Sound Verification of Security Protocols: From Design to Interoperable Implementations

Linard Arquint* , Felix A. Wolf* , Joseph Lallemand†, Ralf Sasse* ,
Christoph Sprenger* , Sven N. Wiesner* , David Basin* , and Peter Müller*
*Department of Computer Science, ETH Zurich, Switzerland
†Univ Rennes, CNRS, IRISA, France
{linard.arquint, felix.wolf, ralf.sasse, sprenger, sven.wiesner,
basin, peter.mueller}@inf.ethz.ch, joseph.lallemand@irisa.fr

***Abstract*—We provide a framework consisting of tools and metatheorems for the end-to-end verification of security protocols, which bridges the gap between automated protocol verification and code-level proofs. We automatically translate a Tamarin protocol model into a set of I/O specifications expressed in separation logic. Each such specification describes a protocol role's intended I/O behavior against which the role's implementation is then verified. Our soundness result guarantees that the verified implementation inherits all security (trace) properties proved for the Tamarin model. Our framework thus enables us to leverage the substantial body of prior verification work in Tamarin to verify new and existing implementations. The possibility to use any separation logic code verifier provides flexibility regarding the target language. To validate our approach and show that it scales to real-world protocols, we verify a substantial part of the official Go implementation of the WireGuard VPN key exchange protocol.**

***Index Terms*—Protocol verification, Symbolic security, Automated verification, Tamarin, Separation logic, Implementation.**

1. Introduction

Security protocols are central to securing communication and distributed computation and, by nature, they are often employed in critical applications. Unfortunately, as history amply demonstrates, they are notoriously difficult to get right, and their flaws can be a source of devastating attacks. Hence the importance of their formal modeling and verification.

Over the past decades, expressive and highly automated security protocol verifiers have been developed, including the two state-of-the-art tools Tamarin [1], [2] and ProVerif [3], which have been used to analyze real-world protocols such as TLS [4], 5G [5], and EMV [6]. These tools build on a model of cryptographic protocols called the *symbolic* or *Dolev-Yao model*, where cryptographic primitives are idealized, protocols are modeled by process algebras or rewriting systems, and the attacker is an abstract entity controlling the network and manipulating messages represented as terms. However, the protocol verified is a highly abstract version of the actual protocol executed and there is *a priori* no formal link between these two versions. The problem of verifying the security of protocol implementations has been studied before, but the solutions usually come with severe limitations, the most important of which we highlight here. Section 7 contains a detailed overview of related work.

Many existing approaches are based on code generation or model extraction, and either extract executable code from a relatively abstract verified model (e.g. [7], [8], [9]) or conversely extract a model from the code for verification (e.g. [10]). Another recent approach, DY⋆ [11], provides a framework to write an executable implementation in F⋆ and obtain a corresponding model that can be reasoned about symbolically, also in F⋆. Other methods adopt an approach, where security is proved in a computational model (e.g. [12], [13], [14]). These models are more precise and thus give stronger guarantees than symbolic ones. However, their proofs are very difficult to automate.

These previous approaches are usually tied to a specific implementation language, like ML, F#, or Java dialects, and they are difficult to extend to other languages. They are therefore ill-suited to verifying pre-existing implementations, especially when used for code extraction. In addition, the extraction mechanisms used are not always proved correct or even formalized, which weakens the guarantees for the resulting code. Moreover, in many cases, the security proof is tailored to the implementation considered rather than constructed at an abstract level by a standard security protocol verifier such as Tamarin or ProVerif. Hence one can neither leverage these tools' automation capabilities nor the substantial prior work invested in security protocol proofs using them.

Our approach. We propose a novel approach to end-to-end verified security protocol implementations. Our approach leverages the combined power of state-of-the-art security protocol verifiers and source code verifiers. This provides abstract, concise, and expressive security protocol specifications on the modeling side and flexibility and versatility on the implementation side.

More precisely, our approach bridges abstract security protocol models expressed in Tamarin as multi-set rewriting

systems with concrete program specifications expressed as I/O specifications (in a dialect of separation logic [15]), against which implementations can be verified. Its technical core is a procedure, implemented in an associated tool, that translates Tamarin models into I/O specifications along with a soundness proof, stating that an implementation satisfying the I/O specifications refines the abstract model in terms of trace inclusion. As a result, any *trace property* proved for the abstract model using Tamarin, including standard security protocol properties such as secrecy and authentication, also holds for the implementation.

Our approach provides a modular and flexible way to verify security protocol implementations. On the model verification side, we can leverage Tamarin's proof automation capabilities to prove protocols secure. Moreover, we can prove a given protocol's security once in Tamarin, and reuse this proof to verify multiple implementations of this protocol, rather than having to produce a custom security proof for each implementation. In fact, numerous complex, real-world protocols have been analyzed using Tamarin over the years. Using our method, this substantial body of prior work can be exploited to verify implementations.

On the code verification side, the I/O specifications we produce can be encoded in many existing verifiers that support separation logic. Our tool currently generates I/O specifications for the Go code verifier Gobra [16] and for the Java code verifier VeriFast [17], which we respectively use for our case study and for our running example. It could easily be extended to other verifiers supporting I/O specifications such as Nagini [18] for Python code. In addition, the requirements for adding other code verifiers based on separation logic to our arsenal are low: they need to only support abstract predicates to encode I/O specifications and to guarantee that successful verification implies trace inclusion between the I/O traces of the program and those of its I/O specification.

We establish our central soundness result relating Tamarin models via I/O specifications to implementations. This result follows a methodology inspired by the Igloo framework [19], which provides a series of generic steps that gradually transform an abstract model into an I/O specification, and requires establishing a refinement relation between each successive pair of steps. We take similar steps and prove these refinements *once and for all* starting from a generic Tamarin system, so that our method can be applied to obtain an I/O specification from any Tamarin protocol model (under some mild syntactic assumptions) without any additional proof.

Our contributions. We summarize our contributions as follows. First, we design a framework for the end-to-end verification of security protocol implementations. This consists of a procedure and an associated tool to extract I/O specifications from a Tamarin model, which can be verified on implementation code. Our soundness result ensures that the implementation inherits all properties proven in Tamarin.

Second, we propose a novel approach to relate the I/O specifications' symbolic terms to the code's bytestring messages. We parameterize the code verifiers' semantics with an abstraction function, instantiate it to a message abstraction function, and identify assumptions and proof obligations to verify that the code correctly implements terms as bytestrings.

Finally, to validate our approach, we perform a substantial case study on a complex, real-world protocol: the WireGuard key exchange, which is part of the widely-used WireGuard VPN in the Linux kernel. We verify a part of the official Go implementation of WireGuard which is interoperable with the full version. Using our method, we thereby obtain an end-to-end symbolically verified WireGuard implementation. All our models, code, and proofs are available online [20], [21].

2. Background

We present background on tools and methodology.

2.1. Tamarin and multiset rewriting

The Tamarin prover [1], [2] is an automatic, state-of-the-art, security protocol verification tool that works in the symbolic model. Tamarin has been used to find weaknesses in and verify improvements to substantial real-world protocols like the 5G AKA protocol [5], [22], TLS 1.3 [23], the Noise framework [24], and EMV payment card protocols [6], [25].

Protocols are represented as *multiset rewriting (MSR) systems*, where each rewrite rule represents a step or action taken by a protocol participant or the attacker. We present the following building blocks: messages, facts, and rules.

Message *terms* are elements of a *term algebra* $\mathcal{T} = \mathcal{T}_\Sigma(\mathcal{N} \cup \mathcal{V})$. These are built over a signature Σ of function symbols and a set of names $\mathcal{N} = \mathit{fresh} \cup \mathit{pub}$ consisting of a set of fresh names fresh (for secret values, generated by parties, unguessable by the attacker), a countably infinite set of public names pub (for globally known values), and a set of variables $\mathcal{V}$. Cryptographic *messages* $\mathcal{M}$ are modeled as ground terms, i.e., terms without variables. The term algebra is equipped with an *equational theory* E, which is a set of equations, and we denote by $=_\mathsf{E}$ the equality modulo E.

Example 1 (Diffie-Hellman equational theory)**.** The signed Diffie-Hellman (DH) protocol is a well-known key exchange protocol, where two agents A and B exchange two DH public keys, g^x and g^y, to establish the shared key g^{xy} (where g is a group generator). For the Tamarin model, we use a term signature containing symbols $\hat{}$, g, sign, verify, pk modeling respectively exponentiation, the group generator, signature, verification, and public keys. We use the simplified theory:

$$(g^x)^y = (g^y)^x \qquad \mathit{verify}(\mathit{sign}(x, k), \mathit{pk}(k)) = \mathit{true}$$

Tamarin's actual model includes further equations. The protocol's informal description is as follows:

$A \to B:$	g^x	x fresh
$B \to A:$	$\mathit{sign}(\langle 0, B, A, g^x, g^y \rangle, k_B)$	y fresh
$A \to B:$	$\mathit{sign}(\langle 1, A, B, g^y, g^x \rangle, k_A)$	agree on $(g^x)^y =_\mathsf{E} (g^y)^x$

The tags 0 and 1 are used to distinguish the last two messages.

All parties, including the attacker, can use the equational theory. The attacker also has its own set of rewriting rules, expressing that it can intercept, modify, block, and recombine

all network messages, following the classic Dolev-Yao (DY) model [26]. These rules are generated automatically from the equational theory, but users may formalize additional rules giving the attacker further scenario-specific capabilities.

Facts are simply atomic predicates applied to message terms, constructed over a signature $\Sigma_{\mathsf{facts}} = \Sigma_{\mathsf{lin}} \uplus \Sigma_{\mathsf{per}}$ of *fact symbols*, partitioned into *linear* (Σ_{lin}), i.e., single-use, and *persistent* (Σ_{per}) facts, which encode the state of agents and the network. We write $\mathcal{F} = \{F(t_1, \ldots, t_k) \mid F \in \Sigma_{\mathsf{facts}} \text{ with arity } k, \text{ and } t_1, \ldots, t_k \in \mathcal{T}\}$ for the set of facts instantiated with terms, partitioned into $\mathcal{F}_{\mathsf{lin}} \uplus \mathcal{F}_{\mathsf{per}}$ as expected. In addition, $\cup^{\mathsf{m}}$, $\cap^{\mathsf{m}}$, $\setminus^{\mathsf{m}}$, $\subseteq^{\mathsf{m}}$, and $\in^{\mathsf{m}}$ denote the usual operations and relations on multisets, and for a multiset m, $\mathrm{set}(m)$ denotes the set of its elements.

A *multiset rewriting (MSR) rule*, written $\ell \xrightarrow{a} r$, contains multisets of *facts* ℓ and r on the left and right-hand side, and is labeled with a, a multiset of actions (also facts, but disjoint from state facts) used for property specification. A MSR system $\mathcal{R}$ is a finite set of rewrite rules. A MSR system $\mathcal{R}$ and an equational theory E have a semantics as a labeled transition system (LTS), whose states are multisets of ground facts from $\mathcal{F}$, the initial state is empty ($[]$), and the transition relation $\Longrightarrow_{\mathcal{R},\mathsf{E}}$ is defined by

$$\frac{\ell \xrightarrow{a} r \in \mathcal{R} \qquad \ell' \xrightarrow{a'} r' =_{\mathsf{E}} (\ell \xrightarrow{a} r)\theta \qquad \ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} S \qquad \mathrm{set}(\ell') \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(S)}{S \overset{a'}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} S \setminus^{\mathsf{m}} (\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'}, \quad (1)$$

where θ is a ground instance of the variables in ℓ, a, and r. Intuitively, the relation describes an update of state S to a successor state, that is possible when a given rule in $\mathcal{R}$ is applicable, i.e., an instantiation with some θ (mod E) of its left-hand side appears in S. Applying the rule consumes the linear but not the persistent facts appearing in its left-hand side, and adds the instantiations under θ of all the facts of its right-hand side to the resulting successor state.

The MSR rules used in Tamarin feature the reserved fact symbols $\mathsf{K} \in \Sigma_{\mathsf{per}}$, and $\mathsf{Fr}, \mathsf{in}, \mathsf{out} \in \Sigma_{\mathsf{lin}}$, modeling respectively the attacker's knowledge, freshness generation, inputs, and outputs. The attacker is modeled by a set of *message deduction rules* MD_Σ, giving it the DY capabilities mentioned above. A distinguished *freshness rule*, labeled $\mathsf{Fr}(n)$, generates fresh values n, which either protocol agents or the attacker directly learn, but never both.

Finally, a protocol's observable behaviors are its *traces*, which are sequences of multisets of actions labeling a sequence of transitions. We define the sets of full traces and of filtered traces with empty labels removed.

$$\mathrm{Tr}(\mathcal{R}) = \{\langle a_i \rangle_{1 \le i \le m} \mid \exists s_1, \ldots, s_m.\ [] \overset{a_1}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} s_1 \overset{a_2}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} \cdots \overset{a_m}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} s_m\}$$
$$\mathrm{Tr}'(\mathcal{R}) = \{\langle a_i \rangle_{1 \le i \le m, a_i \neq []} \mid \langle a_i \rangle_{1 \le i \le m} \in \mathrm{Tr}(\mathcal{R})\}.$$

To ensure that fresh values are unique, we exclude traces with colliding fresh values by defining

$$\mathrm{Tr}_{\mathsf{t}}(\mathcal{R}) = \{\langle a_i \rangle_{1 \le i \le m} \in \mathrm{Tr}'(\mathcal{R}) \mid \forall i, j, n.\ \mathsf{Fr}(n) \in a_i \cap^{\mathsf{m}} a_j \Rightarrow i = j\}.$$

We will abbreviate inclusions between each kind of trace sets using the relation symbols $\preccurlyeq$, $\preccurlyeq'$, and $\preccurlyeq_{\mathsf{t}}$. For example, $\mathcal{R}_1 \preccurlyeq_{\mathsf{t}} \mathcal{R}_2$ denotes $\mathrm{Tr}_{\mathsf{t}}(\mathcal{R}_1) \subseteq \mathrm{Tr}_{\mathsf{t}}(\mathcal{R}_2)$, and similarly for the other two. Note that $\preccurlyeq \subseteq \preccurlyeq' \subseteq \preccurlyeq_{\mathsf{t}}$.

We focus here on Tamarin's *trace properties* (i.e., sets of traces) such as secrecy and authentication [27]. An MSR $\mathcal{R}$ satisfies a trace property Φ, if $\mathrm{Tr}_{\mathsf{t}}(\mathcal{R}) \subseteq \Phi$.

Example 2 (Diffie-Hellman)**.** Continuing Example 1, we use the linear fact symbols $\mathsf{Setup}_{Alice}(\overline{init})$, $Step^1_{Alice}(\overline{init}, x)$, $Step^2_{Alice}(\overline{init}, x, g^y)$ to initialize and record the progress of agent A playing the role of Alice in the protocol. The parameters of the facts store her knowledge. Alice's state is initialized with $\overline{init} = rid, A, k_A, B, pk_B$, i.e., a thread identifier, her identity, her private key, and her partner's identity and public key. This state is then extended with her share of the secret x, and the DH public key g^y she received. For Alice, the two steps of the protocol can then be modeled by the rules:

$$[\mathsf{Setup}_{Alice}(\overline{init}), \mathsf{Fr}(x)] \xrightarrow{[]} [Step^1_{Alice}(\overline{init}, x), \mathsf{out}(g^x)]$$

$$[Step^1_{Alice}(\overline{init}, x), \mathsf{in}(sign(\langle 0, B, A, g^x, Y\rangle, k_B))] \xrightarrow{[Secret(Y^x)]} [Step^2_{Alice}(\overline{init}, x, Y), \mathsf{out}(sign(\langle 1, A, B, Y, g^x\rangle, k_A))]$$

The received signature is checked using pattern-matching, knowing that $pk_B = pk(k_B)$. The action fact $Secret(Y^x)$ in the second rule is used to specify key secrecy. It records Alice's belief that the key she computes from the value Y (supposedly g^y) received from Bob remains secret. The fact $\mathsf{Setup}_{Alice}(\overline{init})$ in the first rule is produced by another rule, modeling the environment initializing Alice's knowledge:

$$[\mathsf{Fr}(rid), \mathsf{sk}(A, k_A), \mathsf{pk}(B, pk_B)] \xrightarrow{[]} [\mathsf{Setup}_{Alice}(rid, A, k_A, B, pk_B)].$$

2.2. Separation logic and I/O specifications

Separation logic enables sound and modular reasoning about heap manipulating programs by associating every allocated heap location with a *permission*. Permissions are a static concept used to verify programs, but do not affect their runtime behavior. Each permission is held by at most one function execution at each point in the program execution. To access a heap location, a function must hold the associated permission; otherwise, a verification error occurs. The *separating conjunction* $\star$ sums up the permissions in its conjuncts. Permissions to an unbounded set of locations, for instance, all locations of a linked list, can be expressed via co-recursive predicates. Moreover, *abstract* predicates are useful to specify permissions to an unknown set of locations.

Permission-based reasoning generalizes from heap locations to other kinds of program resources. Penninckx et al. [28] reason about a program's I/O behavior by associating each I/O operation with a permission that is required to call the operation and is then consumed. They equip the main function's precondition with an *I/O specification* that grants all permissions necessary to perform the desired I/O operations of the entire program execution. These I/O specifications can easily be encoded in standard separation

```
requires token(?p1) && out(p1,v,?p2)
ensures  ok ⟹ token(p2)
ensures !ok ⟹ token(p1) && out(p1,v,p2)
func send(v int) (ok bool)
```

Figure 1. Specification of the `send` operation with I/O permissions. Variables starting with `?` are implicitly existentially quantified. The code verifier uses `&&` to denote the separating conjunction $\star$.

logic, such that existing program verifiers supporting different programming languages can be used to verify I/O behavior.

Every I/O operation `io`, such as sending or receiving a value, is associated with an abstract predicate **io**, called an *I/O permission*. Intuitively, $\mathbf{io}(p_1, \bar{v}, \bar{w}, p_2)$ expresses the permission to perform `io` with outputs $\bar{v}$ and inputs $\bar{w}$. We use $\bar{x}$ to denote a vector of zero or more values. The parameters p_1 and p_2 are called *source and target places*, respectively. An abstract predicate $token(p)$ is called a *token* at place p. The I/O operation `io` moves a token from the source place p_1 to the target place p_2 by consuming $token(p_1)$ and producing $token(p_2)$. Hence, the position of the token indicates the currently allowed I/O operations.

Example 3 (Send I/O operation)**.** Figure 1 shows the signature and specification of a `send` function. The precondition requires an I/O permission **out** to send the value `v` at some source place p_1 with the corresponding token. When the send operation succeeds, the I/O permission is consumed and the token is moved to some target place p_2. In case of failure, the token remains at the source place and the I/O permission is not consumed.

The separating conjunction of I/O permissions with the same source place encodes non-deterministic choice between such permissions. Moreover, co-recursion enables repeated as well as non-terminating sequences of I/O operations.

Example 4 (I/O specification for a server)**.**

$$\begin{aligned} P(p, S) &= Q(p, S) \star R(p, S) \\ Q(p_1, S) &= \exists v, p_2, p_3.\, \mathbf{in}(p_1, v, p_2) \star \mathbf{out}(p_2, v, p_3) \\ &\quad \star P(p_3, S \cup \{v\}) \\ R(p_1, S) &= \exists p_2.\, \mathbf{out}(p_1, \text{“Ping”}, p_2) \star P(p_2, S) \end{aligned}$$

The formula $\phi = token(p) \star P(p, \emptyset)$ specifies a non-terminating server that repeatedly and non-deterministically chooses between receiving and forwarding a value v or sending a “Ping” message. All received values v are recorded in the state S, which is initially empty. Input parameters, like v in **in** here, are existentially quantified to avoid imposing restrictions on the values received from the environment.

To enforce certain state updates between I/O operations, it is useful to associate permissions also with certain internal (that is, non-I/O) operations and include those *internal permissions* in an I/O specification. For instance, the above server could include an internal operation to reset the state S when it exceeds a certain size.

I/O specifications induce a transition system and hence have a trace semantics. The traces can intuitively be seen as the sequences of I/O permissions consumed by possible executions of the programs that satisfy it. We write $\mathrm{Tr}(\phi)$ for the set of traces of an I/O specification ϕ. In the example above, the sequence $\mathbf{in}(5) \cdot \mathbf{out}(5) \cdot \mathbf{out}(\text{“Ping”}) \cdot \mathbf{in}(7) \cdot \mathbf{out}(7)$ is one example of a trace of ϕ. Note that the I/O permissions' place arguments do not appear in the trace.

3. From Tamarin models to I/O specifications

In this section, we present our transformation of a Tamarin protocol model, given as an MSR system $\mathcal{R}$, into a set of I/O specifications ψ_i, one for each protocol role i. The ψ_i serve as program specifications, against which the roles' implementations c_i are verified (Section 4). Our main result is an overall soundness guarantee stating that the traces of the complete system $C(c_1, \ldots, c_n, \mathscr{E})$, composed of the roles' verified implementations c_i and the environment $\mathscr{E}$, are contained in the traces of the protocol model $\mathcal{R}$ (Section 5):

$$C(c_1, \ldots, c_n, \mathscr{E}) \preccurlyeq_{\mathrm{t}} \mathcal{R}.$$

Hence, any trace property Φ proven for the protocol model, i.e., $\mathrm{Tr}_{\mathrm{t}}(\mathcal{R}) \subseteq \Phi$, is inherited by the implementation.

The sound transformation of an MSR protocol model $\mathcal{R}$ into a set of I/O specifications is challenging:

1) Tamarin's MSR formalism is very general and offers great flexibility in modeling protocols and their properties. We want to preserve this generality as much as possible.
2) For the transformation to I/O specifications, we require a separate description of each protocol role and of the environment, with a clear interface between the two parts. This interface will be mapped to I/O permissions in the I/O specification and eventually to (e.g., I/O or cryptographic) library calls in the implementation.
3) The protocol models operate on abstract terms, whereas the implementation manipulates bytestrings. We need to bridge this gap in a sound manner.

Our solution is based on a general encoding of the MSR semantics into I/O specifications. To separate the different roles' rewrite rules from each other and from the environment, we partition the fact symbols and rewrite rules accordingly. The interface between the roles and the environment is defined by identifying I/O fact symbols, for which we introduce separate I/O rules. This isolates the I/O operations from others and allows us to map them to I/O permissions and later to library functions. Moreover, we keep I/O specifications as abstract as possible by using message terms rather than bytestrings. We handle the transition to bytestrings in the code verification process (Section 4).

The proofs for the results stated in this section can be found in the full version of this paper [20].

3.1. Protocol format

We introduce a few mild formatting assumptions on the Tamarin model. They mostly correspond to common modeling practice and serve to cleanly separate the different protocol roles and the environment. They do not restrict

Tamarin's expressiveness for modeling protocols. In particular, all protocols in the Tamarin distribution would fit our assumptions after some minor modifications.

3.1.1. Rule format. To model an n-role protocol, we use a fact signature of the form

$$\Sigma_{\mathsf{facts}} = \Sigma_{\mathsf{act}} \uplus \Sigma_{\mathsf{env}} \uplus \Big(\biguplus_{1 \le i \le n} \Sigma^i_{\mathsf{state}} \Big),$$

where (i) Σ_{act}, Σ_{env}, and $\Sigma^i_{\mathsf{state}}$ are mutually disjoint sets of fact symbols, used to construct action facts (used in transition labels), environment facts, and each role i's state facts; (ii) Σ_{env} contains two disjoint subsets, Σ_{in} and Σ_{out}, of input and output fact symbols such that $\mathsf{Fr}, \mathsf{in} \in \Sigma_{\mathsf{in}}$, $\mathsf{out} \in \Sigma_{\mathsf{out}}$, and $\mathsf{K} \in \Sigma_{\mathsf{env}} \setminus (\Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}})$; and (iii) there is an initialization fact symbol $\mathsf{Setup}_i \in \Sigma_{\mathsf{in}}$ for each protocol role i.

We consider MSR systems $\mathcal{R}$ whose rules are given by:

$$\mathcal{R} = \mathcal{R}_{\mathsf{env}} \uplus \Big(\biguplus_{1 \le i \le n} \mathcal{R}_i \Big).$$

Here, $\mathcal{R}_{\mathsf{env}}$ and the $\mathcal{R}_i$s are pairwise disjoint rule sets containing rules for the environment and each protocol role. Protocol rules use input and output facts to communicate with the environment. For example, the following two environment rules transfer a message to and from the attacker's knowledge:

$$[\mathsf{out}(x)] \xrightarrow{[]} [\mathsf{K}(x)] \qquad [\mathsf{K}(x)] \xrightarrow{[]} [\mathsf{in}(x)]. \tag{2}$$

The attacker rules MD_Σ, the freshness rule, and the rules that generate the Setup_i facts (cf. Example 2), are also in $\mathcal{R}_{\mathsf{env}}$. The protocol rules for role i (and only those) use role state facts from $\Sigma^i_{\mathsf{state}}$ to keep track of the role's progress. We require that their first $k_i \ge 1$ arguments are reserved for role i's parameters from the Setup_i fact, and that the first of these parameters is the thread identifier rid.

For a more detailed specification of the format restrictions on the rewrite rules, see Appendix A.

3.1.2. Protocol messages. We support both the usual ways of checking received messages using pattern matching and explicit equality checks. The latter are formalized, as usual in Tamarin, as a combination of action facts labeling the given rule (e.g., $\mathsf{Eq}(x, \mathsf{hash}(z))$) and restrictions associating these facts with (a boolean combination of) equalities (e.g., $x =_{\mathsf{E}} y$ whenever $\mathsf{Eq}(x, y)$ occurs in a trace). These restrictions act as assumptions on the traces considered by Tamarin.[1] The I/O specifications resulting from our procedure require that these equality checks are implemented (Section 3.3).

Furthermore, we recommend, but do not require, the replacement of nested pairs and tuples by *formats* [29]. These are user-defined function symbols, along with projections for all arguments, that behave like tuples. In the implementation, each format is mapped to a combination of tags (i.e., constant bytestrings), fixed-size fields, and variable-sized fields prepended with a length field. Formats help to soundly relate term and bytestring messages, if we prove that they are unambiguous and non-overlapping (see Section 4).

Example 5 (Diffie-Hellman formatting)**.** The rules for Alice's role from Example 2 satisfy the format conditions above. The initiator setup rule produces a fact $\mathsf{Setup}_{Alice}(\overline{init})$, whose parameters $\overline{init}$ appear as the first parameters of the state facts $Step^1_{Alice}(\overline{init}, \ldots)$ and $Step^2_{Alice}(\overline{init}, \ldots)$. Both protocol rules produce an out fact to send a message. The second rule also consumes an in fact to receive a message. To follow our recommendation to use formats, we can model the message $\langle 0, B, A, g^x, Y \rangle$ being signed as a format with five fields, rather than a tuple. Note that, at the Tamarin level, tuples containing unique tags to distinguish them behave essentially the same as formats.

3.2. Transformation to component models

We decompose an MSR system $\mathcal{R}$ that satisfies our format requirements into several component models, one for each role, and a separate environment model, which includes the attacker. In doing so, we move from a global view of the protocol, useful for security analysis, to a local view of each role, more appropriate for their implementation. In Section 3.3, we transform the component models into I/O specifications for the programs implementing them.

As a preparatory step, we refine $\mathcal{R}$ into an *interface model* which starts decoupling the roles from the environment by introducing separate rewrite rules for their interactions.

3.2.1. Interface model. The protocol roles and the environment interact using input and output facts, including the built-in facts Fr, in, and out. For example, the protocol roles receive messages by consuming in facts produced by the attacker. The interface model adds an *I/O rule* for each such fact, which turns it into a buffered version. These I/O rules will later be implemented as calls to library functions.

Let Σ^-_{in} be the set Σ_{in} without the initialization facts Setup_i. We first add to the fact signature, for each non-setup input or output fact F and role i, a copy (the "buffer") F_i:

$$\Sigma^i_{\mathsf{buf}} = \{F_i \mid F \in \Sigma^-_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}\} \qquad \Sigma^i_{\mathsf{role}} = \Sigma^i_{\mathsf{state}} \cup \Sigma^i_{\mathsf{buf}}$$
$$\Sigma'_{\mathsf{facts}} = \Sigma_{\mathsf{act}} \uplus \Sigma_{\mathsf{env}} \uplus (\textstyle\biguplus_i \Sigma^i_{\mathsf{role}}).$$

We then replace the facts used by the protocol rules as follows. For each role i, let $\mathcal{R}'_i$ be the set of rules obtained by replacing, in all rules in $\mathcal{R}_i$, each fact $F(t_1, \ldots, t_k)$ such that $F \in \Sigma^-_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}$ by $F_i(rid, t_1, \ldots, t_k)$. The latter fact has rid as an additional parameter.

We also introduce the set $\mathcal{R}_{\mathsf{io}}$ of *I/O rules*, which translate between input or output facts and their buffered versions. The set $\mathcal{R}_{\mathsf{io}}$ contains the following rules, for each role i.

$$[F(x_1, \ldots, x_k)] \xrightarrow{[]} [F_i(rid, x_1, \ldots, x_k)] \quad \text{for } F \in \Sigma^-_{\mathsf{in}}$$
$$[G_i(rid, x_1, \ldots, x_k)] \xrightarrow{[]} [G(x_1, \ldots, x_k)] \quad \text{for } G \in \Sigma_{\mathsf{out}}.$$

1. To handle these, we use a modified, but equivalent MSR semantics, where the equalities are checked at each step rather than globally on traces. This semantics allows us to translate these checks into the I/O specifications and thus enforce their correct implementation. For simplicity, we present our results under the standard semantics and refer to [20] for more details.

For reasons that will become clear later, we also count the role setup rules as I/O rules. Hence, we move them from $\mathcal{R}_{\mathsf{env}}$ to $\mathcal{R}_{\mathsf{io}}$, calling the remaining environment rules $\mathcal{R}^-_{\mathsf{env}}$.

Finally, the interface model is specified by:

$$\mathcal{R}_{\mathsf{intf}} = \mathcal{R}^-_{\mathsf{env}} \uplus \mathcal{R}_{\mathsf{io}} \uplus (\biguplus_i \mathcal{R}'_i). \tag{3}$$

Example 6. Continuing Example 5, we introduce the buffer facts in_{Alice}, out_{Alice}, and Fr_{Alice}. Recall that rid is included in $\overline{init}$. This yields the modified set $\mathcal{R}'_{Alice}$ for the role $Alice$:

$$[\mathsf{Setup}_{Alice}(\overline{init}), \mathsf{Fr}_{Alice}(rid, x)] \xrightarrow{[]} [Step^1_{Alice}(\overline{init}, x), \mathsf{out}_{Alice}(rid, g^x)]$$

$$[Step^1_{Alice}(\overline{init}, x), \mathsf{in}_{Alice}(rid, sign(\langle 0, B, A, g^x, Y\rangle, k_B))] \xrightarrow{[Secret(Y^x)]} [Step^2_{Alice}(\overline{init}, x, Y), \mathsf{out}_{Alice}(rid, sign(\langle 1, A, B, Y, g^x\rangle, k_A))].$$

We show that the interface model refines the original one.

Lemma 1. $\mathcal{R}_{\mathsf{intf}} \preccurlyeq' \mathcal{R}$.

3.2.2. Decomposition. We can now decompose the interface model into the role components and the environment. In a nutshell, we assign the rules $\mathcal{R}'_i$ to the component for role i and the rules $\mathcal{R}^-_{\mathsf{env}}$, including the attacker rules MD_Σ, to the environment. The protocol communicates with the environment using the I/O rules. We split them into two synchronized parts, one belonging to the environment and the other to the protocol. Below, we show that the re-composed system implements the interface model.

We explain the splitting of the I/O rules into a protocol part and an environment part using the example rule

$$[\mathsf{out}_i(rid, x)] \xrightarrow{[]} [\mathsf{out}(x)].$$

This rule models instance rid of role i outputting a message to the attacker. We split this rule into two parts:

$$[\mathsf{out}_i(rid, x)] \xrightarrow{[\lambda_{\mathsf{out}}(rid,x)]} [], \tag{4}$$

$$[] \xrightarrow{[\lambda_{\mathsf{out}}(rid,x)]} [\mathsf{out}(x)], \tag{5}$$

where the first rule belongs to role i and the second to the environment. We label both rules with a new action fact $\lambda_{\mathsf{out}}(rid, x)$, which uniquely identifies the original I/O rule and has as parameters all variables occurring in it. We call this fact a *synchronization label*, as we later use it for synchronizing the two parts to recover the original rule's behavior. We similarly split all rules in $\mathcal{R}_{\mathsf{io}}$, yielding two sets $\mathcal{R}^i_{\mathsf{io}}$ and $\mathcal{R}^e_{\mathsf{io}}$ belonging to the protocol role i and to the environment.

The components for each protocol role i and for the environment are then defined as follows.

$$\mathcal{R}^i_{\mathsf{role}} = \mathcal{R}'_i \uplus \mathcal{R}^i_{\mathsf{io}} \qquad \mathcal{R}^e_{\mathsf{env}} = \mathcal{R}^-_{\mathsf{env}} \uplus \mathcal{R}^e_{\mathsf{io}}. \tag{6}$$

Note that these two rule sets operate on pairwise disjoint sets of facts, namely over the signatures Σ^i_{role} and Σ_{env}, respectively. Hence they can interact with each other only by synchronizing the split I/O rules.

Example 7 (Component for Diffie-Hellman)**.** The MSR system $\mathcal{R}^{Alice}_{\mathsf{role}}$ for Alice's role contains the two protocol rules in $\mathcal{R}'_{Alice}$ from Example 6, the output rule (4) described above, and similar rules for inputs and freshness generation. In addition, it contains the protocol part of the split setup rule for Alice's role from Example 2, i.e.

$$[] \xrightarrow{[\lambda_{Alice}(rid,A,k_A,B,pk_B)]} [\mathsf{Setup}_{Alice}(rid, A, k_A, B, pk_B)].$$

The traces of the recomposition of all roles with the environment are included in the traces of the interface model. We define two kinds of parallel compositions on LTSs (induced here by the MSR systems' transition semantics). We give the intuition here and refer the reader to Appendix B for the formal definitions. The (indexed) parallel composition $|||$ interleaves the transitions of a family of component systems without communication. The (binary) parallel composition $\|_\Lambda$ synchronizes transitions with labels from the set Λ, resulting in a transition labeled $[]$, and interleaves all other transitions. We then show:

Lemma 2 (Decomposition)**.** *Let $\mathcal{R}^i_{\mathsf{role}}(rid)$ be the MSR system $\mathcal{R}^i_{\mathsf{role}}$ for a fixed thread id rid. Then*

$$(|||_{i,rid} \mathcal{R}^i_{\mathsf{role}}(rid)) \;\|_\Lambda\; \mathcal{R}^e_{\mathsf{env}} \preccurlyeq \mathcal{R}_{\mathsf{intf}},$$

where $\Lambda = \bigcup_i \{a\theta \mid \exists \ell, r.\, \ell \xrightarrow{a}_{\mathsf{io}} r \in \mathcal{R}^i_{\mathsf{io}} \wedge range(\theta) \subseteq \mathcal{M}\}$ consists of all ground instances of synchronization labels.

3.3. Transformation to I/O specifications

Finally, we extract an I/O specification ψ_i from each role i's MSR system $\mathcal{R}^i_{\mathsf{role}}$, which serves as the specification for the role's implementation at the code level. ψ_i is parameterized by the thread identifier rid, and it associates a token with the starting place p of the predicate P_i:

$$\psi_i(rid) = \exists p.\; token(p) \star P_i(p, rid, []).$$

The predicate $P_i(p, rid, S)$'s parameters are a place p, a thread identifier rid, and a state S of the MSR system $\mathcal{R}^i_{\mathsf{role}}$ (i.e., a multiset of ground facts). Note that ψ_i invokes P_i with the initial state, i.e., the empty multiset $[]$ (see Section 2.1). It is defined co-recursively as the separating conjunction over the formulas ϕ_R, one for each rewrite rule $R \in \mathcal{R}^i_{\mathsf{role}}$:

$$P_i(p, rid, S) = \bigstar_{R \in \mathcal{R}^i_{\mathsf{role}}} \phi_R(p, rid, S).$$

ϕ_R encodes an application of the rewrite rule R to the model state S. It contains an I/O or internal permission $\mathbf{R}(p, \ldots, p')$, which an implementation must hold in order to execute the program part implementing R. ϕ_R co-recursively calls $P_i(p', rid, S')$ with the target place p' of $\mathbf{R}$ and the updated state S'. We define the formulas ϕ_R separately for protocol rules in $\mathcal{R}'_i$ and for I/O rules in $\mathcal{R}^i_{\mathsf{io}}$.

Consider a protocol rule $R = \ell \xrightarrow{a} r \in \mathcal{R}'_i$ with variables $\overline{x}$. We associate the internal permission $\mathbf{R}(p, \overline{x}, \ell', a', r', p')$ to R, and define $\phi_R(p, rid, S)$ by

$$\begin{aligned}&\phi_R(p, rid, S) = \forall \overline{x}, \ell', a', r'.\\ &\quad M(\ell', s) \wedge \ell' =_{\mathsf{E}} \ell \wedge a' =_{\mathsf{E}} a \wedge r' =_{\mathsf{E}} r \wedge \Phi_R(\overline{x})\\ &\quad \Longrightarrow \exists p'.\; \mathbf{R}(p, \overline{x}, \ell', a', r', p') \star P_i(p', rid, U(\ell', r', S))\end{aligned}$$

where $M(\ell', S) = (\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} S) \wedge (\mathrm{set}(\ell') \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(S))$, $U(\ell', r', S) = S \setminus^{\mathsf{m}} (\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'$, and Φ_R is the

conjunction of all (boolean combinations of) equality checks (mod E) that the rule R performs using a combination of action facts in $\mathscr{a}$ and associated restrictions (cf. Section 3.1.2).

This formula specifies that, for any instantiation (mod E) $\ell', \mathscr{a}', \mathscr{r}'$ of the facts in the rule, if the matching condition $M(\ell', S)$ and the equational formula Φ_R are satisfied, then we have an internal permission $\mathbf{R}$ to execute the rule's implementation. This yields an updated state $U(\ell', \mathscr{r}', S)$, on which P_i is co-recursively applied to produce the permissions for the rest of the execution. The formula Φ_R thus enforces that the implementation performs the explicit equality checks on messages specified in the rule R (see also Section 4.3.4).

We define similar formulas for all I/O rules. For output rules of the form $R_G = [G_i(rid, \overline{x})] \xrightarrow{[\lambda_G(rid,\overline{x})]} [] \in \mathcal{R}^i_{\mathsf{io}}$, we define the formula

$$\phi_{R_G}(p, rid, S) = \forall \overline{x}.\ G_i(rid, \overline{x}) \in^{\mathsf{m}} S \\ \implies \exists p'.\ \mathbf{R_G}(p, rid, \overline{x}, p') \star P_i(p', rid, S \setminus^{\mathsf{m}} [G_i(rid, \overline{x})])),$$

which grants the I/O permission $\mathbf{R_G}(p, rid, \overline{x}, p')$ for any rid and terms $\overline{x}$ for which the fact $G_i(rid, \overline{x})$ exists in S. P_i is called co-recursively with the target place of the permission and the updated state, where $G_i(rid, \overline{x})$ is removed.

For input rules $R_F = [] \xrightarrow{[\lambda_F(rid,\overline{z})]} [F_i(rid, \overline{z})] \in \mathcal{R}^i_{\mathsf{io}}$, we define the formula

$$\phi_{R_F}(p, rid, S) = \\ \exists p', \overline{z}.\ \mathbf{R_F}(p, rid, \overline{z}, p') \star P_i(p', rid, S \cup^{\mathsf{m}} [F_i(rid, \overline{z})])),$$

which grants the I/O permission $\mathbf{R_F}(p, rid, \overline{z}, p')$ to read inputs $\overline{z}$ for any rid. Note that the input variables $\overline{z}$ are existentially quantified (cf. Section 2.2) and the fact $F_i(rid, \overline{z})$ is added to the state in the co-recursive call to P_i.

Example 8 (I/O specification for Diffie-Hellman)**.** Continuing Examples 6 and 7, the component system is translated into an I/O specification that features the following conjunct corresponding to the rule for the second step of Alice's role:

$$\phi_{Alice_2}(p, rid, S) = \forall \overline{init}, x, Y, \ell', \mathscr{a}', \mathscr{r}'. \\ M(\ell', S) \wedge \\ \ell' =_{\mathsf{E}} \{\!|Step^1_{Alice}(\overline{init}, x), \mathsf{in}_{Alice}(rid, sign(\langle 0, B, A, g^x, Y\rangle, k_B))|\!\} \wedge \\ \mathscr{a}' =_{\mathsf{E}} \{\!|Secret(Y^x)|\!\} \wedge \\ \mathscr{r}' =_{\mathsf{E}} \{\!|Step^2_{Alice}(\overline{init}, x, Y), \mathsf{out}_{Alice}(rid, sign(\langle 1, A, B, Y, g^x\rangle, k_A))|\!\} \wedge \\ \implies \exists p'.\ \mathbf{Alice_2}(p, \overline{init}, x, Y, \ell', \mathscr{a}', \mathscr{r}', p') \star P_i(p', rid, U(\ell', \mathscr{r}', S)).$$

Namely, the permission to execute this step is granted, provided S contains instantiations of the previous state fact and the correct input fact, and that S is updated by replacing them with the new state and output facts. Recall that $\overline{init}$ abbreviates rid, A, k_A, B, pk_B.

The construction of ψ_i from $\mathcal{R}^i_{\mathsf{role}}$ can be seen as an instance of the method presented in [19] and by the soundness result from that paper, we get the following trace inclusion.

Theorem 1 ([19])**.** *For all MSR systems $\mathcal{R}^i_{\mathsf{role}}(rid)$, where the fresh name rid instantiates the thread id in all facts,*

$$\pi(\psi_i(rid)) \preccurlyeq \mathcal{R}^i_{\mathsf{role}}(rid),$$

where $\pi = \pi_{int} \circ \pi_{ext}$ relabels (the LTS induced by) $\psi_i(rid)$. Here, π_{int} and π_{ext} are the identity functions except on the following labels:

$$\pi_{int}(\mathbf{R}(\overline{x}, \ell', \mathscr{a}', \mathscr{r}')) = \mathscr{a}' \qquad \text{for } R \in \mathcal{R}'_i \\ \pi_{ext}(\mathbf{F}(rid, \overline{x})) = [\lambda_F(rid, \overline{x})] \qquad \text{for } F \in \mathcal{R}^i_{\mathsf{io}}.$$

4. Verified protocol implementations

The implementation step consists of providing the code $c_i(rid)$ implementing each role i and proving that it satisfies its I/O specification $\psi_i(rid)$. The challenge here is bridging the abstraction gap between the message terms in the I/O specifications $\psi_i(rid)$ and the bytestring messages manipulated by the code. In Section 4.1, we present an extension of the code verifiers' semantics of Hoare triples to accommodate such abstractions. In Section 4.2, we explain how we concretely relate bytestrings to terms. Finally, in Section 4.3, we show how we verify the roles' I/O specifications based on appropriate I/O and cryptographic library specifications.

4.1. Code verification with abstraction

In Penninckx et al.'s program logic [28], the statement that a program c satisfies an I/O specification ϕ is expressed as the Hoare triple

$$\{\phi\}\ c\ \{\mathsf{true}\}, \tag{7}$$

with the I/O specification ϕ in the precondition and the postcondition true. We assume that the program c has an LTS semantics $\mathscr{C}$ given by the programming language's operational semantics, where the labels represent the program's I/O (and internal) operations and the program's traces consist of sequences of such labels. We leave the exact semantics unspecified here, to keep our formulation generic with respect to the programming language used. The semantics of the Hoare triple (7) implies that the program c's traces are included in the traces of ϕ, i.e. $\mathscr{C} \preccurlyeq \phi$.

To bridge the gap between message terms and bytestring messages, we extend Penninckx et al.'s approach by introducing an *abstraction* or relabeling function α between the implementation's transition labels and the I/O specification's transition labels. For example, α may map a concrete label $\mathbf{in}_c(l)$ to an abstract version $\mathbf{in}_a(s)$, where l is a list implementation and s is the mathematical set of l's elements. We also extend the soundness assumption on the code verifier accordingly.

Assumption 1 (Verifier assumption)**.**

$$\vdash_\alpha \{\phi\}\ c\ \{\mathsf{true}\} \implies \alpha(\mathscr{C}) \preccurlyeq \phi.$$

This means that a successful verification implies that the program traces, abstracted under α, are included in the I/O specification ϕ's traces. In Appendix C, we sketch a semantics for Hoare triples that entails this trace inclusion.

To ensure this extension's soundness, we require that the I/O operations' contracts are *consistent with* α, i.e., they imply a correct mapping of transition labels under α. More

precisely, suppose the specification of such an operation op induces a concrete transition label $\mathbf{op}_c(\overline{a})$, where $\overline{a}$ are op's inputs and outputs, and the I/O permission in the precondition induces the abstract transition label $\mathbf{op}_a(\overline{b})$. Then we define α as lifting from an (overloaded) function α that maps concrete parameter types to abstract ones, i.e., $\alpha(\mathbf{op}_c(\overline{a})) = \mathbf{op}_a(\alpha(\overline{a}))$. We therefore require that $\overline{b} = \alpha(\overline{a})$ follows from op's precondition (for arguments) and postcondition (for return values). Moreover, we allow α to be a partial function, in which case the specification must also imply that the concrete arguments are in its domain.

Application to role verification. We now apply this idea to the verification of the protocol's role implementations (using Gobra in our case study). That is, we wish to establish

$$\vdash_\alpha \{\psi_i(rid)\}\ c_i(rid)\ \{\mathsf{true}\} \tag{8}$$

for a suitable α. An obvious possibility would be to define an abstraction function $\alpha : \mathbb{B}^\star \to \mathcal{M}$ from bytestrings to messages and then lift it to trace labels. For example, a concrete $\mathbf{in}_c(rid, b)$ would be abstracted to $\alpha(\mathbf{in}_c(rid, b)) = \mathbf{in}_a(rid, \alpha(b))$. However, this mapping assumes that each bytestring corresponds to exactly one term, and consequently that *every* bytestring can be uniquely parsed as a term. To minimize our assumptions, however, we do not a priori want to exclude collisions between bytestrings, i.e., we allow a bytestring to have several term interpretations.

In Section 4.2, we therefore relate bytestrings and terms using a concretization function $\gamma : \mathcal{M} \to \mathbb{B}^\star$. Since a bytestring may be related to several terms, we cannot define a function α mapping concrete labels to abstract I/O labels. Our solution is based on adding ghost term parameters to the I/O operations in the implementation code. For example, the operation receiving a bytestring b gets an additional ghost return value term m with $b = \gamma(m)$ and the corresponding transition label is $\mathbf{in}_c(rid, (b, m))$. These ghost terms aid verification (see Section 4.3), but are not present in the executable code. We instantiate α to the function π'_{ext} that removes the bytestrings from the concrete I/O operation's labels and keeps only the ghost terms used for the reasoning. For instance, $\pi'_{ext}(\mathbf{in}_c(rid, (b, m))) = \mathbf{in}_a(rid, m)$. This function is defined only for $b = \gamma(m)$, which is guaranteed by the receive operation's contract (cf. Figure 2).

Our proposed method enables us to verify that pre-existing real-world code satisfies I/O specifications produced from abstract Tamarin models (see Section 6).

4.2. Relating terms and bytestrings

In Tamarin's MSR semantics, messages in $\mathcal{M}$ are ground terms. We model the concrete messages and the operations on them as *bytestring algebras* defined as Σ-algebras $\mathcal{B}$ with the set of bytestrings $\mathbb{B}^\star$ as the carrier set. To relate terms to bytestrings, we use a surjective Σ-algebra homomorphism γ: $\mathcal{M} \to \mathcal{B}$, which maps (fresh and public) names to bytestrings and the signature's symbols to functions on bytestrings:

$$\begin{aligned} \gamma(n) &= n^{\mathcal{B}} && \text{for } n \in \mathcal{N} \\ \gamma(f(t_1, \ldots, t_k)) &= f^{\mathcal{B}}(\gamma(t_1), \ldots, \gamma(t_k)) && \text{for } f \in \Sigma^k \end{aligned}$$

```
ensures seq(ciph) = enc^B(seq(key), seq(msg))
func encrypt(key, msg []byte) (ciph []byte)

ensures ok ⟹ seq(c) = enc^B(seq(k), seq(m))
func decrypt(k, c []byte) (m []byte, ok bool)

requires token(?p1) && in(p1,?m,?p2)
ensures  ok ⟹ token(p2) && seq(b) = γ(m)
ensures !ok ⟹ token(p1) && in(p1,m,p2)
func receive() (b []byte,ghost m term,ok bool)
```

Figure 2. Simplified specifications for encryption, decryption, and receive. The function seq abstracts an in-memory byte array to $\mathcal{B}$. We omit Gobra's memory annotations needed to reason about heap data structures and conditions on the size of bytestrings.

With the requirement that γ is surjective, we avoid junk bytestrings that do not represent any term (i.e., the algebra $\mathcal{B}$ is term-generated). This is without loss of generality as there are countably infinitely many public names that can be mapped to potential junk bytestrings.

Note that Σ-algebra homomorphisms are required to preserve equalities. For example, a symbolic equality $\mathsf{dec}(k, \mathsf{enc}(k, m)) =_{\mathsf{E}} m$ on terms implies the equality

$$\mathsf{dec}^{\mathcal{B}}(key, \mathsf{enc}^{\mathcal{B}}(key, msg)) = msg$$

on bytestrings. In what follows, we will use the bytestring algebra's functions in our cryptographic library's specification. This enables us to reason about message parsing and construction.

4.3. Verifying the I/O specification

The verification of the I/O specification generally follows the same approach as in previous work [28], [19]. Every I/O operation performed by the code requires that a corresponding I/O permission is held. The required I/O permissions must be obtained from the I/O specification. However, our introduction of abstraction makes reasoning about what is sent and, in particular, received more challenging.

4.3.1. Sending and receiving messages. For a sent pair of a bytestring and a ghost message (in $\mathcal{M}$), we must verify that the I/O specification permits sending the message. Similarly, for a received pair of a bytestring and a ghost message, we must verify that the received message matches a term in the I/O specification, describing the expected protocol message. We refer to such terms as *patterns*. In Example 8, there is a single pattern, namely $sign(\langle 0, B, A, g^x, Y\rangle, k_B)$, where the unconstrained Y is a variable and all other entities are constrained by the fact $Step^1_{Alice}(\overline{init}, x)$.

Verifying that the I/O specification permits sending a message boils down to verifying that the bytestring $\gamma(m)$ for a permitted message m was constructed and then sent. This becomes straightforward by equipping the cryptographic library with suitable specifications. Consider the simplified specification of an encryption function shown in Figure 2. The function seq abstracts an in-memory byte array into a mathematical sequence of bytes, i.e., an element of $\mathbb{B}^\star$. Due to the specification and the surjectivity

```
1 // seq(key) = γ(k) holds
2 ciph, c, ok := receive(); if !ok {return}
3 assert seq(ciph) = γ(c)
4 msg, ok := decrypt(key, ciph); if !ok {return}
5 assert ∃u. seq(msg) = γ(u)
6          && seq(ciph) = γ(enc(k, u))
7 PaR1(m, ...) // using the pattern requirement
8 assert ∃w. c =E enc(k, w) && seq(msg) = γ(w)
```

Figure 3. Reasoning about receiving and parsing a ciphertext.

```
req token(p) && P_Ann(p,r,S) && Step^1_Ann(k)∈mS
req ∃x. γ(enc(k,x)) = γ(m)
ens token(p) && P_Ann(p,r,S) && ∃x'. m =E enc(k,x')
ghost func PaR1(m,p,r,S,k)
```

Figure 4. Ghost function for the pattern requirement of Example 9. There is the single pattern $\mathsf{enc}(k, x)$, where k is a constant and x is a variable.

of γ, the result of `encrypt(key, msg)` is equal to $\gamma(\mathsf{enc}(m_{key}, m_{msg}))$ for some messages m_{key} and m_{msg}, where $\gamma(m_{key}) = \texttt{seq(key)}$ and $\gamma(m_{msg}) = \texttt{seq(msg)}$. To verify the construction of an entire message, we combine the information of all such calls.

Verifying that a message m returned by `receive()` (cf. Figure 2) matches a pattern t is more involved. Using our cryptographic library's specifications, we can verify that $\gamma(m)$ is equal to $\gamma(t\sigma)$, where the substitution σ instantiates the variables of t with messages. Unfortunately, this does not entail that the received message m matches the pattern t. The function γ may have collisions and hence $\gamma(m)$ may equal $\gamma(t\sigma)$, while m and $t\sigma$ differ. We address this issue by requiring that instances of the I/O specification's patterns do not collide with other bytestrings; we discuss below how we justify this requirement.

Definition 1. The *pattern requirement* for a pattern $t \in \mathcal{T}$ is defined by

$$\gamma(t\sigma) = \gamma(m) \implies \exists \sigma'.\, m =_{\mathsf{E}} t\sigma'. \qquad (\mathrm{PaR}(t))$$

This requirement states that if messages m and $t\sigma$ coincide under γ, then m must match the pattern t (mod E) with some substitution σ', which may differ from σ.

We need the pattern requirement for all patterns of the I/O specification. For code verification, we express the pattern requirement as a ghost function whose pre- and postcondition are the left-hand and right-hand side of the pattern requirement, for each pattern respectively. To apply the pattern requirement, the corresponding ghost function is called in the code. In Sections 4.3.2 and 4.3.3, we will explain how to prove the pattern requirement for a given pattern t.

Example 9 (Checking a ciphertext)**.** Consider a simple protocol where a role Ann expects a message matching the pattern $\mathsf{enc}(k, x)$, where k is a pre-shared key. We use the fact $Step^1_{Ann}(k)$ to bind k in the model state. Figure 3 shows part of an implementation. The variable `key` stores the pre-shared key, expressed as $\texttt{seq(key)} = \gamma(\texttt{k})$. After successfully receiving a bytestring `ciph` with a message `c`, $\texttt{seq(ciph)} = \gamma(\texttt{c})$ holds due to `receive`'s specification (cf. Figure 2). Next, the code decrypts `ciph`. If successful, `ciph` equals the bytestring $\gamma(\texttt{enc(k,u)})$ for some message `u` with $\texttt{seq(msg)} = \gamma(\texttt{u})$ (lines 5–6) by `decrypt`'s postcondition (cf. Figure 2) and γ being a surjective homomorphism. Furthermore, we know that $\gamma(\texttt{enc(k,u)})$ equals $\gamma(\texttt{c})$, but not yet that the received message `c` matches the pattern $\mathsf{enc}(k, x)$ (line 8). For this, we apply the pattern requirement by calling the ghost function `PaR1` (cf. Figure 4). The constant k of the pattern $\mathsf{enc}(k, x)$ is passed as an argument to the call, and related to the state facts of the I/O specification via the ghost function's precondition (with $Step^1_{Ann}(\texttt{k}) \in^{\texttt{m}} \texttt{S}$).

4.3.2. Deriving the pattern requirement. The pattern requirement for a given pattern t can be derived from two more basic properties. We define these properties here and prove this implication. In Section 4.3.3, we will discuss assumptions and justifications regarding these properties.

The first property is *image disjointness*, which has two parts: (i) the images of (public and fresh) names under γ are pairwise disjoint and (ii) the image of any function $f^{\mathcal{B}}$ for $f \in \Sigma$ neither collides with the image of any other function $g^{\mathcal{B}}$, for $g \in \Sigma$, nor with the image of names under γ.

Definition 2 (Image disjointness)**.** Image disjointness holds under the following two conditions:

(i) γ is injective on the set of names $\mathcal{N}$ and
(ii) for all $f, g \in \Sigma$ such that $f \neq g$,

$$img(f^{\mathcal{B}}) \cap (img(g^{\mathcal{B}}) \cup \gamma(\mathcal{N})) = \emptyset. \qquad (\mathrm{ID}_{f,g})$$

The second property is *pattern injectivity* for a pattern t. This constitutes a much weaker form of standard injectivity. It is required to hold only for subterms $t' \sqsubseteq t$ and where, again, equality is guaranteed only modulo a substitution σ'.

Definition 3 (Pattern injectivity)**.** Pattern injectivity holds for a pattern t if, for all $f \in \Sigma$ occurring in t,

$$f(t'_1, ..., t'_k) \sqsubseteq t \wedge f^{\mathcal{B}}(\gamma(t'_1\sigma), ..., \gamma(t'_k\sigma)) = f^{\mathcal{B}}(b_1, ..., b_k) \implies \exists \sigma'.\, b_1 = \gamma(t'_1\sigma') \wedge ... \wedge b_k = \gamma(t'_k\sigma'). \qquad (\mathrm{PaI}_f(t))$$

Proposition 1. *Given a linear pattern t (where every variable occurs only once), image disjointness and pattern injectivity for t imply the pattern requirement for t.*

Proof. We prove the proposition's statement for all $t' \sqsubseteq t$ by induction on t'. □

We split non-linear patterns into multiple linear ones. For instance, the non-linear pattern $t = \langle x, \mathsf{hash}(x)\rangle$ can be split into $t_1 = \langle x, \_\rangle$ and $t_2 = \langle \_, \mathsf{hash}(x)\rangle$ (where _ matches any term). Conceptually, we then first match a given term $\langle u, \mathsf{hash}(u)\rangle$ against t_1, which binds x to u, and then against $\langle \_, \mathsf{hash}(u)\rangle = t_2[x \mapsto u]$. This is equivalent to matching against t. This turned out to be simpler to work with than a single linearized pattern with additional equality constraints.

4.3.3. Assumptions and proof obligations. We discuss assumptions and proof obligations regarding image disjointness and pattern injectivity. In doing so, we distinguish cryptographic operations from formats.

Since we are working in a symbolic (Dolev-Yao) model, which assumes perfect cryptography, we maintain this assumption for cryptographic operations at the bytestring level in the following form.

Assumption 2 (Cryptographic operations)**.** We assume that

(i) γ is injective on the set of names $\mathcal{N}$,
(ii) ($\mathrm{ID}_{f,g}$) holds for cryptographic $f \in \Sigma$ and all $g \in \Sigma$,
(iii) ($\mathrm{PaI}_f(t)$) holds for all protocol patterns t and all cryptographic $f \in \Sigma$ occurring in t.

We justify these assumptions by noting that we can expect collisions violating these assumptions to occur only with negligible probability in good cryptographic libraries. Also recall that pattern injectivity is a much weaker requirement than standard injectivity.

The situation is different for formats (cf. Section 3.1.2). We can expect that the formats of a well-designed protocol are unambiguously parseable (i.e., injective and hence pattern-injective) and mutually disjoint (i.e., image disjoint). We therefore require that these properties are *proved* for formats, e.g., using the techniques proposed in [29], [30].

Remark. An obvious way to achieve image disjointness and pattern injectiveness is to *tag* each construct of the bytestring algebra with a different bytestring. This approach is followed, e.g., in [19] but it is unrealistic for real protocols.

Alternatively, image disjointness holds if different operations result in differently sized bytestrings. For operations with varying output sizes, such as stream encryption, this may require restricting the allowed argument sizes in the implementation. In some cases, this approach may allow us to *prove* image disjointness even for some cryptographic operations. Indeed, we do this for a pre-existing implementation of the WireGuard protocol, which does not use tagging. However, this approach also has its limitations; for example, AES-256 or SHA-256 have the same output size.

4.3.4. Proving term equalities. Obligations to prove term equalities during code verification arise from equality constraints in the I/O specification. However, while the code can check that an equality holds on the bytestring level (e.g., $\gamma(x) = \gamma(\mathsf{hash}(z))$), this does not in general imply the prescribed term equality (e.g., $x =_{\mathsf{E}} \mathsf{hash}(z)$). Following [31], [32], [33], [34], we can reasonably assume that collisions violating this implication do not occur (with overwhelming probability) in actual protocol executions and thus prove the specification under the condition that bytestring equality implies term equality for the two concrete bytestrings at hand (e.g., $\gamma(x) = \gamma(\mathsf{hash}(z)) \Longrightarrow x =_{\mathsf{E}} \mathsf{hash}(z)$). We call this a *collision-freedom* assumption for a given equation.

4.3.5. Summary. The verification of the role implementations, i.e., the Hoare triples $\vdash_{\pi'_{ext}} \{\psi_i(rid)\}\ c_i(rid)\ \{\mathsf{true}\}$, relies on the following assumptions:

1) contracts for the I/O and cryptographic libraries, where the former's operations are consistent with π'_{ext};
2) the pattern requirement for each pattern t occurring in the I/O specification; and
3) collision-freedom for all equalities Φ_R occurring in the I/O specification.

We suggest to also prove the pattern requirement for a given pattern t whenever possible, e.g., by showing image disjointness and pattern injectivity (at least) for formats and assuming them for the cryptographic operations (Assumption 2).

5. Concrete environment and overall soundness

We now derive an overall soundness result for our approach, which relates the abstract Tamarin model to the concrete protocol implementation.

5.1. Concrete environment

To formulate such a result, we must first define a concrete environment model $\mathscr{C}$, including a concrete attacker, which can interact with the roles' implementations. These implementations communicate with the environment using I/O library functions, which include non-ghost and ghost parameters: they send and receive both bytestrings (used by the program) and ghost terms (used for the reasoning), related by γ. The ghost parameters should be reflected in $\mathscr{C}$'s interface, i.e., its synchronization labels. Moreover, to fit into an overall soundness result, $\mathscr{C}$ must be trace-included in $\mathcal{R}^e_{\mathsf{env}}$. Hence, the concrete attacker must not be more powerful than the Dolev-Yao attacker, i.e., we must prevent attacks at the bytestring level such as exploiting collisions.

To achieve this, we construct the concrete environment from the term-level environment $\mathcal{R}^e_{\mathsf{env}}$ by changing only its *interface* with the protocol. Concretely, we rename and extend every synchronization label $[\lambda_F(rid, \overline{x})]$ of (the LTS induced by) $\mathcal{R}^e_{\mathsf{env}}$ to the label $\mathbf{F}(rid, \gamma(\overline{x}), \overline{x})$ in $\mathscr{C}$ and we keep the labels of $\mathcal{R}^e_{\mathsf{env}}$'s internal actions. Note that applying the relabeling $\pi_{ext} \circ \pi'_{ext}$ to $\mathscr{C}$ recovers $\mathcal{R}^e_{\mathsf{env}}$'s original labels. Hence, we record the following property.

Proposition 2. $\pi_{ext}(\pi'_{ext}(\mathscr{C})) \preccurlyeq \mathcal{R}^e_{\mathsf{env}}$.

5.2. Overall soundness result

Our goal now is to show that any trace property proved for the Tamarin model is preserved in the concrete system

$$\left(|||_{i,rid}\ \pi_{int}(\mathscr{C}_i(rid)) \right) \|_{\Lambda'} \mathscr{C},$$

which is composed of the verified programs' LTSs $\mathscr{C}_i$ and the concrete environment $\mathscr{C}$, where the programs' internal operations are mapped back to their action fact arguments and $\Lambda' = (\pi_{ext} \circ \pi'_{ext})^{-1}(\Lambda)$ synchronizes I/O permissions that also include bytestrings beside terms. Note that our soundness result assumes that the role implementations are already verified and that the verifier assumption (Assumption 1) holds. The verification of the role implementations themselves and the related assumptions are discussed in Section 4.

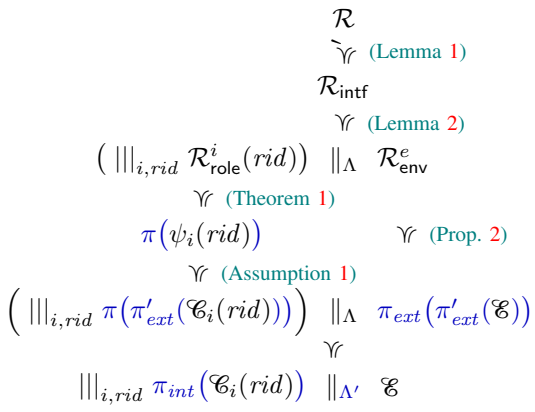

Figure 5. Overview of soundness proof, where π and Λ' are defined by $\pi = \pi_{int} \circ \pi_{ext}$ and $\Lambda' = (\pi_{ext} \circ \pi'_{ext})^{-1}(\Lambda)$.

Theorem 2 (Soundness)**.** *Suppose Assumption 1 holds and that we have verified, for each role i, the Hoare triple $\vdash_{\pi'_{ext}} \{\psi_i(rid)\}\ c_i(rid)\ \{\mathsf{true}\}$. Then*

$$(|||_{i,rid}\ \pi_{int}(\mathscr{C}_i(rid)))\ \|_{\Lambda'}\ \mathscr{E} \preccurlyeq_{\mathrm{t}} \mathcal{R}.$$

Proof. We decompose the proof into a series of trace inclusions. Figure 5 gives an overview of the proof.

The first trace inclusion is

$$\begin{aligned}&\left(|||_{i,rid}\ \pi_{int}(\mathscr{C}_i(rid))\right) \|_{\Lambda'}\ \mathscr{E}\\ \preccurlyeq\ &(|||_{i,rid}\ \pi(\pi'_{ext}(\mathscr{C}_i(rid)))) \ \|_\Lambda\ \pi_{ext}(\pi'_{ext}(\mathscr{E})),\end{aligned} \tag{9}$$

where the first line is obtained from the second by pushing the relabeling $\pi_{ext} \circ \pi'_{ext}$ into the parallel composition, thus changing the set of synchronizing labels from Λ to Λ'. Next, we deduce

$$\pi(\pi'_{ext}(\mathscr{C}_i(rid))) \preccurlyeq \mathcal{R}^i_{\mathsf{role}}(rid) \tag{10}$$

from Theorem 1 and from the combination of Assumption 1 and the assumption $\vdash_{\pi'_{ext}} \{\psi_i(rid)\}\ c_i(rid)\ \{\mathsf{true}\}$. We then leverage a general composition theorem [19, Theorem 2.3] that implies that trace inclusion is compositional for a large class of composition operators including $|||$ and $\|_\Lambda$. We apply this to the trace inclusion (10) and the one from Proposition 2 to derive the trace inclusion

$$\begin{aligned}&(|||_{i,rid}\ \pi(\pi'_{ext}(\mathscr{C}_i(rid)))) \ \|_\Lambda\ \pi_{ext}(\pi'_{ext}(\mathscr{E}))\\ \preccurlyeq\ &(|||_{i,rid}\ \mathcal{R}^i_{\mathsf{role}}(rid)) \ \|_\Lambda\ \mathcal{R}^e_{\mathsf{env}}.\end{aligned} \tag{11}$$

Our result follows by combining the trace inclusions (9) and (11) with Lemmas 1 and 2 and the relation inclusions $\preccurlyeq \subseteq \preccurlyeq' \subseteq \preccurlyeq_{\mathrm{t}}$ from Section 2. □

Corollary 1 (Property preservation)**.** *Any trace property Φ that holds for $\mathcal{R}$ also holds for $(|||_{i,rid}\ \pi_{int}(\mathscr{C}_i(rid)))\ \|_{\Lambda'}\ \mathscr{E}$.*

Example 10 (Secrecy for Diffie-Hellman)**.** We have verified the secrecy of and agreement on the exchanged key for the Tamarin protocol model from Example 2. The former is expressed as the requirement that in all possible traces of the system, the attacker never learns a value k for which a $Secret(k)$ action occurs in the trace. We then implemented both roles by programs, and proved they satisfy their I/O specifications (Example 8). Our soundness result and its corollary guarantee that the composed system also satisfies key secrecy and authentication.

6. Application and WireGuard case study

To provide evidence that our approach to verifying cryptographic protocol implementations is general, powerful, and scales to complex real-world protocols, we use it to verify our Diffie-Hellman running example and the WireGuard protocol. Both case studies are available open-source [21].

6.1. Applying our approach

The application of our approach involves three steps:

1) *Protocol model specification and verification in Tamarin.* Protocol models must satisfy our mild format restrictions (Section 3.1). Existing protocol models may require minor syntactic modifications. Verify the desired trace properties such as secrecy and authentication.
2) *Generation of the roles' I/O specifications.* Our tool automatically generates each protocol role's I/O specification along with definitions of types and internal operations. It accepts all protocol models satisfying our format assumptions. The tool currently supports Gobra (for Go) and VeriFast (for Java).
3) *Role implementation and verification.* Verify an existing or new role implementation against its I/O specification. This relies on user-provided (reusable) contracts for the I/O and cryptographic libraries used and on proofs of the relevant instances of the pattern requirement (e.g., using Assumption 2 and Proposition 1).

By Corollary 1 (and Assumption 1), all properties proven for the Tamarin model are inherited by the implementation.

For our Diffie-Hellman example, we have verified a Go implementation (using Gobra) and a Java implementation (using VeriFast) against their generated specifications. These two implementations are interoperable and exchange messages via UDP. We have also produced a faulty implementation that sends x instead of g^x as the first message and for which verification fails because the I/O permissions do not permit sending this payload.

6.2. The WireGuard key exchange

WireGuard is an open VPN (Virtual Private Network) system that is widely deployed on various platforms and integrated into the Linux kernel. Its core is the WireGuard cryptographic protocol.

The WireGuard protocol mainly consists of a handshake, where two agents establish secret session keys and authenticate each other, and a transport phase, where they use these keys to set up a secure channel for message transport. We give an overview of the protocol in Figure 6. The complete protocol additionally features denial of service (DoS) protection mechanisms, which we omit.

// handshake phase

$A \to B: \quad \langle 1, sid_I, g^{ek_I}, c_{pk_I}, c_{ts}, mac1_I, mac2_I \rangle$

$B \to A: \quad \langle 2, sid_R, sid_I, g^{ek_R}, c_{empty}, mac1_R, mac2_R \rangle$

// transport phase

$A \to B: \quad \langle 4, sid_R, 0, \mathrm{aead}(k_{IR}, 0, p_0, '') \rangle$

$B \to A: \quad \langle 4, sid_I, 0, \mathrm{aead}(k_{RI}, 0, p_0', '') \rangle$

$A \to B: \quad \langle 4, sid_R, 1, \mathrm{aead}(k_{IR}, 1, p_1, '') \rangle$

...

Figure 6. The WireGuard protocol.

The protocol involves two roles, the initiator (Alice) and the responder (Bob), each with long-term private and public keys. It is assumed that Alice and Bob know each other's public keys pk_I and pk_R in advance. They may optionally use a pre-shared secret. The protocol relies on an authenticated encryption with associated data (AEAD) construction aead, and hash and key derivation functions. The exact algorithms used are irrelevant for our presentation. All protocol messages contain a tag: 1 and 2 for handshake messages, 3 for the optional DoS prevention messages (not shown), and 4 for transport messages. They also contain randomly generated unique session identifiers sid_I or sid_R (for each role).

The *handshake phase* comprises two messages. Alice and Bob generate fresh ephemeral Diffie-Hellman keys ek_I, ek_R, and exchange the associated public keys g^{ek_I}, g^{ek_R}. They also exchange ciphertexts c_{pk_I}, c_{ts}, and c_{empty}, which respectively encrypt Alice's public key, a timestamp, and the empty string, with keys derived from both long term and ephemeral secrets. The messages also contain message authentication codes (MACs) for the DoS protection mode, not described here. At the end of the handshake, both agents compute two symmetric keys k_{IR} and k_{RI}. The detailed message construction and key computation is given in Appendix D.

These two keys are afterwards used to encrypt messages in the *transport phase*: k_{IR} for messages from initiator to responder and k_{RI} for the other direction. Alice and Bob both keep two counters n_{IR} and n_{RI}, counting the number of messages sent in each direction. When Alice sends a message, she encrypts it with k_{IR}, using the current value of n_{IR} as the AEAD nonce, and increments n_{IR}. Thus, no confusion is possible regarding the order of messages. The use of different keys and counters allows messages to be sent independently in each direction, without requiring strict alternation.

Note that the protocol mandates that Alice sends the first transport message. The reason is that it is used by Bob to confirm she has received his key. In contrast, Alice confirms this for Bob when she receives the second handshake message.

6.3. Tamarin model

We model the WireGuard protocol in Tamarin as a MSR system satisfying the assumptions from Section 3.1. Our model features rules for each of the two roles' behavior, as well as environment rules modeling the initial long-term key distribution. The environment may spawn any number of instances of each role, i.e., an unbounded number of sessions running in parallel, between the same or different agents.

Table 1. PROPERTIES VERIFIED FOR THE WIREGUARD CASE STUDY. BY COROLLARY 1, THE CODE INHERITS THE PROTOCOL'S PROPERTIES.

Level	Verified properties
Protocol	Agreement on the keys, forward secrecy
Code	Memory safety, conformance with generated I/O spec

Note that we not only model the handshake and first transport message, after which the key exchange is concluded, but also the loop that follows where agents may exchange any number of transport messages, in any order, using the computed keys. Verifying such unbounded loops is challenging for automated tools, and often leads to non-termination. For this reason, they are usually not modeled in their full generality. However, the presence of the loop in the implementation required its inclusion in the model as well, so that the implementation adheres to the model's behavior. We had to manually write three lemmas and an *oracle* (a heuristic for Tamarin's proof search) to help the tool terminate. Tamarin can then prove these lemmas and verify the model automatically.

We formulate and prove in Tamarin trace properties expressing authentication (see Table 1). More precisely, we show that, after the first transport message, the participants mutually agree on the resulting keys: if Alice believes she has exchanged k_{IR} and k_{RI} with Bob, then Bob also believes so, and conversely. Moreover, we prove the forward secrecy of the keys k_{IR} and k_{RI}: they remain secret from the attacker, provided neither Alice nor Bob's long-term secrets were corrupted *before the end of the key exchange*. The Tamarin file consists of about 250 lines of MSR rules, and 100 lines of lemmas and properties. It is verified automatically by Tamarin in about 3 minutes.

6.4. Implementation and code verification

We separately verified the initiator and responder code of the official Go implementation of WireGuard [35] in Gobra. We first proved memory safety, which is independent of our approach, but a required initial step in tools like Gobra and VeriFast. We then verified each role implementation against its I/O specification, which we generated with our tool from the WireGuard Tamarin model. We annotated the code with specifications, namely pre- and postconditions and loop invariants, and proof annotations, namely assertions, lemma calls, and predicate unfolding commands. The code can be compiled, since the annotations appear inside comments. Table 1 summarizes all properties we proved.

Changes to the implementation. We modified the official Go implementation in three ways. First, we removed features not included in our Tamarin model, namely DDoS protection. Second, we made changes to simplify proving memory safety. For this, we removed metrics and load balancing, which requires complex concurrency reasoning currently not supported by Gobra. Lastly, we performed changes related to our approach. Namely, we wrote stubs for cryptographic and network operations and equipped them with trusted specifications (cf. Section 4.3) and adapted the code accordingly.

Our verified implementation is interoperable with the official implementation and can delegate OS traffic over a VPN.

Verified components. We verified both the handshake and transport phase for both roles. These components are responsible for all I/O operations of the implementation. We did not verify the setup code for network sockets and cryptographic keys. The stubs for cryptographic and network operations are trusted by assumption and thus also not verified.

I/O specification. In addition to the I/O specifications generated by our tool, we declared the bytestring algebra operations and the homomorphism γ.

Our verification of the I/O specification is standard. To verify a call to an I/O operation or an internal operation, we extract the corresponding I/O permission from the predicate $P_i(p, rid, S)$. This requires facts about the model state S. For instance, to send a bytestring b, there must exist a term t with $\gamma(t) = b$ such that $\mathsf{out}_i(rid, t) \in S$. We verify such facts by relating the program state to the model state S.

Tool expertise requirements. Performing the code-level verification requires a basic understanding of separation logic and tool-specific knowledge to complete the memory-safety proof (as is necessary when verifying any heap-manipulating program). Justifying each of the program's I/O operations by relating them to an I/O permission of the I/O specification is mostly straightforward.

Pattern requirement. Each of the protocol's three non-linear patterns induces two instances of the pattern requirement, realized as lemma functions (cf. Figure 4).

We proved the pattern requirement instances using Proposition 1 and Assumption 2 by showing that all formats are (i) image-disjoint with each other and names and (ii) pattern-injective. Point (i) holds, since all formats start with a different constant and their lengths differ from the lengths of bytestrings representing names. Formats are also injective and hence satisfy (ii), as the arguments of all formats appear at fixed offsets in the bytestring representation.

Statistics. Our WireGuard implementation consists of 608 lines of verified Go code, excluding library stubs. Specifications and proof annotations make up 3936 lines of code, 1241 (32%) of which are generated by our tool. We required 87 lines of code to declare the term and bytestring algebras and the axioms about γ. The verification runtime is about 148 and 138 seconds for the initiator and responder, respectively. This case study demonstrates that our approach is applicable to pre-existing real-world security protocols with implementations of considerable size.

The smaller Diffie-Hellman implementations consist of 106 lines of verified Go code and 113 lines of verified Java code (both excluding library stubs). The Go and Java code require 1015 and 1022 lines of specification of which 623 (61%) and 780 (76%) are generated by our tool, respectively.

7. Related work

We compare our work with different kinds of approaches to formally verifying protocol implementations. We focus on symbolic approaches, as we have already discussed their relation to computational approaches in the introduction.

Model extraction and code generation. Bhargavan et al. [10] present a sound model extractor from (a first-order subset of) F# to ProVerif models. They work with an abstract datatype of bytestrings and corresponding interfaces for the cryptographic and network libraries, which they instantiate both to symbolic terms for prototyping and to actual library implementations. This approach is used in [36] to verify TLS 1.0. In [4], the authors extract models from a typed JavaScript reference implementation of TLS 1.2 and TLS 1.3. Several works generate both models for verification and executable code from abstract protocol descriptions. In [7], [8], Alice&Bob-style protocol specifications are translated into ProVerif models and into JavaScript or Java implementations. While in [8], the authors prove the correctness of a partial translation from a high-level to a low-level semantics, neither paper proves the full translation's correctness. Sisto et al. [9] generate a ProVerif model and a refined Java implementation from an abstract Java protocol specification and prove the implementation's soundness. We have already discussed the drawbacks of this family of approaches in the introduction.

Code verification only. Bhargavan et al. [37] modularly verify protocol code written in F# using the F7 refinement type checker [38]. They rely on protocol-specific invariants for cryptographic structures, e.g., stating which messages are public. Vanspauwen and Jacobs [33], [34] use a similar approach for protocols implemented in C and verified using VeriFast. They allow the concrete attacker to directly manipulate bytestrings, which they overapproximate symbolically by a set of terms. However, it is unclear what effect these manipulations have on message parsing in the protocol roles. While the verification of global protocol properties in the earlier work [37] required additional hand-written proofs, the more recent work [11] enables their verification in a single tool, F*, by explicitly incorporating a global event trace.

While these approaches are also modular and thus scale to protocols like Signal, finding a suitable protocol-specific invariant is challenging. Our approach not only decouples proving the security properties from verifying the implementation's correctness, it also leverages Tamarin's automated proof search.

Combined model and code verification. Dupressoir et al. [31], [32] use the interactive prover Coq for model verification in combination with the C code verifier VCC. Their approach involves reasoning about concrete bytestrings and their relation to terms. The central definitions of the protocol model are duplicated in Coq and in VCC and some theorems proven in Coq are imported as axioms into VCC.

Igloo [19] is a framework for distributed system verification that soundly combines model refinement in Isabelle/HOL with code verification using I/O specifications. Their case studies include a simple authentication protocol. We follow similar steps to extract I/O specifications from Tamarin models, but do this generically and automatically for a large class of protocol models. In contrast, these steps must be repeated in Igloo for each protocol, which requires

Isabelle/HOL expertise. Moreover, our way of relating terms and bytestrings is more flexible and realistic than theirs, which assumes an injective function from bytestrings to terms. Penninckx et al. [28] introduced I/O specifications for verifying programs' I/O behaviors, but they did not propose a method for verifying global system properties.

Message parsing. Mödersheim and Katsoris [29] show that an abstract symbolic model using message formats soundly abstracts a more concrete model, which includes associative message concatenation, variable and fixed length fields, and tags. Their result holds for protocols whose formats are uniquely parseable and image-disjoint and they give algorithms to check these conditions. This work has inspired our use of bytestring algebras and our decomposition of the pattern requirement into image disjointness and pattern injectivity. Their focus is on abstraction soundness, whereas ours is on code verification. EverParse [30] is a framework to generate provably secure (i.e., injective and surjective) parsers and serializers for authenticated message formats.

8. Conclusion

We have proposed a novel approach to cryptographic protocol verification that soundly bridges abstract design models, specified as multiset rewriting systems, with code-level specifications. This allows us to leverage the automation and proof techniques available in Tamarin for design verification together with state-of-the-art program verifiers to obtain security guarantees for protocol implementations. Our approach is general, compatible with different code verification tools, and applicable to real-world protocols.

There are several exciting directions for future work. Our framework is based on a Dolev-Yao attacker. It would be interesting to relax this assumption and allow the concrete attacker to perform additional (non-Dolev-Yao) bytestring operations. Moreover, we currently only support trace properties. Some security properties such as privacy properties can be formalized as observational equivalences, which Tamarin also supports [39]. How to obtain implementation-level equivalence guarantees from a symbolic model is an open problem.

Acknowledgements. We thank the Werner Siemens-Stiftung (WSS) for their generous support of this project. This work received funding from the France 2030 program managed by the French National Research Agency under grant agreement No. ANR-22-PECY-0006. We would also like to thank the anonymous reviewers for their helpful feedback and Sofia Giampietro for her useful comments on an earlier draft of this paper.

References

[1] B. Schmidt, S. Meier, C. J. F. Cremers, and D. A. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, 2012, pp. 78–94. [Online]. Available: https://doi.org/10.1109/CSF.2012.25

[2] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, 2013, pp. 696–701. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_48

[3] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*. IEEE Computer Society, 2001, pp. 82–96. [Online]. Available: https://doi.org/10.1109/CSFW.2001.930138

[4] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 483–502. [Online]. Available: https://doi.org/10.1109/SP.2017.26

[5] D. A. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5G authentication," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 1383–1396. [Online]. Available: https://doi.org/10.1145/3243734.3243846

[6] D. A. Basin, R. Sasse, and J. Toro-Pozo, "The EMV standard: Break, fix, verify," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1766–1781. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00037

[7] P. Modesti, "AnBx: Automatic generation and verification of security protocols implementations," in *Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. García-Alfaro, E. Kranakis, and G. Bonfante, Eds., vol. 9482. Springer, 2015, pp. 156–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-30303-1_10

[8] O. Almousa, S. Mödersheim, and L. Viganò, "Alice and Bob: Reconciling formal models and implementation," in *Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, 2015, pp. 66–85. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-25527-9_7

[9] R. Sisto, P. B. Copet, M. Avalle, and A. Pironti, "Formally sound implementations of security protocols with JavaSPI," *Formal Asp. Comput.*, vol. 30, no. 2, pp. 279–317, 2018. [Online]. Available: https://doi.org/10.1007/s00165-017-0449-8

[10] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 1, pp. 5:1–5:61, 2008. [Online]. Available: https://doi.org/10.1145/1452044.1452049

[11] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele, "DY*: A modular symbolic verification framework for executable cryptographic protocol code," in *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 523–542. [Online]. Available: https://doi.org/10.1109/EuroSP51992.2021.00042

[12] D. Cadé and B. Blanchet, "Proved generation of implementations from computationally secure protocol specifications," in *Principles of Security and Trust - Second International Conference, POST 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, ser. Lecture Notes in Computer Science, D. A. Basin and J. C. Mitchell, Eds., vol. 7796. Springer, 2013, pp. 63–82. [Online]. Available: https://doi.org/10.1007/978-3-642-36830-1_4

[13] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the TLS 1.3 record layer," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 463–482. [Online]. Available: https://doi.org/10.1109/SP.2017.58

[14] A. Delignat-Lavaud, C. Fournet, B. Parno, J. Protzenko, T. Ramananandro, J. Bosamiya, J. Lallemand, I. Rakotonirina, and Y. Zhou, "A security model and fully verified implementation for the IETF QUIC record layer," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1162–1178. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00039

[15] J. C. Reynolds, "Separation Logic: A logic for shared mutable data structures," in *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. [Online]. Available: https://doi.org/10.1109/LICS.2002.1029817

[16] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of Go programs," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12759. Springer, 2021, pp. 367–379. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_17

[17] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "VeriFast: A powerful, sound, predictable, fast verifier for C and Java," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Springer, 2011, pp. 41–55. [Online]. Available: https://doi.org/10.1007/978-3-642-20398-5_4

[18] M. Eilers and P. Müller, "Nagini: A static verifier for Python," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 596–603. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_33

[19] C. Sprenger, T. Klenze, M. Eilers, F. A. Wolf, P. Müller, M. Clochard, and D. A. Basin, "Igloo: Soundly linking compositional refinement and separation logic for distributed system verification," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 152:1–152:31, 2020. [Online]. Available: https://doi.org/10.1145/3428220

[20] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. Basin, and P. Müller, "Sound verification of security protocols: From design to interoperable implementations (extended version)," 2022. [Online]. Available: https://arxiv.org/abs/2212.04171

[21] ——, "Sound verification of security protocols: From design to interoperable implementations," Aug. 2022, artifact containing the specification generation tool and the case studies. [Online]. Available: https://doi.org/10.5281/zenodo.7409524

[22] C. Cremers and M. Dehnel-Wild, "Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://doi.org/10.14722/ndss.2019.23394

[23] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of TLS 1.3," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1773–1788. [Online]. Available: https://doi.org/10.1145/3133956.3134063

[24] G. Girol, L. Hirschi, R. Sasse, D. Jackson, C. Cremers, and D. Basin, "A spectral analysis of Noise: A comprehensive, automated, formal analysis of Diffie-Hellman protocols," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, aug 2020. [Online]. Available: https://usenix.org/conference/usenixsecurity20/presentation/girol

[25] D. Basin, R. Sasse, and J. Toro-Pozo, "Card brand mixup attack: Bypassing the PIN in non-Visa cards by using them for Visa transactions," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. [Online]. Available: https://usenix.org/conference/usenixsecurity21/presentation/basin

[26] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Information Theory*, vol. 29, no. 2, pp. 198–207, 1983. [Online]. Available: https://doi.org/10.1109/TIT.1983.1056650

[27] G. Lowe, "A hierarchy of authentication specification," in *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, 1997, pp. 31–44. [Online]. Available: https://doi.org/10.1109/CSFW.1997.596782

[28] W. Penninckx, B. Jacobs, and F. Piessens, "Sound, modular and compositional verification of the input/output behavior of programs," in *Programming Languages and Systems*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 158–182.

[29] S. Mödersheim and G. Katsoris, "A sound abstraction of the parsing problem," in *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society, 2014, pp. 259–273. [Online]. Available: http://dx.doi.org/10.1109/CSF.2014.26

[30] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "EverParse: Verified secure zero-copy parsers for authenticated message formats," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 1465–1482. [Online]. Available: https://microsoft.com/en-us/research/publication/everparse

[31] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*. IEEE Computer Society, 2011, pp. 3–17. [Online]. Available: https://doi.org/10.1109/CSF.2011.8

[32] ——, "Guiding a general-purpose C verifier to prove cryptographic protocols," *Journal of Computer Security*, vol. 22, no. 5, pp. 823–866, 2014. [Online]. Available: https://doi.org/10.3233/JCS-140508

[33] G. Vanspauwen and B. Jacobs, "Verifying protocol implementations by augmenting existing cryptographic libraries with specifications," in *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, ser. Lecture Notes in Computer Science, R. Calinescu and B. Rumpe, Eds., vol. 9276. Springer, 2015, pp. 53–68. [Online]. Available: https://doi.org/10.1007/978-3-319-22969-0_4

[34] ——, "Verifying cryptographic protocol implementations that use industrial cryptographic APIs," Department of Computer Science, KU Leuven, Belgium, Tech. Rep., 2017. [Online]. Available: https://lirias.kuleuven.be/retrieve/456879

[35] J. A. Donenfeld, "Go implementation of WireGuard," https://git.zx2c4.com/wireguard-go, [Online; accessed 11-March-2021].

[36] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, "Verified cryptographic implementations for TLS," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 3:1–3:32, 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133378

[37] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, M. V. Hermenegildo and J. Palsberg, Eds. ACM, 2010, pp. 445–456. [Online]. Available: https://doi.org/10.1145/1706299.1706350

[38] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. IEEE Computer Society, 2008, pp. 17–32. [Online]. Available: https://doi.org/10.1109/CSF.2008.27

[39] D. Basin, J. Dreier, and R. Sasse, "Automated symbolic proofs of observational equivalence," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1144–1155. [Online]. Available: https://doi.org/10.1145/2810103.2813662

Appendix A.
Protocol format details

We require that the rules' labels only contain facts from Σ_{act}, i.e. for all $\ell \xrightarrow{\mathscr{a}} r \in \mathcal{R}$, $\mathrm{facts}(\mathscr{a}) \subseteq \Sigma_{\mathsf{act}}$. Here, $\mathrm{facts}(s)$ denotes the set of fact symbols that occur in a multiset of facts s. We assume that environment rules do not directly use the agents' internal states. Namely, for all $\ell \xrightarrow{\mathscr{a}} r \in \mathcal{R}_{\mathsf{env}}$, $\mathrm{facts}(\ell \cup r) \subseteq \Sigma_{\mathsf{env}}$. In addition, any rule in $\mathcal{R}_{\mathsf{env}}$ producing a Setup_i fact must not produce any other facts on its right-hand side and its label must be empty.

We also require that rules for role i only use i's state and may consume facts in Σ_{in} (but must not produce them) and may produce facts in Σ_{out} (but must not consume them). More formally, we require, for all $\ell \xrightarrow{\mathscr{a}} r \in \mathcal{R}_i$, $\mathrm{facts}(\ell) \subseteq \Sigma^i_{\mathsf{state}} \cup \Sigma_{\mathsf{in}}$ and $\mathrm{facts}(r) \subseteq \Sigma^i_{\mathsf{state}} \cup \Sigma_{\mathsf{out}}$.

Finally, we require that for a protocol rule $\ell \xrightarrow{\mathscr{a}} r \in \mathcal{R}_i$, at least one state fact appears in r, and that there is a $k_i \geq 1$ such that the tuple of the first k_i arguments of all state facts in $\ell \xrightarrow{\mathscr{a}} r$ is the same. Intuitively, these first k_i arguments represent parameters of the run of the protocol role: their value remains fixed throughout the role's execution. They can be, for instance, the agent's identity, a thread identifier, or any value that is assumed to be known beforehand by the agent. We assume that the first one of these arguments, which we call rid, is of type *fresh*. It is intended to represent a thread identifier. For readability, we will usually group these k_i initial parameters as a tuple, denoted by $\overline{init}$.

In summary, these formatting rules only impose very mild constraints on Tamarin models. All protocol models in the Tamarin distribution could easily be adapted to conform to these constraints with only minor modifications. The main changes would be related to providing a separate setup rule for each role i and keeping the arguments of the resulting Setup_i fact as the initial arguments of all state facts as described above.

Appendix B.
Formal definition of parallel compositions

We define the (indexed) interleaving parallel composition $|||$ and the (binary) synchronizing parallel composition $\|_\Lambda$. These compose their argument MSR systems into a labeled transition system.

The (indexed) interleaving parallel composition $|||_{i,rid}\mathcal{R}_i(rid)$ has as states functions f that map each pair (i, rid) to a multiset of state facts and transitions $f \xrightarrow{\mathscr{a}} f'$ if, for some i and rid, $f(i, rid) \overset{\mathscr{a}}{\Longrightarrow}_{\mathcal{R}_i(rid)} S'$ and $f' = f[(i, rid) \mapsto S']$, where f' agrees with f except that it maps (i, rid) to S'.

The synchronized composed system $\mathcal{R}_1 \|_\Lambda \mathcal{R}_2$ has states of the form (S_1, S_2) and transitions $(S_1, S_2) \xrightarrow{\mathscr{a}} (S'_1, S'_2)$ if either

(i) $\mathscr{a} = []$ and there is an $\mathscr{a}' \in_{\mathsf{E}} \Lambda$ such that $S_1 \overset{\mathscr{a}'}{\Longrightarrow}_{\mathcal{R}_1} S'_1$ and $S_2 \overset{\mathscr{a}'}{\Longrightarrow}_{\mathcal{R}_2} S'_2$,

(ii) $\mathscr{a} \notin_{\mathsf{E}} \Lambda$, $S_1 \overset{\mathscr{a}}{\Longrightarrow}_{\mathcal{R}_1} S'_1$ and $S'_2 = S_2$, or

(iii) $\mathscr{a} \notin_{\mathsf{E}} \Lambda$, $S_2 \overset{\mathscr{a}}{\Longrightarrow}_{\mathcal{R}_2} S'_2$ and $S'_1 = S_1$.

Here, $\mathscr{a}' \in_{\mathsf{E}} \Lambda$ means that $\mathscr{a}' =_{\mathsf{E}} \mathscr{a}$ for some $\mathscr{a} \in \Lambda$.

Appendix C.
Extended verifier assumption

Following Penninckx et al. [28], we sketch an example of semantic assumptions on programs and Hoare triples that make our extended verifier assumption (Assumption 1) hold semantically. The soundness of the program logic itself, i.e., that a provable Hoare triple $\vdash_\alpha \{\phi\}\ c\ \{\psi\}$ implies its semantic validity $\models_\alpha \{\phi\}\ c\ \{\psi\}$, is a separate topic beyond the scope of our paper.

We assume that the programming language semantics makes judgements of the form $s, c \Downarrow s', \tau$, meaning that the program c when started in state s terminates in state s' and produces the I/O trace τ. This semantics induces a LTS $\mathscr{C}$, whose set of traces for a given starting state s_0 is thus

$$\mathrm{Tr}(\mathscr{C}) = \{\tau \mid \exists s'.\, s_0, c \Downarrow s', \tau\}.$$

I/O specifications ϕ have both a static and a dynamic semantics, which are defined in terms of *(I/O) heaps*. Heaps are multisets of (ground) I/O permission and token predicates. The static semantics, written $h \models \phi$, intuitively means that h contains (at least) the I/O permissions and tokens prescribed by ϕ. The dynamic semantics defines the set of traces allowed by an I/O specification ϕ to contain those traces that are possible in all heap models of ϕ, i.e.,

$$\mathrm{Tr}(\phi) = \{\tau \mid \forall h.\, h \models \phi \Longrightarrow h \xrightarrow{\tau}\},$$

where $h \xrightarrow{\tau}$ intuitively means that it is possible to produce a trace τ by successively pushing the tokens in h through the I/O permissions in h (and thus consume these permissions).

The semantics of Hoare triples of the form $\{\phi\}\, c\, \{\mathsf{true}\}$ with respect to an abstraction function α from program-level I/O operations to abstract I/O permissions is given by

$$\begin{aligned}
&\models_\alpha \{\phi\}\ c\ \{\mathsf{true}\} \\
&\overset{\text{def}}{\iff} \forall s, \tau, s', h.\ s, c \Downarrow s', \tau \wedge h \models \phi \Longrightarrow h \xrightarrow{\alpha(\tau)} \\
&\iff \alpha(\mathrm{Tr}(\mathscr{C})) \subseteq \mathrm{Tr}(\phi) \\
&\iff \alpha(\mathscr{C}) \preccurlyeq \phi.
\end{aligned}$$

Here, $\alpha(\mathscr{C})$ denotes the LTS $\mathscr{C}$ whose transition labels are renamed under α. The final equivalence uses the equality $\alpha(\mathrm{Tr}(\mathscr{C})) = \mathrm{Tr}(\alpha(\mathscr{C}))$.

Penninckx et al. [28]'s semantics is formulated for the case where α is the identity function. Both our and their semantics of Hoare triples also include non-trivial post-conditions, which we omit here to simplify the presentation.

Appendix D.
WireGuard message construction

The details of the construction of the ciphertexts in the messages for the WireGuard protocol are displayed in Figure 7, where:

- aead is an AEAD algorithm, h is a hash function, kdf_1, kdf_2, kdf_3 are key derivation functions;
- in $\mathrm{aead}(k, n, p, a)$, k is the key, n a nonce or counter, p the payload (both authenticated and encrypted), and a the additional data (authenticated, but not encrypted);
- (k_I, pk_I) and (k_R, pk_R) are the initiator and responder's long-term private and public keys;
- $(ek_I, epk_I = g^{ek_I})$ and $(ek_R, epk_R = g^{ek_R})$ are the initiator and responder's ephemeral private and public Diffie-Hellman keys, g being the group generator;
- *info* and *prologue* are fixed strings containing protocol information (version, etc.); and
- psk is an optional pre-shared key – if unused it is set to a string of zeros.

c_{pk_I}, c_{ts}, c_{empty} are computed as follows.

$$
\begin{array}{ll}
c_0 = & \mathrm{h}(\textit{info}) \\
h_0 = & \mathrm{h}(\langle c_0, \textit{prologue}\rangle) \\
h_1 = & \mathrm{h}(\langle h_0, pk_R\rangle) \\
c_1 = & \mathrm{kdf}_1(\langle c_0, epk_I\rangle) \\
h_2 = & \mathrm{h}(\langle h_1, epk_I\rangle) \\
c_2 = & \mathrm{kdf}_1(\langle c_1, g^{k_R * ek_I}\rangle) \\
k_1 = & \mathrm{kdf}_2(\langle c_1, g^{k_R * ek_I}\rangle) \\
c_{pk_I} = & \mathrm{aead}(k_1, 0, pk_I, h_2) \\
h_3 = & \mathrm{h}(\langle h_2, c_{pk_I}\rangle) \\
c_3 = & \mathrm{kdf}_1(\langle c_2, g^{k_R * k_I}\rangle) \\
k_2 = & \mathrm{kdf}_2(\langle c_2, g^{k_R * k_I}\rangle) \\
c_{ts} = & \mathrm{aead}(k_2, 0, \textit{timestamp}, h_3) \\
h_4 = & \mathrm{h}(\langle h_3, c_{ts}\rangle) \\
c_4 = & \mathrm{kdf}_1(\langle c_3, epk_R\rangle) \\
h_5 = & \mathrm{h}(\langle h_4, epk_R\rangle) \\
c_5 = & \mathrm{kdf}_1(\langle c_4, g^{ek_R * ek_I}\rangle) \\
c_6 = & \mathrm{kdf}_1(\langle c_5, g^{ek_R * k_I}\rangle) \\
c_7 = & \mathrm{kdf}_1(\langle c_6, psk\rangle) \\
\pi = & \mathrm{kdf}_2(\langle c_6, psk\rangle) \\
k_3 = & \mathrm{kdf}_3(\langle c_6, psk\rangle) \\
h_6 = & \mathrm{h}(\langle h_5, \pi\rangle) \\
c_{empty} = & \mathrm{aead}(k_3, 0, \text{""}, h_6)
\end{array}
$$

k_{IR} and k_{RI} are the resulting exchanged keys: $k_{IR} = \mathrm{kdf}_1(c_7)$, $k_{RI} = \mathrm{kdf}_2(c_7)$.

Figure 7. The WireGuard message construction