use KDF or AEAD) and for vertical composition (as showcased by compiling the global HPKE sub-invariant to a local AEAD sub-invariant). Finally, proofs on protocols that use HPKE can be composed together by applying our framework again on the HPKE sub-invariant: we will see an example for this in the next section.

3.5. Case study: TreeKEM

In [44], Wallez et al. used our sub-invariant framework to conduct security proofs for the TreeKEM sub-protocol of MLS [8]. Their security proof features vertical composition because TreeKEM relies on the guarantees offered by HPKE (§3.4) and by TreeSync [42]. It also features horizontal composition: HPKE is used in two places in TreeKEM with a domain separator, AEAD is used both in HPKE and in TreeKEM, KDF is used in HPKE and in multiple places of TreeKEM; furthermore, TreeKEM relies on multiple signatures (via TreeSync) that use the same signature key but are domain-separated. Finally, the proofs of each subcomponent of TreeKEM (e.g. the tree, the key schedule, or the welcome process) are independent of each other, since they only need to reason with their local sub-invariants, and are unaffected by changes in local sub-invariants of unrelated subcomponents.

This case study shows that our sub-invariant framework scales well and can handle complex cryptographic protocols with substantial cryptographic sub-invariants. We further argue that it was crucial to Wallez et al.'s success: if they had written a single big monolithic cryptographic sub-invariant for TreeKEM, it would have spanned roughly around a thousand lines of code, it would have been difficult to maintain, and each time it would be tweaked, existing proofs would risk to break (even the proofs not related to the tweak). Therefore, we believe this modular sub-invariant framework was a crucial technique for the success of Wallez et al. [44] and will be an essential tool for the next generation of security proofs performed using our new version of DY*.

4. Security Labels as Trace Predicates

In symbolic protocol provers (such as DY*, but also ProVerif and Tamarin) confidentiality theorems are of the form "if the attacker knows this bytestring (e.g. a secret key), then they must have performed these type of compromises". To prove this kind of property, DY* relies on a construction called *security labels*: each bytestring is associated with a label that encodes an over-approximation of the compromise scenarios that may lead the attacker to know this bytestring. For example, the label of Alice's private signature key may ensure that if the attacker knows this key, then they must have compromised one of Alice's states. Note that this is an over-approximation because the converse is not true: for example, the attacker may have compromised the state of Alice where she stores pictures of cats, but this does not reveal Alice's private signature key to the attacker.

Defining security labels. Security labels are built using three ingredients:

- a type for labels and functions to construct them (so that the label type can remain abstract to users),
- a predicate is_corrupt that tells whether the attacker compromises fall within the over-approximation encoded by the label,
- and a relation can_flow that tells whether a label is less secret than another.

The label of a bytestring restricts how the bytestring can be used in a cryptographic protocol: for example, a plaintext can be encrypted with a key only when the key is more secret (as per can_flow) than the plaintext. This ensures that, for example, if the attacker knows the key (hence can decrypt and obtain the plaintext), the plaintext label is corrupt (because the key label is both corrupt and more secret than the plaintext label), hence stays a correct over-approximation of the compromise scenarios that may lead the attacker to know the plaintext.

In the original DY* [11], although security theorems can express compromise scenarios using the full expressivity of F*, the expressivity of security labels is restricted to a simple language: in addition to conjunctions and disjunctions, they can talk about the compromise of a participant, or of a specific state of a participant, identified by a state identifier (an integer), or of a specific version of that state, identified by a version identifier (an integer). The state identifier itself is just a pointer and does not convey any information on whether the state contains an ephemeral key or a long-term signature key. Furthermore, using these labels, one cannot express fine-grained temporal guarantees: for example that a signature key was compromised *before* a participant verified some corresponding signature (e.g. in a handshake protocol).

Limitation in the original DY*. We therefore notice a mismatch. On the one hand, we can use the full expressivity of F* to describe compromise scenarios in security theorems. On the other hand, the language for security labels is restricted. Because confidentiality proofs internally rely on security labels, this means that even if we were able to prove a precise confidentiality theorem on a key exchange protocol (written in F*), we might not be able to encode the same compromise scenario in a security label and therefore rely on this result when composing our key exchange with another protocol that uses the exchanged key. To conduct such a proof, we would need to use a security label that over-approximates the compromise scenario of the key exchange security theorem, thereby losing precision on the compromise scenarios of the security theorem for the composed protocol.

```
type label =
  | Participant: participant → label
  | Join: label → label → label
  | Meet: label → label → label
  | Public: label
  | Secret: label
```

Figure 5: Inductive type for labels.

Contributions. In this section, we present a new labeling framework, where security labels can be expressed using the full expressivity of F*, thereby merging the two ways in which DY* can express compromise scenarios. As such, our new labeling framework is the most general one; in particular, it can express the compromise of some particular state type (e.g. signature key) and temporal properties (e.g. compromise of a signature key before some event). Our new labeling framework was already used by and crucial to the security proofs of the TreeKEM protocol [44].

Soundness. The soundness of security labels in DY* is justified by the Attacker Knowledge Theorem (discussed in §2.3), which says that if the attacker knows a bytestring, then the compromise scenario encoded by the label of this bytestring must have happened.

The soundness of our new version of DY* is still mechanically checked in F* (as is the original DY*); in particular, we updated the proof of the Attacker Knowledge Theorem with our new labeling framework.

Outline. We present the new labeling framework in four steps:

- First, we present the definition of labels in the original DY* [11].
- Second, we simplify the definition of is_corrupt and can_flow.
- Third, we change the label type and is_corrupt so that labels can use the full expressivity of F*.
- Finally, we show how the security proof of TreeKEM leveraged this new labeling framework.

4.1. Labels in the original DY*

We first give an introduction to the design of labels in the original DY* [11], before showing a new and simpler design in the next section.

The label type. Labels are defined as an Abstract Syntax Tree (AST) with one node per operation on labels (e.g. meet

or join) or per base label (e.g. the public label or label corresponding to a participant), shown in Figure 5.

The can_flow relation. In the original DY* [11], can_flow is defined by doing a case analysis on the two labels being compared (as shown in Figure 6a), therefore its definition size is quadratic in the number of node types in the label AST. Furthermore, this definition of can_flow must be a partial order, i.e. it must be reflexive and transitive. The transitivity proof is done by doing a case analysis on the three labels involved, resulting in a proof script whose size is cubic in the number of node types in the label AST. Then a linear number of properties with linear proof script must be proved, e.g. public satisfies the greatest element property, join satisfies the least upper bound property, etc. Finally, a label is defined to be corrupt when it can flow to the public label.

4.2. Simplifying labels

We design a new version of can_flow (shown in Figure 6b) whose definition size is linear in the number of node types in the label AST, and where every related proof (e.g. reflexivity, transitivity, least upper bound property) is of constant size. As a result, this new version of can_flow is easier to adapt if we extend the label AST with new node types.

Defining is_corrupt. In Figure 6b, we define is_corrupt directly, instead of defining it from can_flow (as in Figure 6a). The label Participant is corrupt when there exists a compromise event in the trace for the corresponding participant. The Public label is always corrupt (regardless the compromises from the attacker). Finally, the Join label is interpreted as a disjunction, and the Meet label is interpreted as a conjunction.

Defining can_flow from is_corrupt. We define that a label l1 flows to a label l2 with respect to a trace tr when for any possible way to extend the trace (e.g. by adding compromise events) the label l1 is corrupt when the label l2 is corrupt. This means it is impossible for the attacker to corrupt l2 without corrupting l1, which reflects the intuition that l1 is "less secret" than l2.

Proving properties of can_flow. With this new definition of can_flow, most properties are now easy to prove: for example, transitivity of can_flow follows immediately from transitivity of implication, and, Public flows to any label because anything implies $\top$.

Relationship between the old and the new can_flow. We formally prove that the previous and new is_corrupt are equivalent and that our new can_flow definition is the largest label flow relation that satisfies these three properties: (1) it is transitive, (2) it stays true when the trace grows, (3) flowing to public is equivalent to being corrupt. In particular, the previous can_flow obeys these three properties and therefore implies our new can_flow.

4.3. Generalizing labels

We now show how we modify the label type and is_corrupt to enable users to leverage the full expressivity of F* to define labels. We argue that our new labeling framework is the most general one, since any labeling framework implemented in F* cannot be more expressive that F* itself.

Labels as trace predicates. Looking back at §4.2, we may understand labels as being a deep-embedded domain-specific language (DSL) for trace predicates, where is_corrupt (Figure 6b) is an interpreter for this DSL. We use this insight to design the most general label type possible: our new labels are now full-fledged trace predicates, which then benefit from the full expressivity of F*. The labels presented in Figure 6b can now be implemented outside of DY*'s core (see Figure 7), and now nothing prevents users from implementing other more expressive labels, as we will see in §4.4.

New definition of can_flow. We show the new definition of is_corrupt and can_flow in Figure 8. The definition of can_flow is the same as in the previous section (see Figure 6b), and the definition of is_corrupt is deferred to labels themselves (e.g. compare the join label in Figure 7 and the Join case in Figure 6b).

4.4. Examples of more expressive labels

In the security proof of TreeKEM [44], Wallez et al. rely on these new, more expressive labels, and are able to define new types of labels needed to conduct their security proof without having to fork DY*, demonstrating the success and generality of our approach. We now dig into the example of the label for a signed ephemeral key in TreeKEM, which introduces along the way several general-purpose label combinators that could not have been expressed in the labels of the original DY*, and which we have since upstreamed in the standard library of our new version of DY*.

Signed ephemeral key. In TreeKEM, an ephemeral public encryption key is signed using a long-term signature key (in the so-called "key package"). After a participant has verified this signature, we want the following guarantee: when this participant encrypts data with this ephemeral public encryption key, this data is safe from the attacker, unless (1) the attacker compromised the corresponding private decryption key, or (2) the attacker compromised this long-term signature key *before* this participant checked the signature. We allow for such a proof by designing a label encompassing these two possibilities, and proving that this label is *less secret* than (i.e. can flow to) the ephemeral private decryption key label. This ensures that if the attacker knows the ephemeral private decryption key, then the label we designed is corrupt, from which we deduce that one of the two aforementioned possibilities happened.

Label for signed ephemeral key. Because this label is stated as a disjunction, it is naturally expressed as the

```
let rec can_flow (tr:trace) (l1 l2:label) =
  match l1, l2 with
  | Participant p, Public →
exists_compromise_event tr p
  | Join l11 l12, Join l21 l22 → ...
      (can_flow tr l1 l21 ∧ can_flow tr l1 l22) ∨
      (can_flow tr l11 l2 ∨ can_flow tr l12 l2)
  | Participant p, Meet l21 l22 → ...
  | Meet l11 l12, Join l21 l22 → ...
  ...

let is_corrupt (tr:trace) (l:label) = can_flow tr l Public
```

(a) The can_flow relation as defined in the original DY* [11]. Parts omitted for brevity.

```
let rec is_corrupt (tr:trace) (l:label) =
  match l with
  | Participant p → exists_compromise_event tr p
  | Join l1 l2 → is_corrupt tr l1 ∨ is_corrupt tr l2
  | Meet l1 l2 → is_corrupt tr l1 ∧ is_corrupt tr l2
  | Public → ⊤
  | Secret → ⊥

let can_flow (tr:trace) (l1 l2:label) =
  ∀tr_later. tr ≤ tr_later ⟹
    (is_corrupt tr_later l2 ⟹ is_corrupt tr_later l1)
```

(b) Our new definition of can_flow. The definition as shown here is complete, no parts were omitted.

Figure 6: Definition of labels in DY*

```
let participant_label (p:participant) = {
  is_corrupt = (λ tr →
    exists_compromise_event tr p
  );
}
let join (l1 l2:label) = {
  is_corrupt = (λ tr →
    l1.is_corrupt tr ∨ l2.is_corrupt tr
  );
}
```

Figure 7: Example of labels as trace predicates.

```
let is_corrupt (tr:trace) (l:label) =
    l.is_corrupt tr

let can_flow (tr:trace) (l1 l2:label) =
    ∀tr_later. tr ≤ tr_later ⟹ (
        is_corrupt tr_later l2 ⟹
        is_corrupt tr_later l1
    )
```

Figure 8: The new definition of can_flow, with labels as trace predicates.

join of two labels: eph_key_label, which is corrupt when the attacker compromised the ephemeral key state (case (1)), and temporal_sig_key_label, which is corrupt when the attacker compromised the signature key state *before* the participant verified the signature (case (2)). Upon signature verification, we deduce from the properties of signatures that either (a) the signature was performed honestly, in which case the signature predicate must hold on the ephemeral key, otherwise (b) the signature was performed by the attacker, in which case they must have known the corresponding signature key before the participant verified the signature, hence temporal_sig_key_label must be corrupt. This allows us to prove that the join of eph_key_label and temporal_sig_key_label can flow to (or is "less secret than") the ephemeral key label. In case (a), by designing the

```
let eph_key_label
  (p:participant) (pk:bytes) = {
    is_corrupt = (λ tr →
      ∃st.
        exists_compromise_event tr st ∧
        st.who == p ∧
        st.tag == "EphemeralKey" ∧
        sk_to_pk st.content.sk == pk
    );
}
```

Figure 9: Precise label for a ephemeral private key of participant p corresponding to the public key pk.

signature predicate such that it implies that eph_key_label can flow to the ephemeral key label, we directly conclude. In case (b), because temporal_sig_key_label must be corrupt, it can flow to the public label, and we can also directly conclude. We now describe how we are able to define eph_key_label and temporal_sig_key_label using our new labeling framework.

Precise state. In participant_label p (Figure 7), the corruption predicate says "there exist a state of participant p that is compromised by the attacker", but this does not express what this state actually contains. We can now build more precise labels, as shown in Figure 9: eph_key_label further says that this state contains an ephemeral key (using the tag "EphemeralKey") and that this ephemeral private key corresponds to some given public key pk. We can design a similar label for signature keys (sig_key_label), but to define temporal_sig_key_label, we still need to deal with temporal properties.

Temporal properties. We express temporal properties with a new label combinator: guarded_event l e, which is corrupt when the label l is corrupt before the event e was logged. If e is an event that is logged after verifying the ephemeral key signature, we define temporal_sig_key_label to be guarded_event (sig_key_label p vk) e, which correctly expresses a temporal property: it is corrupt when the at-

```
let guarded (l1 l2:label) = {
  is_corrupt = (λ tr →
    ∃tr_before.
      tr_before ≤ tr ∧
      l1.is_corrupt tr_before ∧
      ¬(l2.is_corrupt tr_before)
  );
}
```

Figure 10: Implementation of guarded.

```
let event_label (p:participant) (e:event) =
{
  is_corrupt = (λ tr →
    event_was_logged tr p e
  );
}
```

Figure 11: Implementation of event_label.

tacker compromised the signature key before the participant verified the ephemeral key signature. The label combinator guarded_event is itself built using three different combinators, which we now discuss.

The guarded label combinator. The first building block of guarded_event is the guarded label combinator (Figure 10): guarded l1 l2 is corrupt when there exists a time in the past where l1 was corrupt but l2 was not corrupt. This label combinator expresses a temporal property. In combination with the event label (see next paragraph), it can be used to express temporal properties with respect to an event (e.g. verification of a signature).

Event labels. The second building block of guarded_event is the event label (Figure 11): event_label p e is corrupt when participant p has logged the event e. We can combine it with guarded to almost obtain guarded_event: guarded l (event_label p e) is corrupt when l was corrupt before participant p logged the event e. The last problem to solve before obtaining guarded_event is the fact that participant p appears in this label. In security proofs with DY*, it is generally useful that different participants associate the same label for the same piece of data (here, the label for the ephemeral private key). This cannot happen if the participant p who is checking the signature appears in the ephemeral private key label.

Participant-independent event label. To solve this problem, a key insight is that to conduct an active attack by forging the signature, the attacker must compromise the signature key before *any* participant checked this signature (which is in particular before participant p checks this signature). To implement this insight, we could modify event_label to be corrupt when any participant logged some event. However, a more elegant solution is to keep our general event_label and use it inside a join over all possible participants: this would correspond to the fact that any participant logged this event. In the original DY* [11], we can only do joins over two

```
let unbounded_join (f:α → label) = {
  is_corrupt = (λ tr →
    ∃(x:α) . (f x).is_corrupt tr
  );
}
```

Figure 12: Implementation of unbounded join.

labels, and we can repeat it to do joins over a finite number of labels. However, we cannot do it for an infinite number of labels: with our new labeling framework, it is now possible.

Unbounded join and meet. Looking back at Figure 7, join between two labels corresponds to the logical or. To implement an unbounded version of join, we therefore use the unbounded version of the logical or, which is the logical exists, as shown in Figure 12. We can implement unbounded meet similarly using a logical forall.

Discussion. We have seen that using our new labeling framework, we can now define new labels expressing intricate security properties, such as the label for signed ephemeral keys in TreeKEM, which would have been impossible to express in the labeling system of the original DY* [11]. Along the way we saw new general-purpose labels, such as guarded (Figure 10), event_label (Figure 11), and unbounded_join (Figure 12), that we have since upstreamed into DY*'s standard library, so that they can be used by the next generation of security proofs performed using our new version of DY*. Finally, because our new labeling framework is the most general one, users of our new version of DY* are not limited by the labels we have implemented in the standard library: should a general-purpose label be missing, users are free to define it themselves as part of their security proof, as have done Wallez et al. [44] in their security proof before we upstreamed their general-purpose labels.

5. Improving Specification Auditability

In the broad DY* methodology, users write a trace invariant and then prove that each protocol step preserves this trace invariant, which ultimately shows that every trace reachable by an attacker must satisfy this trace invariant. In the original DY* [11], the proof that a protocol step preserves the trace invariant is performed at the same time the protocol step is specified, by interleaving lines of code for specifications and lines of code for proofs. This style of doing proofs is also known as *intrinsic proofs*. Intrinsic proofs are popular in program verification, where an efficient implementation is usually shown to adhere to a specification, which is then audited. In this case, it is acceptable for the implementation code to be interleaved with proofs, since what must be audited is the specification, which is free of lines of code related to proofs. However, in the original DY*, it is the specifications that are interleaved with proofs: we therefore argue that in the context of DY*, intrinsic proofs hinder auditability of specifications. The original

DY* paper [11] advocates auditing specifications through *executability*, by executing protocol specifications against test vectors or interoperability testing. However, we argue that this method should complement human auditing instead of replacing it. In this section, we present a new approach toward proofs in DY*, which improves the auditability by humans by offering a clean separation between specification and proofs, while keeping the possibility to execute specifications.

Particular challenges in DY* proofs. In DY*, a protocol step is an F* function that extends the trace and is itself composed of several sub-steps (e.g. sending a message on the network, retrieving some state, or performing computations) that may extend the trace gradually. Therefore, to prove that a protocol step preserves the trace invariant under some pre-conditions, we prove that each one of its sub-steps preserves the trace invariant (and prove their corresponding pre-conditions). The main challenge is that the proof must reason on the several traces that exist throughout the execution of a protocol step (that is, one trace per sub-step it contains), and that a property (e.g. bytes invariant) we may prove with respect to one of these traces may not hold with respect to another one.

Fortunately, this issue is mitigated by a key characteristic of properties related to traces in DY*: they are monotone with respect to the trace, that is, if they hold with respect to one trace, they still hold when the trace is extended. Since every sub-step extends the trace, this means that once we prove some property with respect to a trace, we can deduce that this property still holds after the execution of subsequent sub-steps.

The original DY* [11] exploits this key characteristic of DY* properties by requiring users to write specifications in the effect system of F* [38] (which imposes doing proofs in an intrinsic style) and then use its witness mechanism [4] to leverage the monotonicity of DY* properties.

Our approach toward DY* proofs. We propose a different approach that allows for *extrinsic proofs*, that is, it allows to cleanly separate specifications from their invariant preservation proofs. We depict our approach in Figure 13. DY* users write protocol specifications in a lightweight monad that combines a state monad on traces with an error monad (using refinement types to ensure the trace is extended), and write protocol (sub-)steps in this lightweight monad. Then, they are able to write theorems on how this monadic function preserves the trace invariant, under which pre-conditions, and what post-condition it guarantees. Protocol steps that are exposed to the attacker must have trivial pre-conditions, meaning that they preserve the trace invariant with any input. We exploit the monotonicity of properties in DY* by leveraging F*'s SMT patterns, which allow to automatically instantiate the monotonicity lemmas when they are needed in invariant preservation proofs, so that when a property has been established for a given trace, F* automatically deduces that it holds for every subsequent trace.

```
(*** Types, functions and theorems provided by DY* ***)

// lightweight monad for monotone state + error (§5)
type traceful a =
 tr_in:trace → option a & tr_out:trace{tr_in < tr_out}

// DY* invariants are parametrized by protocol-specific
// sub-invariants to be defined by the user (§2, §3)
// (similar declarations for trace_invariant, get_label, ...)
val bytes_invariant: sub_invariant → trace → bytes → prop

// example monotonicity lemma (§5)
val bytes_invariant_later:
 subinvs: sub_invariants →
 tr1:trace → tr2:trace → b:bytes → Lemma
 (requires bytes_invariant subinvs tr1 b ∧ tr1 < tr2)
 (ensures bytes_invariant subinvs tr2 b)
 // SMT pattern to instantiate this lemma automatically
 [SMTPat (bytes_invariant subinvs tr1 b);
  SMTPat (tr1 < tr2)]

(*** Example usage by a DY* user ***)

// a protocol step (or sub-step, written by the user)
let some_sub_step (b:bytes): traceful bytes =
 // the specification can be tidy, free of proof artifacts (§5)

let complex_label: label =
 // precise compromise scenario using our new framework (§4)

// trace invariant preservation theorem (written by the user)
val some_sub_step_proof:
 // note that proof is generic in sub-invariants (§3)
 subinvs: sub_invariants → b:bytes → tr:trace → Lemma
 (requires
  // if sub-invariants contain some local invariants (§3)
  contains_my_local_invs subinvs ∧
  // if trace invariant holds before execution of protocol step
  trace_invariant subinvs tr ∧
  // and some pre-condition holds (example)
  bytes_invariant subinvs tr b
 )
 (ensures (
  // after execution of the protocol step
  let (opt_res, tr_out) = some_sub_step b tr in
  // trace invariant must hold regardless of success or failure
  trace_invariant subinvs tr_out ∧ (
   match opt_res with
   | None → ⊤// failure: we don't prove anything else
   | Some res → ( // in case of success,
    // we can deduce a post-condition (example)
    bytes_invariant subinvs tr_out res ∧
    // precise secrecy guarantees on the result (§4)
    complex_label `can_flow tr_out` (get_label subinvs res)
   )
  )
 ))
let some_sub_step_proof b tr =
  // proof goes here, it does not need to be audited,
  // hence can be nasty without hindering auditability (§5)
```

Figure 13: F* code showcasing our DY* overhaul that allows for composing proofs (§3), expressing precise compromise scenarios (§4) and more auditable specifications (§5).

Discussion. We presented an approach toward extrinsic proofs in DY*. We note that our methodology does not rely on ubiquitous features of F*, such as effects [38] or witnesses [4]; instead, it relies on standard mechanisms that are present in several other proof assistants, such as Rocq [41], Lean [35] and Isabelle [36]. Our approach offers executability of specifications by allowing users to parametrize their specification by a typeclass, so that the specification can be executed by instantiating it with a concrete typeclass (e.g. using HACL* [46]), and the specification can be proved secure by instantiating it with DY*. This approach was successfully used by Wallez et al. in their security proof for TreeKEM [44].

Limitations and future work. Although we successfully separate specifications from their invariant preservation proofs, there is still room for improvement regarding the auditability of specifications in our new version of DY*. We now discuss how specifications still contain a small proof artifact when generating fresh random bytestrings, and also still contain artifacts from the execution model.

First, specifications still contain one small proof artifact: when generating a fresh random bytestring, DY* requires the user to provide the security label of this bytestring at the time of generation in the specification. This is because the corresponding trace entry contains the security label for this fresh random bytestring. As a result, this security label is present in the specification, despite being a proof artifact. Although it would be nice to remove this last proof artifact from specifications, it does not significantly hinder specification auditability, we leave this task as future work.

Second, even if specifications are now (almost) free of any proof artifacts, they still contain artifacts from the execution model of DY*. For example, when receiving a message, specifications must supply the timestamp (i.e. index in the trace) when the message was sent (this allows the attacker to choose which message is received), and when storing or retrieving state, specifications must supply a state identifier (which can be thought of as a pointer). We leave this task of hiding the execution model from specifications as future work.

Finally, as noted by Arquint et al. [5], specifications in DY* must follow the discipline that each function only implements one protocol action, because functions are executed atomically: if a function were to send a message m_1 before receiving a message m_2, the attacker would not be able to use the new knowledge of m_1 when supplying the message m_2, which could result in missed attacks. Therefore, functions in DY* specifications follow the pattern of (1) receiving message and retrieving state, (2) performing cryptographic computations, and (3) sending message, storing state, and logging protocol events. We note that this way of specifying protocols is similar to the state of the art and widely used symbolic prover Tamarin [33]: therefore, this style of writing specification is natural for people familiar with Tamarin. However, it is true that DY* could allow for more applied pi-calculus style of specifications like ProVerif [14], but we leave this task for future work.

6. Related Work

In this paper, we have presented a new version of DY*, a symbolic analysis framework, which now allows for composing security proofs. We compare our work against other works in this area.

Semi-automated analysis in the symbolic model. Several tools allow for (semi-)automated analysis of cryptographic protocols in the symbolic model, ProVerif [14] and Tamarin [33] being leading examples of this area. These tools have better automation than DY* (both the original and our new version), which requires users to manually write trace invariants in order to prove security properties. However, they are only able to perform whole-protocol analysis, unlike our new version of DY* presented in this paper, which allows for both modular verification (in the sense that each protocol step can be verified independently) and composing security proofs (in the sense that if someone proved the security of two protocols, they can easily prove security properties on their composition).

Symbolic analysis in proof assistants. The original DY* [11] and its predecessors allow for a modular verification of cryptographic protocols. However, unlike this work, they do not allow for composing security proofs.

Arquint et al. [5] implement a framework in the program verifier Gobra [45] that allows for proving the security of cryptographic protocol implementations in the symbolic model, using a methodology similar to the original DY* [11]. On the one hand, their approach has the same limitations of the original DY* [11] that we address in this paper: although their proofs are modular, they cannot compose; they use the same label construction of the original DY* (see Figure 6a), so they may lose precision on compromise scenarios when studying large composite protocols; they rely on intrinsic proofs, although this does not hinder specification auditability as much as the original DY* because their proof annotations are written as Go comments. On the other hand, they address some limitations of the original DY* [11] that are still limitations of the new version we present in this paper. First, they note that the original DY* requires users to follow the discipline that functions given to the attacker must perform no more than one protocol action (which is discussed in §5 and is similar to specifications in Tamarin [33]): they lift this limitation by using a concurrent execution model, thereby allowing the attacker to interleave their computations during the execution of a function exposed by a participant. Second, they note that DY* does not provide a nice proof mechanism to prove injective authentication properties, and they leverage separation logic to allow for such proofs: although DY* can prove injectivity properties in some scenarios (for example when a protocol event containing a nonce is logged directly after this nonce is generated, DY* can prove that only one such event exists), their use of separation logic may allow for injectivity proofs in more complex scenarios.

Cryptis [6] uses Iris [27] and Rocq [41] to implement a separation logic to prove security properties in the symbolic

model. Like us, they also allow for composing security proofs, but unlike us, they do so by imposing a strong constraint on the protocols they can analyze: they assume that every cryptographic primitive takes a tag as input, and in that case, they can compose proofs for protocols that use disjoint tags (for each cryptographic primitive). Therefore, if two protocols can be securely composed because they use disjoint *keys*, Cryptis cannot compose their security proofs. In other words, whereas we support a broad range of protocol compositions because we can compose security proofs when invariants can be shaped as a decision tree (discussed in §3.1), Cryptis imposes the invariant decision tree to have only one level that dispatches on a domain separator, which limits the range of protocols they can compose. Like our approach, Cryptis is able to express precise compromise scenarios when analyzing large composite protocols: internally, Cryptis reasons on the compromise scenario under which a bytestring b is known to the attacker through the separation logic predicate public b, which therefore benefits from the full expressivity of separation logic. We note that in our new DY*, each bytestring is associated with two trace predicates, the bytes invariant and whether its label is corrupt, whereas Cryptis relies on only one predicate, public, which is similar to the conjunction of our two predicates. This raises the question why Cryptis only needs one predicate when we need two. One answer could be that we support a broader range of cryptographic functions (e.g. including key derivation functions and key encapsulation mechanisms), having two predicates could be useful in this regard (but we do not have evidence they are necessary). Another answer could be that Cryptis benefits from Iris' higher-order ghost state, whereas in DY* we use two levels of trace predicates (i.e. the bytes invariant can talk about labels, but not vice versa) to compensate for not having higher-order ghost state. Finally, regarding auditability of specifications, Cryptis (like us) allows for extrinsic proofs and (unlike us) allows specifying several protocol actions per function thanks to a non-deterministic scheduler.

Composing security proofs in DY*. Bhargavan et al. [12] also propose an approach toward composing security proofs in DY*, which handles protocols that can be described in layers. Their approach is to write a trace invariant for the lowest layer, which is parametrized by invariants for higher layers, and do this for every layer going up in the stack. We allow for a similar technique as showcased in §3.4, and therefore support every composition that their technique allows. Furthermore, their approach only works when the dependency graph between subcomponents can be ordered linearly in a vertical fashion; in particular, unlike us, they do not handle the case where a subcomponent relies on two (or more) subcomponents, and they do not handle *horizontal composition*, where several protocols (which may share secret keys) are run in parallel. Therefore, our methodology is strictly more comprehensive and allows for a broader range of composition techniques, including vertical composition (where a component relies on the guarantees provided by another component) and different flavors of horizontal composition (when two components are safe to run in parallel, for example when they use distinct keys or when they reuse keys but with proper domain separation).

Wallez et al. [43] propose criteria under which cryptographic protocols can be composed, using the viewpoint of message formats. They further build a tool called Comparse to analyze properties of message formats in F*, and they integrate Comparse with the original DY*. We also integrate Comparse in our new version of DY*, and in §3.1 we showed that protocols that fall under the composition criteria of Comparse [43] can be composed using our framework.

General composition results in the symbolic model. Several works establish general results on the composition of cryptographic protocols in the symbolic model [18], [19], [20], [24], [25], [26], [34]. Although we do enable composing security proofs in DY*, our result does not apply in full generality to the symbolic model; instead, we leverage the methodology of DY* to perform security proofs using a *trace invariant*, and we enable the composition of proofs via a modular trace invariant definition. These works are not integrated into tools, whereas DY* users directly benefit from using our methodology to compose security proofs. In addition, these works are meta-results proved on pen and paper, whereas the soundness of our approach is guaranteed by DY*: DY* requires users to find a trace invariant and show that it obeys some property, and our approach is simply a new way to create such a trace invariant.

Composition in the computational model. Composing security proofs for cryptographic protocols has a long history, for example with the line of work on Universal Composability [15]. Some tools in the computational model aim to provide modularity [3], [16], [23]. By working on the computational model, these tools provide more precise security guarantees than DY* which works in the symbolic model. On the other hand, our new version of DY* scales well to large protocols (as exemplified by the security proof of TreeKEM [44]), which are still challenging for tools that provide computational guarantees.

7. Conclusion

We presented a new version of DY* with several core improvements to enable the analysis of large-scale protocols. In particular, our new version of DY* and our improvements were already used by and crucial to the security proofs of the TreeKEM protocol [44]. We enable the composition of security proofs in DY* by designing a modular framework to create trace invariants, which allows for a broad range of protocol compositions, including vertical composition (a component relies on the guarantees provided by another component) and horizontal composition (two components that may share keys run in parallel). We generalize the notion of security labels, which ultimately allows for describing precise compromise scenarios in the security theorems of complex protocols. Finally, we improve the auditability of DY* artifacts by allowing for a clean separation between protocol specifications and their security proofs.

Acknowledgments

We warmly thank the reviewers for their thoughtful comments, as well as Karthikeyan Bhargavan, Cas Cremers and Jonathan Protzenko for their proofreading of draft of this paper.

This work received funding from the French Government, managed by the ANR under grant agreements ANR-19-P3IA-0001.

References

[1] DY* GitHub repository, 2026. https://github.com/REPROSEC/dolev-yao-star-extrinsic.

[2] DY* Unchained: Supplementary material, 2026. https://github.com/TWal/dy-unchained-artifact.

[3] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 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*, pages 1–15. IEEE, 2021.

[4] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL):65:1–65:30, 2018.

[5] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. A generic methodology for the modular verification of security protocol implementations. In *Computer and Communications Security (CCS)*, CCS '23, pages 1377–1391, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Arthur Azevedo de Amorim, Amal Ahmed, and Marco Gaboardi. Cryptis: Cryptographic reasoning in separation logic. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

[7] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *IEEE Symposium on Security and Privacy (S&P)*, pages 777–795, 2021.

[8] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[9] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A Wood. RFC 9180: Hybrid public key encryption. Technical report, Internet Research Task Force, 2022.

[10] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32, 2008.

[11] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A modular symbolic verification framework for executable cryptographic protocol code. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.

[12] Karthikeyan Bhargavan, Abhishek Bichhawat, Pedram Hosseyni, Ralf Küsters, Klaas Pruiksma, Guido Schmitz, Clara Waldmann, and Tim Würtele. Layered symbolic security analysis in DY*. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *Computer Security – ESORICS 2023*, pages 3–21, Cham, 2024. Springer Nature Switzerland.

[13] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 483–502, 2017.

[14] Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

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

[16] Ran Canetti, Alley Stoughton, and Mayank Varia. Easyuc: Using easycrypt to mechanize proofs of universally composable security. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 167–183. IEEE, 2019.

[17] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1445–1459, 2020.

[18] Vincent Cheval, Véronique Cortier, and Bogdan Warinschi. Secure composition of pkis with public key protocols. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 144–158. IEEE Computer Society, 2017.

[19] Ştefan Ciobâcă and Véronique Cortier. Protocol composition for arbitrary primitives. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 322–336. IEEE Computer Society, 2010.

[20] Véronique Cortier, Jérémie Delaitre, and Stéphanie Delaune. Safely composing security protocols. In Vikraman Arvind and Sanjiva Prasad, editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings*, volume 4855 of *Lecture Notes in Computer Science*, pages 352–363. Springer, 2007.

[21] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 1773–1788, 2017.

[22] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983.

[23] Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. 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*, pages 1130–1147. IEEE, 2023.

[24] Sébastien Gondron and Sebastian Mödersheim. Vertical composition and sound payload abstraction for stateful protocols. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021.

[25] Thomas Groß and Sebastian Mödersheim. Vertical protocol composition. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 235–250. IEEE Computer Society, 2011.

[26] Andreas V. Hess, Sebastian Alexander Mödersheim, and Achim D. Brucker. Stateful protocol composition in isabelle/hol. *ACM Trans. Priv. Secur.*, 26(3):25:1–25:36, 2023.

[27] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:20, 2018.

[28] Ehren Kret and Rolfe Schmidt. The PQXDH key agreement protocol, 2023. https://signal.org/docs/specifications/pqxdh/.

[29] Joshua Lund. Technology preview: Sealed sender for signal, 2018. https://signal.org/blog/sealed-sender/.

[30] Moxie Marlinspike and Trevor Perrin. Signal specifications, 2016. https://signal.org/docs.

[31] Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol, 2016. https://signal.org/docs/specifications/x3dh/.

[32] Moxie Marlinspike and Trevor Perrin. The sesame algorithm: Session management for asynchronous message encryption, 2017. https://signal.org/docs/specifications/sesame/.

[33] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*, pages 696–701. Springer, 2013.

[34] Sebastian Mödersheim and Luca Viganò. Sufficient conditions for vertical composition of security protocols. In Shiho Moriai, Trent Jaeger, and Kouichi Sakurai, editors, *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 435–446. ACM, 2014.

[35] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[36] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[37] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm, 2016. https://signal.org/docs/specifications/doubleratchet/.

[38] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Programming and proving with indexed effects, 2021. https://www.fstar-lang.org/papers/indexedeffects/.

[39] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[40] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.

[41] The Coq Development Team. The coq proof assistant, September 2024.

[42] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: Authenticated group management for messaging layer security. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1217–1233, August 2023.

[43] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. Comparse: Provably secure formats for cryptographic protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 564–578, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. TreeKEM: A modular machine-checked symbolic security analysis of group key agreement in messaging layer security. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 4375–4390, 2025.

[45] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*, volume 12759 of *LNCS*, pages 367–379. Springer International Publishing, 2021.

[46] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1789–1806, 2017.