Sound Verification of Security Protocols: From Design to Interoperable Implementations

Abstract—**We address the problem of end-to-end verification of security protocols by providing a framework (consisting of tools and metatheorems) and an associated approach that bridge the gap between automated security protocol verification tools and code-level proofs. Our approach translates an abstract model of a protocol in the language of the Tamarin tool into an I/O specification expressed in separation logic, describing its intended I/O behavior. The implementation can then be verified against this specification at the code level. We prove that this process is sound: the verified implementation inherits all security guarantees proved as trace properties of the Tamarin model. As part of this effort, we provide a novel approach that relates the symbolic messages from the Tamarin model and their bytestring representation in the code.**

Our framework thus lets us leverage the substantial body of prior work on verifying protocols in Tamarin, to verify implementations. Additionally, the resulting specifications can be verified on existing implementations using any separation logic code verifier, providing flexibility regarding the target language. To validate our approach, and show that it scales to real-world protocols and implementations, we apply it to the WireGuard VPN key exchange protocol and verify a substantial part of its official Go 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 [2], [3] and ProVerif [4]. 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. This model allows for powerful automation techniques that enable state-of-the art tools to be used to automatically analyze real-world protocols, such as TLS [5], 5G [6], and EMV [7]. However, this automation comes at a price: the protocol verified is a highly abstract version of the actual protocol that is executed and there is *a priori* no formal link between these two versions. The problem of proving the security of protocol implementations has been studied before, but the solutions usually come with severe limitations. In particular, previous approaches tend to require implementation-specific proofs, meaning that one can no longer benefit from the automation capabilities provided by general-purpose symbolic protocol verifiers.

In this paper, we bridge the gap between abstract, symbolic security proofs and concrete, code-level proofs for security protocols. We present an approach that enables developers to write models and proofs of security protocols at an abstract, symbolic level, using all the available comfort and automation techniques available at that level, and then relate these to the protocols' code to obtain formal security guarantees at the implementation level.

Limitations of existing approaches. We will provide a detailed overview of related work in Section 7 and simply highlight the main limitations of previous approaches here.

Many existing approaches are based on model extraction or code generation, and either extract executable code from a relatively abstract verified model (e.g. [8], [9], [10]) or conversely extract a model from the code for verification (e.g. [11]). Another recent approach, DY* [12], provides a framework to write an executable implementation in F* and obtain a corresponding model that one can reason about symbolically, also in F*. Other methods adopt a computational approach, where security is proved in a computational model (e.g. [13], [14], [15]). 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 not performed at a fully abstract level by a standard security protocol verifier such as Tamarin or ProVerif, but rather tailored to the implementation considered. 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 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 new method 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 [16]), against which implementations can be verified. Its technical core is a procedure 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, by leveraging Tamarin's protocol models we can take advantage of Tamarin's proof automation capabilities to prove protocols secure. Moreover, we can prove a protocol's security once in Tamarin, and reuse this proof to verify multiple protocol implementations, 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, we produce I/O specifications, which can be encoded in many existing verifiers that support separation logic. In this work, we use the Go code verifier Gobra [17] for our case study, but the specifications we produce could be checked by other verifiers for other languages. Notably, the verifiers Nagini [18] for Python code and VeriFast [19] for Java code already support I/O specifications. In addition, the requirements for adding other code verifiers based on separation logic to our arsenal are low: they need only support abstract predicates to enable the encoding of I/O specifications and to guarantee that a successful verification implies a 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 and I/O specifications. This result follows a methodology inspired by the Igloo framework [20], which presents a series of generic steps to 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 to extract an I/O specification from a Tamarin model, which can be verified on implementation code. We prove the soundness of this general procedure.

Second, our approach is modular in two respects. It can be applied to a large class of Tamarin models, and it does not require any additional proofs on the model compared to any usual Tamarin security proof. Moreover, one may use any code verifier that supports separation logic to verify the resulting specification. This allows users to prove a protocol once and get a specification that can be verified on a variety of implementations in different languages.

Third, our method produces I/O specifications that still make use of some abstractions, in particular, they still represent I/O messages as terms rather than bytestrings. To allow code verifiers to prove them, we propose a novel approach to relate byte arrays on the code level with symbolic terms. This lets us write I/O specifications at a relatively high level, making them easier to relate to abstract protocol models and security properties, while still verifying the actual code manipulating bytestrings.

Finally, to validate our approach, we perform a substantial case study on a real-life protocol: the WireGuard key exchange, which is part of the widely-used WireGuard VPN that is part of the Linux kernel. We use our method to obtain an end-to-end symbolically verified implementation of WireGuard. In more detail, we model the protocol and prove its security in Tamarin. Afterwards, we use our approach to obtain an I/O specification that we verify on a Go implementation with the Gobra code verifier, using our method to relate terms and bytestrings. The implementation we verify is a part of the official Go implementation of WireGuard which is interoperable with the full version. This case study demonstrates the applicability of our approach to complex, real-world protocols and their implementations.

2. Background

We briefly present background on the tools and methodology that we use in this work.

2.1. Tamarin and multiset rewriting

The Tamarin prover [2], [3] 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 [6], [21], TLS 1.3 [22], the Noise framework [23], and payment card protocols [7], [24].

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 building blocks in order: messages, facts, and rules.

Message *terms* are elements of a *term algebra* $\mathcal{T} = \mathcal{T}_\Sigma(fresh \cup pub \cup \mathcal{V})$ built over a signature Σ of function symbols and countably infinite sets of fresh names $fresh$ (for secret values, generated by parties, which cannot be guessed by the attacker), public names pub (for globally known values), and 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). Its informal description is as follows:*

$$\begin{array}{lll} A \to B: & g^x & x \text{ fresh} \\ B \to A: & sign(\langle g^x, g^y\rangle, k_B) & y \text{ fresh} \\ A \to B: & sign(\langle g^y, g^x\rangle, k_A) & \text{agreement on } (g^x)^y = (g^y)^x \end{array}$$

To model this in Tamarin, we use a term signature containing the symbols $\hat{}$, g, $sign$, $verify$, pk, modeling respectively exponentiation, the group generator, signature, verification, and public keys. We use the simplified equational theory:

$$(g^x)^y = (g^y)^x \qquad verify(sign(x,k), pk(k)) = x$$

Tamarin's exponentiation model includes further equations.

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 [25]. 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}) 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 rule*, written $\ell \xrightarrow{a} r$, contains a multiset of *facts* ℓ on the left-hand side which is transformed into a multiset of facts r on the right-hand side, and is labeled with an a, which is a multiset of actions (also facts, but disjoint from state facts) used for property specification. Rules may contain variables, in which case they can be applied to any state containing facts that can be unified with the facts in the rule modulo E. A multiset rewriting system $\mathcal{R}$ and an equational theory E have a semantics as a labeled transition system (LTS), where the states are multisets of ground facts from $\mathcal{F}$, the initial state is the empty multiset $[]$, and the transition relation $\stackrel{\cdot}{\Longrightarrow}_{\mathcal{R},\mathsf{E}}$ is defined by the rule

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

where θ is a ground instance of the variables in ℓ, a, and r. Intuitively, the transition relation describes an update of a state S to a successor state that is possible when a given rule in $\mathcal{R}$ is applicable, i.e., an instantiation with a substitution θ of its left-hand side appears in the state S. Applying the rule consumes the linear facts appearing in that rule's left-hand side and adds the instantiations under θ of all the facts of the rule's right-hand side to the resulting successor state. Note that persistent facts are never removed from the state.

The multi-set rewriting 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's behavior is defined 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 can 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.

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

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

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

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

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 an agent A playing the role of Alice in the protocol. The parameters of the facts are used to store the agent's knowledge. In the initial state, the agent's knowledge is initialized with $\overline{init} = rid, A, k_A, pk_B$, i.e., a thread identifier, her identity, her private key, and her partner's public key. This knowledge 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:*

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

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

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) that she received from Bob remains secret. The fact $\mathsf{Setup}_{Alice}(\overline{init})$, consumed by the first rule, is produced by another rule, modeling the environment setting up Alice's initial knowledge with the fresh thread identifier rid, Alice's private key k_A, and Bob's public key pk_B.

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

Tamarin can be used to prove various kinds of properties. In this work, we focus on *trace properties*, i.e., sets of traces.

An MSR $\mathcal{R}$ satisfies a trace property Φ, if $\mathrm{Tr}_t(\mathcal{R}) \subseteq \Phi$. These properties state that each behavior of the protocol remains in a set of "good" traces Φ. These can express, for example, authentication [26] and secrecy. In Tamarin, these properties are specified in a fragment of first-order logic over traces. During verification, Tamarin considers all possible traces; so a property's proof guarantees the absence of any violation. Moreover, if a property is violated, Tamarin produces a counterexample trace, representing an attack.

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. A function may access a heap location only if it holds the associated permission; otherwise, a verification error occurs.

When allocating a new heap location, the current function execution obtains the corresponding permission. Permissions can be transferred between function executions upon call and return. The transfer of permissions during a function call is expressed in the callee's specification: its precondition specifies the permissions it requires from the caller, whereas the postcondition specifies which permissions are returned. For example, the precondition `acc(x) ⋆ *x > 0` specifies that the function requires permission to the heap location pointed by `x` (expressed by the accessibility predicate `acc`) and that the heap location's value is positive. The *separating conjunction* ⋆ sums up the permissions in its conjuncts. For instance, `acc(x) ⋆ acc(y)` denotes the permissions to the heap locations pointed by `x` and `y`. Since there is at most one permission per memory location, this implies that `x` and `y` point to different locations.

Permissions simplify reasoning about heap modifications. Since there is at most one permission per location, a callee function cannot modify any locations to which the caller retains permissions across the call, such that any properties of these locations are known to be preserved by the call. Moreover, the above rules ensure the absence of data races because no two function executions can both hold the permission to access a heap location.

`acc` denotes permission to an individual heap location. Permissions to an unbounded set of locations, for instance, all locations of a linked list, can be expressed via (possibly co-recursive) predicates, which abstract over permissions. *Abstract* predicates specify permissions to an unknown set of locations.

Permission-based reasoning generalizes from heap locations to other forms of resources manipulated by a program. Penninckx et al. [27] reason about the I/O behavior of a program by associating each I/O operation with a permission that is required to call the operation and 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

```
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 ⋆.

execution. These I/O specifications can easily be encoded into standard separation 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 $token(p_2)$. This induces an ordering relation on permitted I/O operations with the token indicating the program's current position in this ordering.

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.*

Since I/O permissions are simple resources, an I/O specification can provide multiple I/O permissions with the same source place by using the separating conjunction. This permits the program to execute *any one* of the corresponding I/O operations. Since this I/O operation will advance the token to the target place, only one of the I/O operations can be executed. That is, multiple I/O permissions with the same source place amount to a non-deterministic choice among the corresponding I/O operations. Moreover, co-recursion enables repeated as well as non-terminating sequences of I/O operations.

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

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

The formula $\phi = token(p) \star P(p, \emptyset)$ *specifies a non-terminating server that repeatedly and non-deterministically chooses between receiving a value* v *and sending the same value* v *back or sending a message with content "Ping". All values* v *that the server receives are recorded in the state* S *that is initially empty. Input parameters, like* v *in* **in** *here,*

are existentially quantified to avoid imposing restrictions on the values that can be 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, expressed as an MSR system $\mathcal{R}$, into a set of I/O specifications ψ_i, one for each protocol role i. They 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 and expressiveness for modeling protocols and their properties. We want to preserve this generality as much as possible. Hence, we seek a transformation that requires only minimal restrictions and covers most common protocol modeling styles.
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 Appendix C.

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. To model an n-role protocol, we will 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 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. We also assume that Σ_{env} contains two disjoint subsets, Σ_{in} and Σ_{out}, of input and output fact symbols. We have $\mathsf{Fr}, \mathsf{in} \in \Sigma_{\mathsf{in}}$, $\mathsf{out} \in \Sigma_{\mathsf{out}}$, and $\mathsf{K} \in \Sigma_{\mathsf{env}} \setminus (\Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}})$. Furthermore, we assume that 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 as follows:

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

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 these) 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, we refer the reader to Appendix A.

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.*

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 the 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.

We first add to the fact signature, for each input or output fact F and role i, a copy (the "buffer") F_i. We define

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

We then replace the facts used by the protocol rules as follows. Let Σ^-_{in} be the set Σ_{in} without the initialization facts Setup_i. 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.

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

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* $Alice$:

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

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

We show that the interface model refines the original one.

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

3.2.2. Decomposition. We are now ready to 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 component. 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. The different role instances and the environment thus operate on mutually disjoint sets of (ground) facts. Below, we will 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 the rule sets $\mathcal{R}^i_{\mathsf{role}}$ and $\mathcal{R}^e_{\mathsf{env}}$ operate on pairwise disjoint sets of facts, namely over the signatures Σ^i_{role} and Σ_{env}, respectively. This means that they can interact with each other only by synchronizing the split I/O events.

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.*

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

The traces of the recomposition of all roles with the environment are included in the traces of the interface model. Formally, we define two kinds of parallel compositions on LTSs (induced here by the MSR systems' transition semantics). 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*

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

where $\Lambda = \bigcup_i \{\alpha\theta \mid \exists \ell, \nu.\, \ell \xrightarrow{\alpha} \nu \in \mathcal{R}^i_{\mathsf{io}} \wedge range(\theta) \subseteq \mathcal{M}\}$ 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 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 $R = \ell \xrightarrow{\alpha} \nu \in \mathcal{R}'_i$ with variables $\overline{x}$. We associate the internal permission $\mathbf{R}(p, \overline{x}, \ell', \alpha', \nu', p')$ to R, and define $\phi_R(p, rid, S)$ by

$$\begin{array}{l}\phi_R(p, rid, S) = \forall \overline{x}, \ell', \alpha', \nu'.\\ \quad \text{if } M(\ell', s) \wedge \ell' =_{\mathsf{E}} \ell \wedge \alpha' =_{\mathsf{E}} \alpha \wedge \nu' =_{\mathsf{E}} \nu\\ \quad \text{then } \exists p'.\ \mathbf{R}(p, \overline{x}, \ell', \alpha', \nu', p') \star P_i(p', rid, U(\ell', \nu', S))\\ \quad \text{else } \top,\end{array}$$

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

This formula specifies that, for any instantiation (mod E) ℓ', α', ν' of the facts in the rule, if the matching condition $M(\ell', S)$ is satisfied, then we have an internal permission $\mathbf{R}$ to execute the rule's implementation. This yields an updated state $U(\ell', \nu', S)$, on which P_i is co-recursively applied to produce the permissions for the rest of the execution.

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

$$\begin{array}{l}\phi_{R_G}(p, rid, S) = \forall \overline{x}.\\ \quad \text{if } G_i(rid, \overline{x}) \in^{\mathsf{m}} S\\ \quad \text{then } \exists p'.\ \mathbf{R_G}(p, rid, \overline{x}, p') \star P_i(p', rid, S \setminus^{\mathsf{m}} [G_i(rid, \overline{x})])\\ \quad \text{else } \top.\end{array}$$

which grants the I/O permission $\mathbf{R_G}(p, rid, \overline{x}, p')$ for any rid and terms $\overline{x}$ for which 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 the fact $G_i(rid, \overline{x})$ 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

$$\begin{array}{l}\phi_{R_F}(p, rid, S) =\\ \quad \exists t', \overline{z}.\ \mathbf{R_F}(p, rid, \overline{z}, p') \star P_i(p', rid, S \cup^{\mathsf{m}} [F_i(rid, \overline{z})])),\end{array}$$

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:*

$$\begin{array}{l}\phi_{Alice_2}(p, rid, S) = \forall \overline{init}, x, Y, \ell', \alpha', \nu'.\\ \text{if } M(\ell', S) \wedge\\ \quad \ell' =_{\mathsf{E}} \{\!|Step^1_{Alice}(\overline{init}, x), \mathsf{in}_{Alice}(rid, sign(\langle g^x, Y\rangle, k_B))|\!\} \wedge\\ \quad \alpha' =_{\mathsf{E}} \{\!|Secret(Y^x)|\!\} \wedge\\ \quad \nu' =_{\mathsf{E}} \{\!|Step^2_{Alice}(\overline{init}, x, Y), \mathsf{out}_{Alice}(rid, sign(\langle Y, g^x\rangle, k_A))|\!\} \wedge\\ \text{then } \exists p'.\ \mathbf{Alice_2}(p, \overline{init}, x, Y, \ell', \alpha', \nu', p') \star P_i(p', rid, U(\ell', \nu', S))\\ \text{else } \top.\end{array}$$

i.e., 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.

The construction of ψ_i is compatible with the method presented in [20] and by the soundness result from that paper, we get the following trace inclusion.

Theorem 1 ([20]). *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 π relabels (the LTS induced by) $\psi_i(rid)$ as follows:

$$\begin{array}{rll}\pi(\mathbf{R}(\overline{x}, \ell', \alpha', \nu')) = \alpha' & & \text{for } R \in \mathcal{R}'_i\\ \pi(\mathbf{F}(rid, \overline{x})) = [\lambda_F(rid, \overline{x})] & & \text{for } F \in \mathcal{R}^i_{\mathsf{io}}.\end{array}$$

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' soundness guarantees 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 crypto library specifications.

4.1. Code verification with abstraction

In Penninckx et al.'s program logic [27], 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. $\mathscr{C} \preccurlyeq \phi$.

Since we formulate our I/O specifications abstractly at the term level (cf. Section 3) whereas our role implementations manipulate bytestrings, there is an abstraction gap between the two. To bridge this gap, we extend Penninckx et al.'s approach. The general idea is to introduce 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:

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

This means that a successful verification implies that the program traces, abstracted under α, are included in the I/O specification ϕ's traces.

To ensure this extension's soundness, we require that the specifications for I/O or internal operations imply a correct mapping of transition labels under α. More precisely, suppose the specification of such an operation $\mathtt{op}$ induces a concrete transition label $\mathbf{op}_c(\overline{a})$, where $\overline{a}$ are $\mathtt{op}$'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 $\mathtt{op}$'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.

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 $\alpha : \{0,1\}^\star \to \mathcal{M}$ 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. However, to minimize our assumptions, 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 $\gamma\colon \mathcal{M} \to \{0,1\}^\star$. 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 t with $b = \gamma(t)$ and the corresponding transition label is $\mathbf{in}_c(rid, (b, t))$. These ghost terms simplify verification (see Section 4.3), but are not present in the executable code. We instantiate α to the function π' that removes the bytestrings from the concrete I/O operation's labels and keeps only the ghost terms used for the reasoning. For instance, $\pi'(\mathbf{in}_c(rid, (b, t))) = \mathbf{in}_a(rid, t)$. This function is defined only for $b = \gamma(t)$, which is guaranteed by the receive operation's contract (cf. Figure 2).

Our proposed method enables us to verify that preexisting 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 Σ-algebras $\mathcal{B}$ with the set of bytestrings $\{0,1\}^\star$ as the carrier set, called *crypto-algebras*. To relate terms to bytestrings, we use a surjective Σ-algebra homomorphism $\gamma\colon \mathcal{M} \to \mathcal{B}$, which maps fresh and public values to bytestrings and the signature's symbols to functions on bytestrings:

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

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). We can make γ surjective without loss of generality, as there is a countably infinite number of public value terms that can be mapped to bytestrings that are not mapped to by any other term.

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

$$\mathsf{dec}^{\mathcal{B}}(key, \mathsf{enc}^{\mathcal{B}}(key, msg)) = msg \tag{9}$$

on bytestrings. In the next section, we will use the crypto-algebra's functions in our crypto 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 [27], [20]. Every I/O

```
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,?t,?p2)
ensures  ok ==> token(p2) && seq(b) == γ(t)
ensures !ok ==> token(p1) && in(p1,t,p2)
func receive() (b []byte,ghost t 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.

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.

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 $sign(\langle g^x, Y\rangle, k_B)$, where the previously known x and g are constants and the unconstrained Y is a variable.

Verifying that the I/O specification permits sending a message boils down to verifying that a 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 $\mathcal{B}$. Due to the specification and the surjectivity 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_{key}, where $\gamma(m_{key}) =$ `seq(key)` and $\gamma(m_{msg}) =$ `seq(msg)`. To verify the construction of an entire message, we combine the information of all such calls.

Verifying that a received message m matches a pattern t is more involved. Using our cryptographic library's specifications, we can verify that a received bytestring 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 therefore $\gamma(m)$ and $\gamma(t\sigma)$ may be equal while m and $t\sigma$ may differ. We address this issue by requiring that the patterns of the I/O specification do not have collisions; we discuss below how we justify this requirement. For a pattern $t \in \mathcal{T}$, we express this as the following general *pattern requirement*:

$$\gamma(t\sigma) = \gamma(m) \implies \exists \sigma'.\, m = t\sigma', \qquad \text{(PaR}(t)\text{)}$$

```
1 // seq(key) == γ(k) holds
2 ciph, m, ok := receive(); if !ok {return}
3 assert seq(ciph) == γ(m)
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 == 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 == 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.

which states that if messages m and $t\sigma$ have a collision under γ, then m must match the pattern t with some substitution σ'. That the substitution changes from σ to σ' is not a problem since the pattern's variables can take on any messages.

We assume the pattern requirement for all patterns of the I/O specification. For code verification, we express this assumption by declaring for each pattern, a ghost function whose pre- and postcondition are the left-hand and right-hand side of the pattern requirement, respectively. To apply the pattern requirement, the corresponding ghost function is called in the code.

Example 9 (Checking a ciphertext)**.** *Consider a simple protocol where a role Ann expects to receive 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* `seq(key)` $==\gamma($`k`$)$*. After successfully receiving a bytestring* `ciph` *with a message* `m`*,* `seq(ciph)` $==\gamma($`m`$)$ *holds due to* `receive`*'s specification (cf. Figure 2). Next, the code decrypts* `ciph`*. If this succeeds, then* `ciph` *is equal to the message* $\gamma($`enc(k,u)`$)$ *for some message* `u` *with* `seq(msg)` $==\gamma($`u`$)$ *(lines 5–6). This follows from* `decrypt`*'s postcondition (cf. Figure 2) and as* γ *is a surjective homomorphism. Furthermore, we know that* $\gamma($`enc(k,u)`$)$ *is equal to* $\gamma($`m`$)$*, but not yet that the received message* `m` *matches the pattern* $\mathsf{enc}(k, x)$ *(line 8). For this, the pattern requirement is applied 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* $Step^1_{Ann}($`k`$)\in^{\mathsf{m}} S$*).*

Deriving the pattern requirement. The pattern requirement for a pattern t can be derived from two more basic properties.

The first property is *image disjointness*: the images of all $f^{\mathcal{B}}$ (for f in Σ) are pairwise disjoint and disjoint to all public constants $c^{\mathcal{B}}$ and fresh values $n^{\mathcal{B}}$. Furthermore, public constants and fresh values occurring in t are also pairwise disjoint. In previous work [20], [11], image disjointness is

guaranteed trivially by using *message tags*. For message tags, the result of every bytestring operation is prepended with a different prefix. Alternatively, our approach also allows us to satisfy image disjointness if each bytestring operation results in a differently sized bytestring. For bytestring operations of varying output sizes, such as stream encryption, this may require restricting in the implementation the allowed argument sizes. Using such restrictions, we can verify a pre-existing implementation of the WireGuard protocol, which does not use message tags.

The second property is *pattern injectivity* for the pattern t, expressed as:

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

The relation $t' \sqsubseteq t$ expresses that t' is a subterm of t. Pattern injectivity is a weaker form of standard injectivity; it holds only for subterms of the pattern t and where, again, equality is guaranteed only modulo a substitution σ'. For a fixed I/O specification, we can formally prove that constants and tuples that are subterms of patterns satisfy pattern injectivity. To prove this, we can use techniques such as those proposed by Mödersheim and Katsoris [28]. For cryptographic constructors such as encryption and hashes, we have to assume pattern injectivity. Although real encryption and hashing have collisions (e.g., AES-256 or SHA-256), one usually expects collisions to happen with negligible probability in good cryptographic libraries.

Proposition 1. *Given a linear pattern t (where no variable occurs twice), 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'. □

Non-linear patterns can be transformed into linear pattern using additional equality constraints.

5. Overall soundness result

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

To formulate such a result, we must first define a concrete environment model $\mathscr{E}$, 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 $\mathscr{E}$'s interface, i.e., its synchronization labels. Moreover, to fit into an overall soundness result, $\mathscr{E}$ 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) $\mathcal{R}^e_{\mathsf{env}}$ with the label $\mathbf{F}(rid,\overline{x},\gamma(\overline{x}))$ in $\mathscr{E}$ and we keep the labels of $\mathcal{R}^e_{\mathsf{env}}$'s internal actions. Note that applying the relabeling $\pi \circ \pi'$ to $\mathscr{E}$ recovers $\mathcal{R}^e_{\mathsf{env}}$'s original labels. Hence, we record the following property.

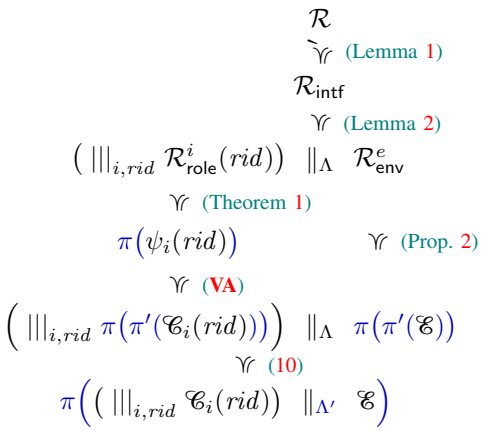

Figure 5. Overview of soundness proof.

Proposition 2. $\pi(\pi'(\mathscr{E})) \preccurlyeq \mathcal{R}^e_{\mathsf{env}}$.

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

$$(|||_{i,rid}\ \mathscr{C}_i(rid))\ \|_{\Lambda'}\ \mathscr{E},$$

which is composed of the program's LTSs $\mathscr{C}_i$ and the concrete environment $\mathscr{E}$, where $\Lambda' = (\pi\circ\pi')^{-1}(\Lambda)$ synchronizes I/O permissions that also include bytestrings beside terms. The following theorem achieves this, modulo the relabeling π.

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

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

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

The first trace inclusion is

$$\begin{aligned}&\pi((|||_{i,rid}\ \mathscr{C}_i(rid))\ \|_{\Lambda'})\mathscr{E}\\ \preccurlyeq\ &(|||_{i,rid}\ \pi(\pi'(\mathscr{C}_i(rid))))\ \|_\Lambda\ \pi(\pi'(\mathscr{E})),\end{aligned} \qquad (10)$$

where the first line is essentially obtained by pulling the relabelings $\pi \circ \pi'$ on the second line over all parallel compositions and changing the set of synchronizing labels from Λ to Λ'. Note that π' is not needed on the outside (first line), as I/O permissions are synchronized to transitions labeled with []. Next, we deduce the trace inclusion

$$\pi(\pi'(\mathscr{C}_i(rid))) \preccurlyeq \mathcal{R}^i_{\mathsf{role}}(rid). \qquad (11)$$

from Theorem 1 and from the combination of the program specification (8) and the verifier soundness assumption (**VA**).

We then leverage a general composition theorem [20, Theorem 2.3], which implies that trace inclusion is a compositional for a large class of composition operators including $|||$ and $\|_\Lambda$. We apply this to the trace inclusion (11) and the one from Proposition 2 to derive the trace inclusion

$$\begin{aligned}&(|||_{i,rid}\ \pi(\pi'(\mathscr{C}_i(rid)))) \ \|_\Lambda\ \pi(\pi'(\mathscr{E}))\\ \preccurlyeq\ &(|||_{i,rid}\ \mathcal{R}^i_{\mathsf{role}}(rid)) \ \|_\Lambda\ \mathcal{R}^e_{\mathsf{env}}.\end{aligned} \tag{12}$$

Our result now follows by combining the trace inclusions (10) and (12) with Lemmas 1 and 2 and the relation inclusions $\preccurlyeq \subseteq \preccurlyeq' \subseteq \preccurlyeq_{\mathrm{t}}$ from Section 2. □

Example 10 (Secrecy for Diffie-Hellman)**.** *We have verified the secrecy of the exchanged key on the Tamarin protocol model from Example 2, 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 guarantees that the composed system also satisfies the key secrecy property (modulo π). This concludes our running example.*

6. Case study: WireGuard

In order to show that our approach to verifying cryptographic protocol implementations is general, powerful, and scales to handle complex real-world protocols, we use it to verify the WireGuard protocol.

6.1. 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, derived from a protocol framework called Noise. This framework defines a family of Diffie-Hellman based protocols to be used in different contexts.

The WireGuard protocol mainly consists of a handshake phase, 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 are activated only when one of the agents is under load. Here we focus on the handshake and transport phases, which are the core of the protocol under normal operations.

The protocol involves two roles, the initiator (here Alice) and the responder (Bob), each with their 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 also use a pre-shared secret, although this is not mandatory. The protocol relies on an authenticated encryption with associated data (AEAD) construction aead, a hash function, 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 additional 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).

// 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
$A \to B:\quad \langle 4, sid_R, 0, \mathrm{aead}(k_{IR}, 0, p_0, '')\rangle$
$B \to A:\quad \langle 4, sid_I, 0, \mathrm{aead}(k_{RI}, 0, p'_0, '')\rangle$
$A \to B:\quad \langle 4, sid_R, 1, \mathrm{aead}(k_{IR}, 1, p_1, '')\rangle$
$\ldots$

Figure 6. The WireGuard protocol.

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

These two session keys are then 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 previously sent in each direction. When Alice sends a message to Bob, she encrypts it with k_{IR}, using the current value of n_{IR} as the nonce for the AEAD algorithm, and increments n_{IR}. Thus, no confusion is possible regarding the order in which messages were sent. The use of different keys and counters allows Alice and Bob to independently send messages in each direction, without a need for strict alternation.

Note that the protocol mandates that the first transport message is sent from the initiator to the responder. The reason is that this message is used by Bob to confirm that Alice has received his public key – in contrast, Alice confirms this for Bob when she receives the second handshake message.

6.2. Tamarin model

We model the WireGuard protocol in Tamarin as a MSR system satisfying the formatting assumptions from Section 3.1. Our model features rules modeling each of the two roles' behavior, as well as environment rules modeling the initial generation of long-term keys. The environment may spawn an arbitrary number of instances of each role (with fresh session identifiers), modeling an unbounded number of sessions of the protocol running in parallel, between the same or different agents.

Note that we not only model the handshake and first transport message, after which the authenticated key exchanged is concluded, but also the loop that follows where

the participants may exchange any number of transport messages, in any order, using the computed keys. Verifying such unbounded loops is challenging for automated tools such as Tamarin and may lead to non-termination. For this reason, such loops typically are not modeled in their full generality. However, the presence of such a loop in the implementation required its inclusion in the model as well, so that we can show that the implementation adheres to the behavior expected by the model. We had to manually write four lemmas and an *oracle* (a heuristic that guides 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 guarantees on the keys. More precisely, we show that, at the end of the handshake and the first transport message, the participants mutually agree on the resulting keys. That is, if Alice believes she has exchanged keys k_{IR} and k_{RI} with Bob, then Bob also believes he has exchanged these keys with Alice, and conversely. Moreover, we prove in Tamarin 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, even if they are corrupted afterwards. The Tamarin file containing the model and the security properties can be found at [1]. It consists of about 250 lines of MSR rules modeling the protocol, and 50 lines of lemmas and properties. It is verified fully automatically by Tamarin in about 45 seconds.

6.3. I/O specification

We apply our method to extract from the Tamarin model I/O specifications for the initiator and responder roles. As explained in previous sections, these specifications state that a program has no more behaviors than the corresponding multiset rewriting system. This extraction process is, for this case study, performed manually. It would not be difficult to automate by implementing an algorithm that performs each step of the transformation described in previous sections to produce the I/O specification.

6.4. Implementation

To simplify verification, we rewrote parts of the official Go implementation of WireGuard [29]. In particular, we removed load balancing and advanced VPN features. Our verified implementation is interoperable with the official implementation and can delegate OS traffic over a VPN. Our code implements both the initiator and responder roles. First, two messages are processed for the handshake phase and then, messages are sent and received in a loop for the transport phase.

We verified the role implementations against their I/O specifications in Gobra. We annotated the code with specifications, namely pre- and postconditions and loop invariants, and proof annotations, namely assertions, lemma calls, and commands managing predicate instances. All specifications and proof annotations are written within comments, so that the verified code can be compiled and executed as is.

For cryptographic and network operations, we wrote stubs that we equipped with trusted specifications as shown in Section 4.3. These trusted specifications are part of our assumptions and not verified. Underneath, the stubs connect to the same network and cryptographic libraries that the official implementation employs. The trusted specifications include the specifications of both I/O operations and internal operations. We add ghost statements that are present for verification only and whose purpose is to call the internal operations and thus advance the token and update the model state.

For the I/O specification, we declared new types for terms, facts, and claims in Gobra. More concretely, we declared uninterpreted types, uninterpreted functions, and axioms specifying properties of the uninterpreted functions, which include properties given by the equational theory. Similarly, we declared the operations of the crypto-algebra together with the homomorphism γ.

The pattern requirement is split into three instances, one for each top-level protocol message, each of which we write as a lemma function (cf. Figure 4). As a technicality, because the initializations of the private, public, and pre-shared key are modeled as I/O operations, we also need the pattern requirement to hold for these terms. For these, we add it to the postcondition of the corresponding I/O operation.

To verify a call to an I/O operation, we must extract from the predicate $P_i(p, rid, S)$ the corresponding I/O permission coinciding with the place of the held token. Extracting this permission 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 to the model state S, which is a multiset of facts and carried around as ghost state. Because the initiator and responder have separate I/O specifications, we can verify them independently. For our implementation, relating the program state to the model state was straightforward; the loop in the transport phase required appropriate invariants.

Our implementation consists of 608 lines of Go code that we verified, excluding library stubs. Specifications and proof annotations make up 2317 lines of code, 577 of which are for the I/O specification. We required 136 lines of code to declare the term and crypto-algebra and the necessary axioms about γ. Despite the large size of the specification, its complexity is relatively low and could even be partially generated. The I/O specification and all necessary types are a straightforward translation from their mathematical description to the syntax used by Gobra. Furthermore, the I/O specification's individual conjuncts can easily be mapped to corresponding sections in the implementation, which makes it easy to annotate functions with the correct specification.

The case study demonstrates that our approach is applicable to pre-existing real-world security protocols with implementations of considerable size. The I/O specification determines the model state and the I/O and internal operations, to which the implementation must connect. Establishing this connection was easy with Gobra. The verification runtime is about 246 and 223 seconds for the initiator and

responder, respectively. However, roughly 70 and 74% of these times are spent on parsing; the actual verification of the code against the I/O specification is quite fast for the code size and the complexity of the property.

7. Related work

We compare our work with different kinds of approaches to formally verifying protocol implementations.

Model extraction and code generation. Bhargavan et al. [11] 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 [30] to verify TLS 1.0. In [5], 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 [8], [9], Alice&Bob-style protocol specifications are translated into ProVerif models and into JavaScript or Java implementations. While in [9], 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. [10] 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. An alternative approach is entirely based on code verification. The work by Bhargavan et al. [31] enables the modular verification of protocol code written in F# using the F7 type checker for refinement types [32]. 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 [31] required additional hand-written proofs, the more recent work [12] enables their verification in a single tool, F*, by explicitly incorporating a global event trace.

The downside of these code-only approaches is that they lack abstraction: instead of verifying global security properties on an abstract model, one immediately faces the full complexity of the protocol code including all its details.

Combined model and code verification. Dupressoir et al. [35], [36] 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. The precise relation between the two parts remains unclear.

Computational approaches. Several authors propose approaches with computational guarantees. Fournet et al. [37] specify idealized functionalities in F#, verify their properties using the F7 refinement type checker, and relate these functionalities to implementations using manually proved game-based program transformations. This approach was used to verify TLS 1.2 [38]. Backes et al. [39] present a computational soundness result, enabling the application of the symbolic approach in [32] to obtain computational guarantees. Cadé and Blanchet [13] prove the soundness of a CryptoVerif to Ocaml compiler, which preserves computational properties proved in CryptoVerif. Bhargavan et al. [30] extract a CryptoVerif (and also a ProVerif) model from a TLS 1.0 implementation written in F#. Delignat-Lavaud et al. use F* to prove the computational security of the TLS 1.3 [14] and the QUIC [15] record layer protocols. In general, while computational approaches offer stronger guarantees than symbolic ones, they are significantly harder to automate.

Message parsing. Mödersheim and Katsoris [28] show that an abstract symbolic model using message formats soundly abstracts a more concrete model, which includes associative message concatenation, variable length fields, and tags. EverParse [40] is a framework to generate secure (i.e., injective and surjective) parsers and serializers for authenticated plaintext message formats. A precise comparison of these works with our approach requires further work and might open interesting possibilities for further developments.

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. First, our framework features a Dolev-Yao attacker both in abstract and concrete models. It would be interesting to relax this assumption and allow the concrete attacker to perform bytestring operations that may not correspond to Dolev-Yao operations. Second, the current soundness result applies only if all roles are verified. For interoperable implementations, it would be interesting to explore whether it could be generalized to provide weaker guarantees if some roles are unverified (e.g., the WireGuard server in our case). Third, our approach is currently limited to trace properties. Some security properties such as privacy properties can be formalized as observational equivalences, which Tamarin also supports. How to obtain implementation-level equivalence guarantees from a symbolic model is an open problem. Finally, we plan to further automate our approach, in particular, the extraction of I/O specifications from the Tamarin model, as well as the generation of the pattern requirements.

References

[1] Anonymous, "Sound verification of security protocols: From design to interoperable implementations," Supplementary material, 2021. [Online]. Available: https://github.com/SoundVerification/wireguard

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[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] 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

[20] 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

[21] C. Cremers and M. Dehnel-Wild, "Component-Based Formal Analysis of 5G-AKA: Channel Assumptions and Session Confusion," in *NDSS*. The Internet Society, 2019.

[22] 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

[23] 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://www.usenix.org/conference/usenixsecurity20/presentation/girol

[24] 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://www.usenix.org/conference/usenixsecurity21/presentation/basin

[25] 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

[26] 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

[27] 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.

[28] 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

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

[30] 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

[31] 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

[32] 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

[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] 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

[36] ——, "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

[37] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular code-based cryptographic verification," in *ACM Conference on Computer and Communications Security*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 341–350.

[38] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 2013, pp. 445–459. [Online]. Available: https://doi.org/10.1109/SP.2013.37

[39] M. Backes, M. Maffei, and D. Unruh, "Computationally sound verification of source code," in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds. ACM, 2010, pp. 387–398. [Online]. Available: https://doi.org/10.1145/1866307.1866351

[40] 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://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

Appendix A. Protocol format details

We require that the rules' labels only contain facts from Σ_{act}, i.e. for all $\ell \xrightarrow{a} r \in \mathcal{R}$, $\mathrm{facts}(a) \subseteq \Sigma_{\mathsf{act}}$. Here, $\mathrm{facts}(s)$ denotes the set of fact symbols that occur in a multiset of facts s. In addition, protocol rules in the sets $\mathcal{R}_i$ may consume facts in Σ_{in} (but must not produce them) and may produce facts in Σ_{out} (but must not consume them).

We assume that environment rules do not directly use the agents' internal states. Namely, for all $\ell \xrightarrow{a} r \in \mathcal{R}_{\mathsf{env}}$, $\mathrm{facts}(\ell \cup r) \subseteq \Sigma_{\mathsf{env}}$. In addition, the fact Setup_i is only allowed to occur as the only fact produced by rules in $\mathcal{R}_{\mathsf{env}}$.

We also require that rules for role i only use i's state, i.e., for all $\ell \xrightarrow{a} r \in R_i$, $\mathrm{facts}(\ell) \subseteq \Sigma^i_{\mathsf{state}} \cup \Sigma_{\mathsf{in}}$ and $\mathrm{facts}(r) \subseteq \Sigma^i_{\mathsf{state}} \cup \Sigma_{\mathsf{out}}$.

Finally, we require that for a protocol rule $\ell \xrightarrow{a} r \in \mathcal{R}_i$, 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 $\ell \xrightarrow{a} r$ 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 $\overline{init}$.

Appendix B. WireGuard message construction

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

- aead is an AEAD algorithm, h is a hash function, kdf_1, kdf_2, kdf_3 are key derivation functions.
- (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.)
- 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.

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

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

Appendix C.
Proofs

In the following proofs, we use a slightly different formalisation compared to the description in Section 3. In that section, we formulated all transformation steps and lemmas in terms of multiset rewriting (MSR) systems, to make them more legible. In contrast, in this appendix, we formulate them in terms of the labelled transition systems (LTS) induced by the MSR systems. This does not change the results, as these are two equivalent descriptions of the same system; we simply use the LTS formalism here so that the steps follow more closely the general Igloo methodology from [20].

C.1. Definitions and notations

In this section we recall standard definitions for multisets, as well as the usual Tamarin model and semantics for multiset rewriting, that were introduced in Section 2.

Definition 1 (Multisets)**.** *A multiset constructed over a set S is a mapping $M : S \to \mathbb{N}$, where $M(x)$ represents the number of copies of x in M. We denote* $\mathrm{set}(M) = \{x \in S | M(x) > 0\}$ *the set associated to M. We write $x \in M$ if $M(x) > 0$. We use the notation $\{\!|\cdot|\!\}$ to define a multiset explicitly: e.g. $\{\!|a, a, b|\!\}$ denotes the multiset containing two copies of a and one of b. In rewriting rules, we alternatively use the notation $[\cdot]$ for the same purpose.*

$\cup^{\mathsf{m}}$, $\setminus^{\mathsf{m}}$ and $\subseteq^{\mathsf{m}}$ respectively denote multiset union, difference, and inclusion. That is, if A, B are two multisets, then for any element x, $(A \cup^{\mathsf{m}} B)(x) = A(x) + B(x)$, and $(A \setminus^{\mathsf{m}} B)(x) = \max(A(x) - B(x), 0)$; and $A \subseteq^{\mathsf{m}} B$ when for all x, $A(x) \leq B(x)$.

In addition, for a set S and a multiset M, we denote $M \cap^{\mathsf{m}} S$ the multiset containing only the elements from M (with the same number of copies) that are also in S, i.e. $(M \cap^{\mathsf{m}} S)(x) = M(x)$ if $x \in S$, and 0 otherwise.

Definition 2 (Terms)**.** *Consider infinite sets of names $fresh$, pub, $\mathcal{V}$, representing respectively fresh values, public values, and variables. Let Σ be a finite signature of function symbols, given with their arity.*

The set of terms *constructed from Σ, $fresh$, pub, and $\mathcal{V}$, denoted by $\mathcal{T} = \mathcal{T}_\Sigma(fresh \cup pub \mathcal{V})$, is the smallest set containing $fresh \cup pub \cup \mathcal{V}$ and closed under application of functions in Σ.*

We also let $\mathcal{G} = \mathcal{T}_\Sigma(fresh \cup pub)$ denote the set of ground terms*, i.e. terms without variables.*

Definition 3 (Equational theory)**.** *An equational theory E on $\mathcal{T}$ is a set of equations of the form $l = r$, where $l, r \in \mathcal{T}$ are terms without fresh names from $fresh$. We denote $=_{\mathsf{E}}$ equality modulo E, i.e. the smallest equivalence relation on $\mathcal{T}$ that contains E, and is stable by context and substitution.*

Definition 4 (Facts)**.** *Consider a signature $\Sigma_{\mathsf{facts}} = \Sigma_{\mathsf{lin}} \uplus \Sigma_{\mathsf{per}}$ of* fact symbols *with their arity, partitioned into* linear *facts Σ_{lin} and* persistent *facts Σ_{per}; as well as a set of terms $\mathcal{T}$ with an equational theory E.*

We write $\mathcal{F} = \{f(t_1, \ldots, t_k) \mid f \in \Sigma_{\mathsf{facts}}$ with arity $k, t_1, \ldots, t_k \in \mathcal{T}\}$ the set of facts instantiated with terms, partitioned into $\mathcal{F}_{\mathsf{lin}} \uplus \mathcal{F}_{\mathsf{per}}$ as expected.

For a set or multiset s of facts, we denote by $\mathrm{facts}(s)$ *the subset of Σ_{facts} containing all fact symbols that occur in s.*

Definition 5 (Multiset rewriting system)**.** *Consider sets of facts $\mathcal{F}$ and terms $\mathcal{T}$ with an equational theory E as defined earlier. A multiset rewriting system on these terms and facts is a set $\mathcal{R}$ of* rewriting rules *of the form $\ell \xrightarrow{a} r$, where ℓ, a, r are multisets of facts in $\mathcal{F}$. We call ℓ the premises of the rule, r the conclusions, and a the events.*

The associated transition relation $\overset{\cdot}{\Longrightarrow}_{\mathcal{R},\mathsf{E}}$ is defined by the rule

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

We will from now on only consider fact signatures containing reserved fact symbols $\mathsf{K} \in \Sigma_{\mathsf{per}}$, and $\mathsf{Fr}, \mathsf{in}, \mathsf{out} \in \Sigma_{\mathsf{lin}}$.

Definition 6 (Message deduction rules)**.** *We let MD_Σ denote the set of message deduction rules for Σ:*

$$\begin{aligned}
[\mathsf{out}(x)] &\xrightarrow{[]} [\mathsf{K}(x)] \\
[\mathsf{K}(x)] &\xrightarrow{[\mathsf{K}(x)]} [\mathsf{in}(pk)] \\
[] &\xrightarrow{[]} [\mathsf{K}(x \in pub)] \\
[\mathsf{Fr}(x \in fresh)] &\xrightarrow{[]} [\mathsf{K}(x)] \\
[\mathsf{K}(x_1), \ldots, \mathsf{K}(x_k)] &\xrightarrow{[]} [\mathsf{K}(f(x_1, \ldots, x_k))] \qquad \text{for } f \in \Sigma \text{ with arity } k
\end{aligned}$$

Definition 7 (Freshness rule)**.** *We let Fresh denote the freshness generation rule:*

$$[] \xrightarrow{[\mathsf{Fresh}(x)]} [\mathsf{Fr}(x \in fresh)]$$

Definition 8 (Traces)**.** *The set of traces of a multiset rewriting system $\mathcal{R}$ (for facts $\mathcal{F}$, terms $\mathcal{T}$, and equations E) is*

$$\mathrm{Tr}(\mathcal{R}) = \{\langle a_1, \ldots, a_m\rangle \mid \exists s_1, \ldots, s_m \text{ ground multisets of facts.} \\ \emptyset \overset{a_1}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} s_1 \overset{a_2}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} \cdots \overset{a_m}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} s_m\}.$$

The set of filtered traces, without empty labels, is

$$\mathrm{Tr}'(\mathcal{R}) = \{\langle a_i\rangle_{1 \le i \le m, a_i \neq \emptyset} | \langle a_1, \ldots, a_m\rangle \in \mathrm{Tr}(\mathcal{R})\}.$$

The set of Tamarin traces, additionally removing collisions in the random generation, is

$$\mathrm{Tr}_{\mathrm{t}}(\mathcal{R}) = \mathrm{Tr}'(\mathcal{R}) \setminus \mathcal{C},$$

where $\mathcal{C} = \{\langle a_1, \ldots, a_m\rangle \mid \exists i, j.\ i \neq j \wedge \mathsf{Fr}(n) \in a_i \cap^{\mathsf{m}} a_j\}$.

Note that we use the notation $\langle x_1, \ldots, x_n\rangle$, or alternatively $\langle x_i\rangle_{1 \le i \le n}$, to denote the ordered sequence containing the elements $x_1, \ldots, x_n$.

C.2. Assumptions

We now recall in detail the formatting assumptions introduced in Section 3.1.

We fix sets of names pub, $fresh$, $\mathcal{V}$, a signature Σ, and the set of terms $\mathcal{T}$ constructed on them, with an equational theory E. Let us also fix $n \geq 1$, the number of roles in the system we consider.

We consider a fact signature of the form

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

for some arbitrary (but disjoint) sets of facts symbols $\Sigma_{\mathsf{env}}, \Sigma_{\mathsf{act}}, \Sigma^i_{\mathsf{state}}$ for $1 \leq i \leq n$, intended to represent respectively environment facts, the action facts, and each role's internal states. Note that each of these sets may contain linear and persistent facts. We additionally assume that Σ_{env} contains subsets Σ_{in}, Σ_{out} which we call input and output fact symbols, with $\mathsf{Fr}, \mathsf{in} \in \Sigma_{\mathsf{in}}$, $\mathsf{out} \in \Sigma_{\mathsf{out}}$, and $\mathsf{K} \in \Sigma_{\mathsf{env}} \setminus (\Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}})$. Furthermore, we assume that there is an initialization fact symbol $\mathsf{Setup}_i \in \Sigma_{\mathsf{in}}$ for each protocol role i.

We then consider a multiset rewriting system $\mathcal{R}$ of the form

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

where $\mathcal{R}_i$ and $\mathcal{R}_{\mathsf{env}}$ are arbitrary (disjoint) sets of rules intended to represent respectively each role and environment rules, and $MD_\Sigma \cup \{Fresh\} \subseteq \mathcal{R}_{\mathsf{env}}$.

We assume that for all rule $\ell \xrightarrow{a} r \in \mathcal{R}$, $\mathrm{facts}(a) \subseteq \Sigma_{\mathsf{act}}$. For all environment rule $\ell \xrightarrow{a} r \in \mathcal{R}_{\mathsf{env}}$, we assume that $\mathrm{facts}(\ell \cup r) \subseteq \Sigma_{\mathsf{env}}$. In addition, the fact Setup_i is only allowed to occur as the only fact produced by rules in $\mathcal{R}_{\mathsf{env}}$. For all i, for all $\ell \xrightarrow{a} r \in \mathcal{R}_i$, we assume:

- $\mathrm{facts}(\ell) \subseteq \Sigma_{\mathsf{in}} \cup \Sigma^i_{\mathsf{state}}$;
- $\mathrm{facts}(r) \subseteq \Sigma_{\mathsf{out}} \cup \Sigma^i_{\mathsf{state}}$;
- at least one state fact appears in r;
- the first $k_i \geq 1$ arguments of all state facts in $\ell \xrightarrow{a} r$, as well as the Setup_i fact, are reserved for role i's parameters, i.e. have the same value in all state facts and all occurrences of Setup_i in the rule. In addition, the first of these k parameters must be a thread identifier, i.e. a value of $fresh$ called rid. These k parameters are thus never changed once the Setup_i fact is produced to start a run of role i. For readability, we will group all these initial parameters as a tuple, denoted by $\overline{init}$.

For each $rid \in fresh$, and each i, we denote $\mathcal{R}_{i,rid}$ the set of rules in $\mathcal{R}_i$ where the first argument of all state facts is instantiated with rid.

C.3. Step 1: Interface model

As stated in Section 3, our goal will be to separate the multiset rewriting system into several transition systems, representing each component. We first introduce interfaces between each component (i.e., each role, and the environment), to make this separation easier. These interfaces are additional facts, which we call *buffer* facts. We add buffers for all operations we wish to consider as I/O, i.e. the input and output facts. These take the form of additional rules, called *I/O rules*, that transform each such fact into a buffered version (for Σ_{in}) or vice versa (for Σ_{out}).

We first extend the fact signature by adding, for each input or output fact F, a "buffered" copy F_i for each role i. We define

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

We then replace the facts used by the protocol rules as follows. Let Σ^-_{in} be the set of input facts without the role initialisation facts Setup_i. 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 $F(t_1, \ldots, t_k)$ (with $F \in \Sigma^-_{\mathsf{in}} \cup \Sigma_{\mathsf{out}}$) with $F_i(rid, t_1, \ldots, t_k)$, where rid is the thread id parameter present in the state facts in the rule.

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.

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

We also count the role setup rules, which generate the Setup_i facts, as I/O rules. Hence, we move them from the set $\mathcal{R}_{\mathsf{env}}$ to $\mathcal{R}_{\mathsf{io}}$, calling the set of remaining environment rules $\mathcal{R}^-_{\mathsf{env}}$.

We then consider the system

$$\mathcal{R}_{\mathsf{intf}} = \mathcal{R}^-_{\mathsf{env}} \uplus \mathcal{R}_{\mathsf{io}} \uplus (\biguplus_{1 \le i \le n} \mathcal{R}'_i).$$

We can now prove the following lemma, which corresponds to Lemma 1 in Section 3.2.1.

Lemma 3.

$$\mathrm{Tr}'(\mathcal{R}_{\mathsf{intf}}) \subseteq \mathrm{Tr}'(\mathcal{R})$$

Proof. We prove this inclusion by establishing a refinement, using a simulation relation $\mathscr{R}$. That is, we show that

(1) $(\emptyset, \emptyset) \in \mathscr{R}$;

(2) for all states $(s_1, s'_1) \in \mathscr{R}$, for all transition steps $s'_1 \overset{a'}{\Longrightarrow}_{\mathcal{R}_{\mathsf{intf}},\mathsf{E}} s'_2$ there exists a sequence of transitions $s_1 \overset{a_1}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} \ldots \overset{a_m}{\Longrightarrow}_{\mathcal{R},\mathsf{E}} s_m$ such that $(s_m, s'_2) \in \mathscr{R}$, and $\langle a_i \rangle_{1 \le i \le m, a_i \neq \emptyset} = \langle a' \rangle$. In other words, there exists a sequence of transitions that reaches a state related to s'_2, and produces a sequence of m actions, among which one is equal to a', while the others are empty. In the proof we will actually only have $m \in \{0, 1\}$.

We use the relation $\mathscr{R}$ such that $(s, s') \in \mathscr{R}$ if and only if s is the state obtained from s' by removing the indices i from all facts, as well as the first argument rid added to buffered facts. The first point, $(\emptyset, \emptyset) \in \mathscr{R}$, is clear.

Let $(s_1, s'_1) \in \mathscr{R}$, and consider a step $s'_1 \overset{a}{\Longrightarrow}_{\mathcal{R}_{\mathsf{intf}},\mathsf{E}} s'_2$. We can distinguish several cases for the rule in $\mathcal{R}_{\mathsf{intf}}$ of which it is an instance.

- if it is a rule $\ell' \xrightarrow{a'} r' \in \mathcal{R}'_i$: the transition is thus

  $$s'_1 \overset{a'\theta}{\Longrightarrow}_{\mathcal{R}_{\mathsf{intf}},\mathsf{E}} s'_2 = s'_1 \setminus^{\mathsf{m}} (\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'\theta$$

  for some ground θ such that $\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} s'_1$ and $\mathrm{set}(\ell'\theta) \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(s'_1)$. Let rid be the name with which θ instantiates the first argument (thread id) of the rule's state facts and buffered facts. By construction, there exists a rule $\ell \xrightarrow{a} r \in \mathcal{R}_i$ that can be

obtained from $\ell' \xrightarrow{a'} r'$ by replacing each symbol F_i with the unlabelled symbol F, and removing its first argument (which is rid). Note that, by assumption, a does not contain fact symbols in Σ_{env}, and thus $a = a'$.
Since $(s_1, s_1') \in \mathcal{R}$ and $\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} s_1'$, we have $\ell\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} s_1$. For similar reasons, we also have $\mathrm{set}(\ell\theta) \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(s_1)$. Hence, by applying rule $(\ell \xrightarrow{a} r)\theta$, we have

$$s_1 \xRightarrow{a\theta}_{\mathcal{R},\mathsf{E}} s_1 \setminus^{\mathsf{m}} (\ell\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r\theta.$$

Since $(s_1, s_1') \in \mathcal{R}$, we also have $(s_1 \setminus^{\mathsf{m}} (\ell\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r\theta, s_1' \setminus^{\mathsf{m}} (\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'\theta) \in \mathcal{R}$.

- if it is a rule $\ell' \xrightarrow{a'} r' \in \mathcal{R}^-_{\mathsf{env}}$: the transition is thus

$$s_1' \xRightarrow{a'\theta}_{\mathcal{R}_{\mathsf{intf}},\mathsf{E}} s_2' = s_1' \setminus^{\mathsf{m}} (\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'\theta$$

for some ground θ such that $\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} s_1'$ and $\mathrm{set}(\ell'\theta) \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(s_1')$. By construction, $\ell' \xrightarrow{a'} r' \in \mathcal{R}_{\mathsf{env}}$. By the formatting assumptions, $\mathrm{facts}(\ell' \cup r') \subseteq \Sigma_{\mathsf{env}}$. Hence, the facts in $\ell'\theta$ and $r'\theta$ are not touched by $\mathcal{R}$. Thus, since $(s_1, s_1') \in \mathcal{R}$ and $\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} s_1'$, we have $\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} s_1$. Similarly, $\mathrm{set}(\ell'\theta) \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(s_1)$. Thus, by applying rule $\ell' \xrightarrow{a'} r'$, we have

$$s_1 \xRightarrow{a'\theta}_{\mathcal{R},\mathsf{E}} s_1 \setminus^{\mathsf{m}} (\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'\theta.$$

Since $(s_1, s_1') \in \mathcal{R}$ and $\mathrm{facts}(\ell' \cup r') \subseteq \Sigma_{\mathsf{env}}$, we also have $(s_1 \setminus^{\mathsf{m}} (\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'\theta, s_1' \setminus^{\mathsf{m}} (\ell'\theta \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'\theta) \in \mathcal{R}$.

- Finally, the case of I/O rules remains. We write the proof for the case of an input rule in Σ_{in}, as the output case is similar, and the setup case is similar to the previous case ($\mathcal{R}^-_{\mathsf{env}}$). Assume the transition rule applied is $[F(x_1, \ldots, x_k)] \xrightarrow{[]} [F_i(rid, x_1, \ldots, x_k)]$. In that case the transition is

$$s_1' = s' \cup^{\mathsf{m}} \{\!|F(M_1, \ldots, M_k)|\!\} \xRightarrow{\emptyset}_{\mathcal{R}_{\mathsf{intf}},\mathsf{E}} s_2' = s' \cup^{\mathsf{m}} \{\!|F_i(rid, M_1, \ldots, M_k)|\!\}$$

for some s', $M_1, \ldots, M_k$, rid. Since $(s_1, s_1') \in \mathcal{R}$, there exists s such that $s_1 = s \cup^{\mathsf{m}} \{\!|F(M_1, \ldots, M_k)|\!\}$ and $(s, s') \in \mathcal{R}$. Hence we also have $(s_1, s_2') \in \mathcal{R}$, and no transition needs to be performed on s_1.

□

C.4. Step 2: express the MSR system as an event system

Before we separate our monolithic system into the composition of several component systems, we switch to the formalism of guarded event systems, which will be more appropriate when extracting an I/O specification.

We associate to $\mathcal{R}_{\mathsf{intf}}$ a guarded transition system $\mathcal{E} = (\mathcal{S}, \mathcal{E}, \mathcal{G}, \mathcal{U})$:

- $\mathcal{S}$ is the set of multisets of ground facts from the signature Σ'_{facts};
- $\mathcal{E} = (\bigcup_{1 \le i \le n, \text{ for each } rid} \mathcal{E}^s_{i,rid}) \cup \mathcal{E}^e \cup \mathcal{E}^{\mathsf{io}}$ with

$$\mathcal{E}^s_{i,rid} = \{\mathsf{skip}\} \cup \{e^s_{i,rid,j}(\theta, \ell', a', r') \mid \begin{array}{l} \ell', a', r' \text{ multisets of ground facts,} \\ \theta \text{ ground inst. of } \mathrm{vars}(\ell) \cup \mathrm{vars}(a) \cup \mathrm{vars}(r) \\ \text{where } \ell \xrightarrow{a} r \text{ is the } j\text{th rule in } \mathcal{R}'_i \text{ with the first} \\ \text{parameter of all state and setup facts} \\ \text{instantiated with } rid\} \end{array}$$

and

$$\mathcal{E}^e = \{\mathsf{skip}\} \cup \{e^e_j(\theta, \ell', \mathscr{a}', \mathscr{r}') \mid \begin{array}{l}\ell', \mathscr{a}', \mathscr{r}' \text{ multisets of ground facts,} \\ \theta \text{ ground inst. of } \mathrm{vars}(\ell) \cup \mathrm{vars}(\mathscr{a}) \cup \mathrm{vars}(\mathscr{r}) \\ \text{where } \ell \xrightarrow{\mathscr{a}} \mathscr{r} \text{ is the } j\text{th rule in } \mathcal{R}^-_{\mathsf{env}}\}\end{array}$$

and

$$\mathcal{E}^{\mathsf{io}} = \{\lambda_{F,i,rid}(\theta, \ell', \mathscr{a}', \mathscr{r}') \mid \begin{array}{l}F \in \Sigma_{\mathsf{in}} \cup \Sigma_{\mathsf{out}} \\ \ell', \mathscr{a}', \mathscr{r}' \text{ multisets of ground facts,} \\ \theta \text{ ground inst. of } \mathrm{vars}(\ell) \cup \mathrm{vars}(\mathscr{a}) \cup \mathrm{vars}(\mathscr{r}) \\ \text{where } \ell \xrightarrow{\mathscr{a}} \mathscr{r} \text{ is the rule associated to fact } F \text{ and} \\ \text{role } i \text{ in } \mathcal{R}_{\mathsf{io}}, \text{ with its first parameter instantiated with } rid\}\end{array}$$

- $\mathcal{G}$: the guard for a parametric event $e(\theta, \ell', \mathscr{a}', \mathscr{r}')$ associated to a rule $\ell \xrightarrow{\mathscr{a}} \mathscr{r} \in \mathcal{R}'$, in the current state s, is

$$\ell' =_{\mathsf{E}} \ell\theta \ \wedge \ \mathscr{a}' =_{\mathsf{E}} \mathscr{a}\theta \ \wedge \ \mathscr{r}' =_{\mathsf{E}} \mathscr{r}\theta \ \wedge \ G(\ell', s)$$

  where

$$G(\ell', s) = (\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} s) \ \wedge \ (\mathrm{set}(\ell') \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(s)).$$

- $\mathcal{U}$: a parametric event $e(\theta, \ell', \mathscr{a}', \mathscr{r}')$ associated to a rule $\ell \xrightarrow{\mathscr{a}} \mathscr{r} \in \mathcal{R}'$ updates the state s to $s' = U(\ell', \mathscr{r}', s)$, where

$$U(\ell', \mathscr{r}', s) = s \setminus^{\mathsf{m}} (\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} \mathscr{r}'.$$

Note that the event label $\lambda_{F,i,rid}(\theta, \ell', \mathscr{a}', \mathscr{r}')$ we assign to events associated with rules in $\mathcal{R}_{\mathsf{io}}$ corresponds to the synchronisation label $\lambda_F(\overline{x})$ described in Section 3.2.2. Since we use the LTS formalism here, it is now an event name, rather than a fact annotating a rewrite rule. We include more parameters here compared to that section to make the form of all our events uniform – even though, since the form of the rules in $\mathcal{R}_{\mathsf{io}}$ is known, by construction, it would be sufficient to only give the instantiation of each parameter of F, as was done in the paper.

We let $\xrightarrow{\cdot}$ denote the reduction relation defined by checking the guards and applying the updates for each event in $\mathcal{E}$. Formally, $s \xrightarrow{E} s'$ if E is an event $e(\theta, \ell', \mathscr{a}', \mathscr{r}')$ associated to a rule $\ell \xrightarrow{\mathscr{a}} \mathscr{r} \in \mathcal{R}_{\mathsf{intf}}$, such that $\ell' =_{\mathsf{E}} \ell\theta \ \wedge \ \mathscr{a}' =_{\mathsf{E}} \mathscr{a}\theta \ \wedge \ \mathscr{r}' =_{\mathsf{E}} \mathscr{r}\theta \ \wedge \ G(\ell', s)$, and $s' = U(\ell', \mathscr{r}', s)$.

Let π denote the mapping from $\mathcal{E}$ to multisets of facts, that associates to each parametric event $e(\theta, \ell', \mathscr{a}', \mathscr{r}') \in \mathcal{E}$ the multiset of facts $\mathscr{a}'$.

We denote $\tilde{\pi}$ the extension of π to sequences of events and multisets, which applies π and removes empty multisets of facts, i.e. for any sequence $e = \langle e_1, \ldots, e_n \rangle$ of events,

$$\tilde{\pi}(e) = \langle \pi(e_i) \mid i = 1, \ldots, n \ \wedge \ \pi(e_i) \neq \emptyset \rangle.$$

Note that Section 3.3 introduced the same mapping, but in the MSR formalism.

The traces of the event system are defined as

$$\mathrm{Tr}(\mathscr{E}) = \{e_1, \ldots, e_m \mid (\forall i.\ e_i \in \mathcal{E}) \ \wedge \ \exists s_1, \ldots, s_m \in \mathcal{S}.\ \emptyset \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_m} s_m\}.$$

Lemma 4.

$$\tilde{\pi}(\mathrm{Tr}(\mathscr{E})) = \mathrm{Tr}'(\mathcal{R}_{\mathsf{intf}})$$

Proof. We show this lemma by proving that for any state s:

(a) for any event $e \in \mathcal{E}$ and any s', if $s \xrightarrow{e} s'$ then $s \overset{\pi(e)}{\Longrightarrow}_{\mathcal{R}_{\mathsf{intf}}, \mathsf{E}} s'$

(b) for any reduction step $s \overset{a}{\Longrightarrow}_{\mathcal{R}_{\mathsf{intf}},\mathsf{E}} s'$, there exists e such that $s \xrightarrow{e} s'$ and $\pi(e) = a$.

Once these two properties are established, the lemma follows easily by applying them successively to each reduction step – (a) proves the first inclusion ($\subseteq$) and (b) proves the second one ($\supseteq$).

We first show (a). Consider states s, s' and an event $e \in \mathcal{E}$ such that $s \xrightarrow{e} s'$. By construction of $\mathcal{E}$, there exist e', θ, ℓ', a', r' such that $e = e'(\theta, \ell', a', r')$. In addition, e is associated to some rule $\ell \xrightarrow{a} r \in \mathcal{R}_{\mathsf{intf}}$. By definition of the guard associated to e, we have $(\ell', a', r') =_{\mathsf{E}} (\ell, a, r)\theta$, $\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} s$, and $s' = \mathrm{set}(\ell') \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(s)$. By definition of the update associated to e, we have $s' = s \setminus^{\mathsf{m}} (\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}}) \cup^{\mathsf{m}} r'$. Thus by multiset rewriting, $s \overset{a'}{\Longrightarrow}_{\mathcal{R}_{\mathsf{intf}},\mathsf{E}} s'$. Finally, by definition of π we have $\pi(e) = a'$, which proves (a).

Let us now show (b). Consider a reduction step $s \overset{a}{\Longrightarrow}_{\mathcal{R}_{\mathsf{intf}},\mathsf{E}} s'$. By definition of multiset rewriting, there exists a rule $\ell \xrightarrow{a} r \in \mathcal{R}_{\mathsf{intf}}$, a ground instantiation θ of $\mathrm{vars}(\ell) \cup \mathrm{vars}(a) \cup \mathrm{vars}(r)$, and multisets of ground facts ℓ', a', r' such that $(\ell', a', r') =_{\mathsf{E}} (\ell, a, r)\theta$, $\ell' \cap^{\mathsf{m}} \mathcal{F}_{\mathsf{lin}} \subseteq^{\mathsf{m}} s$, $\mathrm{set}(\ell') \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(s)$, and $s' = \mathrm{set}(\ell') \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(s)$.

By construction of $\mathcal{R}_{\mathsf{intf}}$, $\ell \xrightarrow{a} r$ is either in $\mathcal{R}'_i$ for some i, in $\mathcal{R}^-_{\mathsf{env}}$, or in $\mathcal{R}_{\mathsf{io}}$. In either case, there exists a parametric event e such that the family of events

$$\{e(\theta'', \ell'', a'', r'') \mid \theta'' \text{ gr. inst. of } \mathrm{vars}(\ell) \cup \mathrm{vars}(a) \cup \mathrm{vars}(r), \ell'', a'', r'' \text{ multisets of gr. facts}\}$$

is in $\mathcal{E}$. The three cases differ only by the event name, which is inconsequential here.

Consider the instance $e(\theta, \ell', a', r')$: its guard is satisfied, and it updates s into $\mathrm{set}(\ell') \cap \mathcal{F}_{\mathsf{per}} \subseteq \mathrm{set}(s)$, i.e. s'. Hence $s \xrightarrow{e(\theta,\ell',a',r')} s'$. Since $\pi(e(\theta, \ell', a', r')) = a'$, this proves (b). □

From now on, we will actually slightly strengthen the guards of the protocol events in $\mathcal{E}$, i.e. those in $\mathcal{E}^s_{i,rid}$, to use true equality instead of equality modulo E. This only reduces the set of possible executions, and thus preserves the previous trace inclusion.

C.5. Step 3: separating the event system into each component

We are now ready to split the event system into its components.

We will separate it into one system $\mathcal{C}^s_{i,rid}$ for each role instance, and a system $\mathcal{C}^e$ for the environment. The interface events, from $\mathcal{E}^{\mathsf{io}}$ (associated to rule in $\mathcal{R}_{\mathsf{io}}$), will be split in two halves, one to be added to $\mathcal{C}^e$, and the other to one of the $\mathcal{C}^s_{i,rid}$.

For each role instance (i, rid) we consider the event system $\mathcal{C}^s_{i,rid}$ containing the events in $\mathcal{E}^s_{i,rid}$ and the associated guards and updates, as well as events, guards and updates encoding the additional rules

$$\begin{array}{rcll} [F_i(rid, \overline{x})] & \xrightarrow{[]} & [] & \text{for } F \in \Sigma_{\mathsf{out}} \\ [] & \xrightarrow{[]} & [F_i(rid, \overline{x})] & \text{for } F \in \Sigma_{\mathsf{in}} \end{array}$$

We denote the events for these transitions $\lambda^s_{F,i,rid}$. Formally:

- if $F \in \Sigma_{\mathsf{out}}$, the guard for $\lambda^s_{F,i,rid}(\overline{x})$ requires that the state contains $F_i(rid, \overline{x})$, and its update simply removes that fact from the state or leaves it, depending on whether $F \in \mathcal{F}_{\mathsf{lin}}$ or not;
- if $F \in \Sigma_{\mathsf{in}}$, the guard for $\lambda^s_{F,i,rid}(\overline{x})$ is always satisfied, and its update adds fact $F_i(rid, \overline{x})$ to the state.

We also consider the event system $\mathscr{E}^e$ containing the events in $\mathcal{E}^e$ and the associated guards and updates, augmented with events encoding the rules

$$\begin{array}{rcll} [F(\overline{x})] & \xrightarrow{[]} & [] & \text{for } F \in \Sigma^-_{\mathsf{in}} \\ [\overline{F'(\overline{x})}] & \xrightarrow{[]} & [] & \text{for } F = \mathsf{Setup}_i \text{ for some } i \\ & & & \text{where } \overline{F'(\overline{x})} \text{ are the premises of the setup rule for } F \\ [] & \xrightarrow{[]} & [F(\overline{x})] & \text{for } F \in \Sigma_{\mathsf{out}}. \end{array}$$

We denote the events associated to these rules $\lambda^e_{F,i,rid}(\overline{x})$ (i.e. we include one copy of the event for each i, rid). Formally:

- if $F \in \Sigma^-_{\mathsf{in}}$, the guard for $\lambda^e_{F,i,rid}(\overline{x})$ requires that the state contains $F(\overline{x})$, and its update either removes this fact from the state or leaves it, depending on whether $F \in \mathcal{F}_{\mathsf{lin}}$ or not;
- if $F = \mathsf{Setup}_i$, the guard for $\lambda^e_{F,i,rid}(\overline{x})$ requires that the state contains $[\overline{F'(\overline{x})}]$, and its update removes from the state the facts in this list that are in $\mathcal{F}_{\mathsf{lin}}$;
- if $F \in \Sigma_{\mathsf{out}}$, the guard for $\lambda^e_{F,i,rid}(\overline{x})$ is always satisfied, and its update adds $F(\overline{x})$ to the state.

We define the partial function $\chi : (\bigcup_{i,rid} \mathscr{E}^s_{i,rid}) \times \mathscr{E}^e \to \mathcal{E}$ (conflating $\mathscr{E}^e$ with its state space, and similarly for $\mathscr{E}^s$) that synchronises labels λ_F, i.e.

- $\chi(\lambda^s_{F,i,rid}(\overline{m}), \lambda^e_{F,i,rid}(\overline{x})) = \lambda_{F,i,rid}([\overline{x} \mapsto \overline{m}], \ell', a', r')$, where $\ell \xrightarrow{a} r$ is the rule associated with F in $\mathcal{R}_{\mathsf{io}}$, instantiated with rid, $\overline{x}$ are its variables, and $\ell' \xrightarrow{a'} r'$ is its instantiation with $\overline{m}$;
- $\chi(\mathsf{skip}, e) = e$ when e is not of the form $\lambda^s_{F,i,rid}$;
- $\chi(e, \mathsf{skip}) = e$ when e is not of the form $\lambda^e_{F,i,rid}$.
- $\chi(e, e')$ is undefined in all other cases.

We introduce two composition operators. $|||$ is parallel composition, i.e. the state space of $A|||B$ is the product of those of A and B, and its events are e_A updating state (s_A, s_B) to (s'_A, s_B) if e_A updates s_A to s'_A in A, and similarly for e_B in B. $||_\chi$ is synchronising composition, and is defined similarly: the state space of $A||_\chi B$ is the product of those of A and B, and its events are $\chi(e_A, e_B)$, updating state (s_A, s_B) to (s'_A, s'_B), if e_A updates s_A to s'_A in A, and similarly for e_B in B.

We can then state the following composition lemma, which corresponds to Lemma 2 in Section 3.2.2 (when chaining it by transitivity with Lemma 4).

Lemma 5.

$$\mathrm{Tr}((|||_{i,rid} \mathscr{E}^s_{i,rid}) ||_\chi \mathscr{E}^e) \subseteq \mathrm{Tr}(\mathscr{E}).$$

Proof. We prove this lemma by establishing a refinement with a simulation relation $\mathscr{R}$ between abstract states (from $\mathscr{E}$) and concrete ones (from $(|||_i \mathscr{E}^s_{i,rid}) ||_\chi \mathscr{E}^e$). The concrete states are of the form $((s_{i,rid})_{1 \le i \le n, \text{ for each } rid}, s_e)$, i.e. they are composed of one multiset of facts for each i, rid, and one for the environment.

The refinement relation we use is defined by $(s, s') \in \mathscr{R}$ iff $s = (\cup^{\mathsf{m}}_{i,rid} s'_{i,rid}) \cup^{\mathsf{m}} s'_e$, where $s' = ((s'_{i,rid}), s'_e)$.

For better legibility we will denote $\mathscr{E}' = (|||_i \mathscr{E}^s_{i,rid}) ||_\chi \mathscr{E}^e$ the composed system, and $\xrightarrow{\cdot}_{\mathscr{E}'}$, $\xrightarrow{\cdot}_{\mathscr{E}}$, $\xrightarrow{\cdot}_{i,rid}$, $\xrightarrow{\cdot}_e$ the transition relations defined respectively by $\mathscr{E}'$, $\mathscr{E}$, $\mathscr{E}^s_{i,rid}$ and $\mathscr{E}^e$.

It is clear that the initial states of $\mathscr{E}'$ and $\mathscr{E}$ are related: $(((\emptyset, \dots, \emptyset), \emptyset), \emptyset) \in \mathscr{R}$. We now show that for all states $(s_1, s'_1) \in \mathscr{R}$, for all transition steps $s'_1 \xrightarrow{e}_{\mathscr{E}'} s'_2$ there exists a transition

$s_1 \xrightarrow{e}_{\mathscr{C}} s_2$ such that $(s_2, s'_2) \in \mathscr{R}$. We will denote $s'_j = ((s'_{j,i,rid})_{1 \leq i \leq n, \text{ for each } rid}, s'_{j,e})$ for $j \in \{1, 2\}$.

Following the definition of χ, we can distinguish several cases for the transition step $s'_1 \xrightarrow{e}_{\mathscr{C}'} s'_2$.

- if $e = \chi(e', \mathsf{skip})$ for some $e' \in \mathcal{E}^s_{i,rid}$: then $e' = e$, and by definition of $||_\chi$ and $|||$, since $s'_1 \xrightarrow{e}_{\mathscr{C}'} s'_2$, we have $s'_{1,i,rid} \xrightarrow{e}_{i,rid} s'_{2,i,rid}$, $s'_{2,j,rid'} = s'_{1,j,rid'}$ for all $(j, rid') \neq (i, rid)$, and $s'_{2,e} = s'_{1,e}$.
  By definition of $\mathscr{C}^s_{i,rid}$, the guard and update of e in that system are the same as in $\mathscr{C}$. In addition, since $(s'_1, s_1) \in \mathscr{R}$, we have $s'_{1,i,rid} \subseteq^{\mathsf{m}} s_1$. It is immediate from the form of the guard in $\mathscr{C}$ that it is stable by supermultiset, and thus holds for s_1. It is also clear from the form of the update in $\mathscr{C}$ that applying it to the whole state is exactly the same as applying it to a submultiset of the state that satisfies the guard, here $s'_{1,i,rid}$, and leaving the rest of the state untouched.
  Therefore, we have
  $$s_1 = (\cup^{\mathsf{m}}_{j,rid'} s'_{1,j,rid'}) \cup^{\mathsf{m}} s'_{1,e} \xrightarrow{e}_{\mathscr{C}} (\cup^{\mathsf{m}}_{(j,rid') \neq (i,rid)} s'_{1,j,rid'}) \cup^{\mathsf{m}} s'_{2,i,rid} \cup^{\mathsf{m}} s'_{1,e}.$$
  As noted earlier, we have $s'_{2,e} = s'_{1,e}$ and $s'_{2,j,rid'} = s'_{1,j,rid'}$ for all $(j, rid') \neq (i, rid)$, hence $(s'_2, (\cup^{\mathsf{m}}_{(j,rid') \neq (i,rid)} s'_{1,j,rid'}) \cup^{\mathsf{m}} s'_{2,i,rid} \cup^{\mathsf{m}} s'_{1,e}) \in \mathscr{R}$, which concludes the proof in this case.
- if $e = \chi(\mathsf{skip}, e')$ for some $e' \in \mathcal{E}^e$: this case is similar to the previous one.
- The remaining case is the synchronisation case, where $e = \chi(\lambda^s_{F,i,rid}(\overline{m}), \lambda^e_{F,i,rid}(\overline{m}))$ for some $F, i, rid, \overline{m}$.
  Then $e = \lambda_{F,i,rid}(\theta_m, \ell', \mathscr{a}', \mathscr{r}')$ where $\theta_m = [\overline{x} \mapsto \overline{m}]$, $\ell \xrightarrow{\mathscr{a}} \mathscr{r}$ is the rule associated with F in $\mathcal{R}_{\mathsf{io}}$, instantiated with rid, $\overline{x}$ are its variables, and $\ell' \xrightarrow{\mathscr{a}'} \mathscr{r}'$ is its instantiation with θ_m.
  By definition of $||_\chi$ and $|||$, since $s'_1 \xrightarrow{e}_{\mathscr{C}'} s'_2$, we have $s'_{1,i,rid} \xrightarrow{\lambda^s_{F,i,rid}(\overline{m})}_{i,rid} s'_{2,i,rid}$, $s'_{1,e} \xrightarrow{e^{e,in}(m)}_e s'_{2,e}$, and $s'_{2,j,rid'} = s'_{1,j,rid'}$ for all $(j, rid') \neq (i, rid)$.
  We now need to distinguish the cases where $F \in \Sigma^-_{\mathsf{in}}$, $F = \mathit{Setup}_i$, and $F \in \Sigma_{\mathsf{out}}$. We write the proof for the case $F \in \Sigma_{\mathsf{out}}$ and $F \in \mathcal{F}_{\mathsf{lin}}$, as the other cases are similar.
  We then have $\ell' = \{\!|F_i(rid, \overline{m})|\!\}$, $\mathscr{a}' = []$, and $\mathscr{r}' = \{\!|F(\overline{m})|\!\}$.
  By definition of $\lambda^s_{F,i,rid}(\overline{m})$, we have $F_i(rid, \overline{m}) \in s'_{2,i,rid} = s'_{1,i,rid}$, and $s'_{1,i,rid} \setminus^{\mathsf{m}} \{\!|F_i(rid, \overline{m})|\!\}$.
  By definition of $\lambda^e_{F,i,rid}(\overline{m})$, we have $s'_{2,e} = s'_{1,e} \cup^{\mathsf{m}} \{\!|F(\overline{m})|\!\}$. Therefore, $s_1 = (\cup^{\mathsf{m}}_{k,rid'} s'_{1,k,rid'}) \cup^{\mathsf{m}} s'_{1,e}$ satisfies the guard of the event $\lambda_{F,i,rid}(\theta_m, \{\!|F_i(rid, \overline{m})|\!\}, \emptyset, \{\!|F(\overline{m})|\!\})$ associated with rule $[F_i(rid, \overline{m})] \to [F(\overline{m})]$ in $\mathscr{C}$. Hence we have
  $$s_1 \xrightarrow{\lambda_{F,i,rid}(\theta_m, \{\!|F_i(rid,\overline{m})|\!\}, \emptyset, \{\!|F(\overline{m})|\!\})}_{\mathscr{C}} ((\cup^{\mathsf{m}}_{(k,rid')} s'_{1,k,rid'}) \cup^{\mathsf{m}} s'_{1,e} \setminus^{\mathsf{m}} \{\!|F_i(rid, \overline{m})|\!\}) \cup^{\mathsf{m}} \{\!|F(\overline{m})|\!\}.$$
  As noted earlier, we have $s'_{2,i,rid} = s'_{1,i,rid} \cup^{\mathsf{m}} \{\!|F(\overline{m})|\!\}$, $s'_{2,e} = s'_{1,e} \setminus^{\mathsf{m}} \{\!|F_i(rid, \overline{m})|\!\}$, and $s'_{2,k,rid'} = s'_{1,k,rid'}$ for all $(k, rid') \neq (i, rid)$. Thus $(s'_2, ((\cup^{\mathsf{m}}_{(k,rid')} s'_{1,k,rid'}) \cup^{\mathsf{m}} s'_{1,e} \setminus^{\mathsf{m}} \{\!|F_i(rid, \overline{m})|\!\}) \cup^{\mathsf{m}} \{\!|F(\overline{m})|\!\}) \in \mathscr{R}$, which concludes the proof in this case.

□