A Generic Methodology for the Modular Verification of Security Protocol Implementations (extended version)

Linard Arquint
Department of Computer Science
ETH Zurich, Switzerland

Malte Schwerhoff
Department of Computer Science
ETH Zurich, Switzerland

Vaibhav Mehta*
Cornell University
Ithaca, NY, USA

Peter Müller
Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Security protocols are essential building blocks of modern IT systems. Subtle flaws in their design or implementation may compromise the security of entire systems. It is, thus, important to prove the absence of such flaws through formal verification. Much existing work focuses on the verification of protocol *models*, which is not sufficient to show that their *implementations* are actually secure. Verification techniques for protocol implementations (e.g., via code generation or model extraction) typically impose severe restrictions on the used programming language and code design, which may lead to sub-optimal implementations. In this paper, we present a methodology for the modular verification of strong security properties directly on the level of the protocol implementations. Our methodology leverages state-of-the-art verification logics and tools to support a wide range of implementations and programming languages. We demonstrate its effectiveness by verifying memory safety and security of Go implementations of the Needham-Schroeder-Lowe, Diffie-Hellman key exchange, and WireGuard protocols, including forward secrecy and injective agreement for WireGuard. We also show that our methodology is agnostic to a particular language or program verifier with a prototype implementation for C.

CCS CONCEPTS

• **Security and privacy → Logic and verification**.

KEYWORDS

Protocol implementation verification, Symbolic security, Separation logic, Automated verification, Injective agreement, Forward secrecy.

1 INTRODUCTION

Cryptographic protocols, such as TLS, WireGuard [1], and Signal [2], are the cornerstones of today's global communication networks because they ensure crucial security properties, such as participant authentication and data privacy. With Lowe's famous attack on the Needham-Schroeder protocol [3, 4], it has become obvious that formal proofs are necessary for verifying that cryptographic protocols actually provide the desired properties.

The vast majority of existing work on automated protocol verification targets protocol *models*, i.e., abstract descriptions of the cryptographic operations and message exchanges that constitute a protocol. The verification of protocol models is useful to show the security of the protocol *design*, but does not guarantee that concrete protocol *implementations* are also secure. Common programming errors (e.g., missing bounds checks in the Heartbleed bug [5]) or incorrect implementations of the design (e.g., accidentally omitted protocol steps in the Matrix SDK [6]) may render the implementation insecure even if the protocol design is secure.

Verifying protocol implementations is substantially more complex than verifying models. Targeting realistic implementations requires reasoning, for instance, about mutable data structures, intricate control-flow (e.g., dynamic dispatch), concurrency, and performance-optimized code. Moreover, implementations are significantly larger than abstract models, and change more frequently, which requires *modular* verification techniques to decompose the verification task and reduce the re-verification effort when code evolves. Modular reasoning is more difficult than the non-modular analyses typically used to verify protocol models.

One approach at obtaining verified protocol implementations is to generate an executable implementation automatically from a verified model (e.g., [7–11]). For instance, Bhargavan et al.'s DY* framework [9–11] generates OCaml or C code from a functional implementation in F* [12]. The generated code is secure by construction (provided the code generator is correct). However, changing the code manually (e.g., to optimize performance) forfeits any security guarantees. To achieve modular verification, DY* relies on a specific coding discipline (at most one protocol step per F* function), which must be enforced manually, and is in general not adhered to by existing implementations. A violation of this discipline unwittingly restricts the capabilities of the attacker and, thus, may cause DY* to miss attacks.

An alternative approach is to verify security properties for a protocol model that is extracted automatically from an implementation (e.g., [13–17]). However, automatic model extraction often requires that implementations follow restrictive coding disciplines. Similar restrictions apply to approaches based on *executable models* (e.g., [18,19]), i.e., models written in specific subsets of programming languages that facilitate reasoning, but typically do not provide the low-level features required for optimized implementations.

*The work was performed during a fellowship at ETH Zurich.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

Instead of automatically generating an implementation or extracting a model, Arquint et al. [20] prove refinement between an *existing* verified model and a corresponding *existing* implementation. Their approach supports realistic implementations, but requires expertise in and relies on the soundness of two tools (a model *and* a program verifier). Moreover, formal models may not always exist, or may not be in sync with an evolving implementation.

This work. We present a methodology for the verification of strong security properties (e.g., injective agreement, forward secrecy) directly on the level of the protocol implementations. Our methodology leverages established program verification techniques that are supported by a wide range of existing automated[1] tools (e.g., [21–25]), which makes it readily applicable. It is based on separation logic [26, 27], a program logic that supports the language features used to write efficient implementations, such as mutable heap data structures and concurrency. As a result, our methodology applies to realistic implementations written in mainstream programming languages such as C, Go, JavaScript, and Rust. Verification in our methodology is *modular*, that is, one can verify each method (or protocol participant) in isolation. Modularity is crucial for scalability, to reduce the re-verification effort when the code evolves, and to provide strong guarantees for libraries.

As is common in protocol verification, we explicitly model the global trace of a protocol execution, which allows us to express security properties in ways familiar to security experts. This trace is expressed and manipulated via *ghost code* [28], that is, program code that is used for verification purposes, but erased by the compiler before the program is executed. The ghost code required to manipulate the global protocol trace is encapsulated in the I/O and crypto libraries used by an implementation to ensure, e.g., that each sent message is correctly reflected on the trace.

Using ghost code allows us to cleanly separate the global trace, which is necessary to prove protocol-wide properties, from the data structures maintained locally by each participant. We treat each participant instance of a protocol (including a Dolev-Yao attacker [29]) as a concurrent thread, and the global trace as shared state among these threads. This approach allows us to reason about unboundedly many participant instances and to leverage existing verification techniques and tools for shared-data concurrency.

Contributions. We make the following contributions:

(1) We present a modular verification methodology for protocol implementations, based on global traces and concurrent separation logic, that applies to a wide range of programming languages, protocol implementations, and verification tools.
(2) We show how to use separation logic's linear resources to *modularly* prove injective agreement, i.e., the absence of replay attacks. To the best of our knowledge, we present the first invariant-based verification technique for this property.
(3) We developed a reusable Go library that facilitates maintaining the global trace; protocol-independent properties are verified once and for all for this library and can, thus, be reused for different protocol implementations.
(4) We demonstrate the practicality of our approach by using the Gobra verifier [22] to verify memory safety and security of Go implementations of three protocols: Needham-Schroeder-Lowe (NSL) [3,4], signed Diffie-Hellman (DH) [30], and WireGuard [1]. We show that our approach supports different programming languages and verifiers by additionally implementing a prototype of the reusable library for C and the VeriFast verifier [21], and using it to verify a C implementation of NSL. The implementations of our reusable verification library and the case studies are open-source [31].
(5) We prove soundness of our approach, in particular, that the global trace correctly reflects all relevant protocol steps and, thus, any security property proved for the trace indeed holds for the protocol implementation.

We build on and substantially extend two lines of prior work: Our use of a global trace and security labels to prove secrecy is inspired by Bhargavan et al. [9], but our approach achieves modularity without relying on a coding discipline (cf. earlier discussion of DY*), and thus handles existing protocol implementations soundly. Our encoding of the global trace as a concurrent data structure is inspired by Dupressoir et al. [32]. Their work depends on specific features of the used programming language (e.g., C's volatile fields) and verifier (VCC [33]), while we present a separation-logic-based methodology applicable across different programming languages and verifiers, as demonstrated by our case studies. The use of separation logic allows us to verify concurrent, heap-manipulating programs and prove security properties that so far were out of reach for invariant-based approaches.

Outline. Sec. 2 introduces background on trace-based verification and our attacker model. In Sec. 3, we explain how we encode the global trace and how we relate it formally to the local state of each participant. In Sec. 4, we show how to prove important security properties based on a suitable trace invariant, and how we use separation logic's linear resources to prove injective agreement. In Sec. 5, we introduce our reusable verification library, which implements our methodology, and substantially reduces the verification effort per protocol. Sec. 6 describes our case studies. We explain the trust assumptions underlying our methodology and sketch its soundness proof in Sec. 7, discuss related work in Sec. 8, and conclude in Sec. 9.

2 TRACE-BASED VERIFICATION

A protocol's security depends on the interplay of the protocol participants in the presence of an attacker. A standard technique to verify security is to record all relevant actions of the participants and the attacker on a *global trace* and to formulate the intended security properties as properties of this trace. Verification then amounts to proving that all possible traces of a protocol satisfy the intended properties. In this section, we give a high-level overview of this approach; we provide the details in the later sections.

Attacker. We consider a Dolev-Yao attacker that has full control over the network and performs symbolic cryptographic operations. These operations are modeled as functions over symbolic values, so-called *terms*, and encode the perfect cryptography assumption, e.g., that decryption succeeds if and only if it uses the correct key.

An attacker can apply these functions to all terms in its knowledge, which initially consists of all publicly-known terms, including string and integer constants. An attacker obtains additional knowledge by reading messages on the network. Furthermore, an attacker

[1] The proof search is automatic but relies on user-provided annotations.

$$\frac{}{\Gamma \vdash [p \mapsto \_]\ \ {*p} := e\ \ [p \mapsto e]}\ (\textsc{Write}) \qquad \frac{\Gamma \vdash [P_1]\ \ C_1\ \ [Q_1] \qquad \Gamma \vdash [P_2]\ \ C_2\ \ [Q_2]}{\Gamma \vdash [P_1 * P_2]\ \ C_1 \,||\, C_2\ \ [Q_1 * Q_2]}\ (\textsc{Par}) \qquad \frac{\Gamma \vdash [P * I_r]\ \ C\ \ [Q * I_r]}{\Gamma, r : I_r \vdash [P]\ \ \textbf{with}\ (r)\ \{C\}\ \ [Q]}\ (\textsc{With})$$

Figure 1: Selected separation logic proof rules: heap writes (cf. Sec. 3.1) along with parallel composition and lock-protected critical sections (cf. Sec. 3.3). Side-conditions are omitted for simplicity.

may corrupt participants, which adds all terms in the state of the corrupted participant to the attacker knowledge. We model two kinds of corruption: Corrupting a *participant* leaks its long-term state, which is common to all instances of this participant, such as long-term secret keys. Corrupting a *participant session* additionally leaks short-term state, e.g., ephemeral secret keys, or exchanged nonces[2].

Trace entries. The global trace is a sequence of events. Each event corresponds to a high-level operation performed by a participant or the attacker. It has a name and takes event-specific arguments. E.g., event *CreateNonce*(n) records that nonce n was created. This event is protocol-independent; we also support protocol-specific events to keep track of the progress within a protocol execution and to express specific security properties. E.g., a protocol-specific event may express which nonces or keys a participant uses to communicate with a peer (cf. Sec. 4).

We use seven protocol-independent events: (1) A *create nonce* event records that a fresh nonce has been generated. (2) A *send message* event records that a message has been sent on the network. Both events may originate from a participant or the attacker. The remaining five protocol-independent events model the capabilities of the attacker. (3) The (unique) *root* event is the first event on every trace and contains the initial attacker knowledge. (4) An *extend attacker knowledge* event models that the attacker learns additional terms, e.g., by applying a cryptographic operation to a term already in the attacker knowledge. Corruption is represented by (5) a *participant corruption* or (6) a *session corruption* event. In both cases, we use extend-events (4) to add the newly-learned terms (from the corrupted state) to the attacker knowledge. At any point during a protocol run, the total attacker knowledge is therefore determined by the union of the root event (3), the send-events (2), and the extend-events (4). Finally, (7) a *drop message* event records that the attacker dropped a message from the network.

Trace invariant. To reason modularly about the (unbounded) set of all possible traces, we introduce a *trace invariant*, a property that must hold for every prefix of each trace produced by a protocol. Verification then consists of two main steps: first, proving that each action of a participant or the attacker (according to the above attacker model) maintains the trace invariant and, second, showing that the trace invariant implies the intended security properties.

An important component of a trace invariant are *message invariants*, which characterize the content of a message. For instance, a message invariant might express that a message parameter is a nonce (as opposed to an arbitrary term).

3 LOCAL REASONING

In the previous section, we have summarized how we can prove security properties based on a global trace of events. In this section, we show how to verify concrete protocol implementations by relating the global trace of the protocol to the local state and operations of each protocol participant. This verification is modular and can be automated using existing verification tools.

3.1 Safety Verification

To support realistic, efficient, and existing protocol implementations, our verification technique needs to handle programming concepts such as mutable heap structures and concurrency. To this end, we employ separation logic [26, 27], the de-facto standard for the modular verification of imperative code. Separation logic is supported by existing verifiers for many languages, including VeriFast [21] for C, Prusti [25] for Rust, and Gobra [22] for Go. All of them can be used in combination with our methodology.

In separation logic, each heap location is conceptually owned by a single function execution (similar to Rust). Attempting to access a location *without* owning it results in a verification failure. Ownership prevents data races in concurrent programs (since at most one function may access a location at any point in time) and facilitates reasoning about side effects (as long as one function owns a location, no other function can possibly modify it).

In specifications, the *points-to assertion* $p \mapsto e$ expresses ownership, i.e., that the current function has an exclusive *permission* to access location p and that p has value e (we write _ if the value is irrelevant). For instance, the proof rule for heap updates (rule Write in Fig. 1) enforces via its precondition that the current function execution may update p only if it holds the corresponding permission.

Permissions are initially obtained when allocating a heap location, and are transferred between function executions upon call and return according to the callee function's specification. Permissions may also be transferred between threads, see Sec. 3.3.

Verifying a protocol implementation in separation logic ensures that it is memory safe (e.g., does not cause null-pointer dereferences or buffer overflows), does not abort (e.g., due to division by zero), and does not exhibit data races. Where needed for our safety or security proof, we also verify functional correctness properties. We omit the details of safety proofs here because they are routine work in and orthogonal to the focus of this paper.

3.2 Relating Bytes with Terms

Our global trace includes symbolic terms, such as keys, nonces, and messages. In concrete implementations, these terms are typically represented by (mutable) byte arrays. In order to relate the two, we use a *concretization function* γ, which maps a term to its byte representation. We use this function in specifications; in particular, we have annotated library functions, e.g., for cryptographic operations, to relate the term representations of their inputs and outputs. E.g., a hash function that maps the byte array *xa* (representing, e.g., a

[2]Session corruption affects the entire short-term state of a participant instance, which might participate in multiple protocol sessions; a more fine-grained treatment of individual sessions is possible, but omitted for simplicity.

$$M1. \quad A \to B : \{\langle 1, na, A\rangle\}_{pk_B}$$
$$M2. \quad B \to A : \{\langle 2, na, nb, B\rangle\}_{pk_A}$$
$$M3. \quad A \to B : \{\langle 3, nb\rangle\}_{pk_B}$$

Figure 2: The NSL public key protocol, where *na* and *nb* are nonces, whose generation is omitted. $\{m\}_{pk}$ and $\langle\cdots\rangle$ denote public key encryption of plain text m under the public key pk and tupling, respectively. Creation and distribution of the participants' authentic keys is not part of the protocol.

message) to the byte array *ra* (representing, e.g., a number) is specified by relating the corresponding terms: $\exists x, r.\, xa = \gamma(x) \wedge ra = \gamma(r) \wedge r = h(x)$, where h is the symbolic hash operation on terms.

Parsing a received message often requires showing that the parsed byte array b corresponds to a given term t: $b = \gamma(t)$. Proving this property generally requires that each byte array corresponds to a *unique* term. However, this requirement is typically not satisfied in realistic implementations where, e.g., a byte array of length four could store an integer or an ASCII-encoded string, which have different term representations. A possible solution [9, 34] is to enforce a unique byte-level representation for every term (for instance, by preceding it with a tag). However, this is not possible when targeting existing implementations with fixed message formats.

Therefore, we adopt a less restrictive solution here. We use the *pattern requirement* from Arquint et al. [20], which allows multiple terms to have the same byte-level representation *in general*, but requires a *unique* representation for the terms corresponding to protocol messages. This requirement allows a participant to uniquely determine the term for a parsed message. It ensures that the concretization function γ is *injective* on the byte arrays received as messages. The pattern requirement is satisfied by many protocols because they include message tags to distinguish the *kinds of messages*, which in turn determines the unique relationship between a byte array and a term. At the same time, it allows clashes among the representations of other terms, such as integers and strings.

We illustrate the approach using the NSL public key protocol [3] in Fig. 2. After receiving message *M1*, Bob parses it as an encrypted triple. The specification of the parse operation ensures $\exists na.\, \gamma(m) = \gamma(\{1, na, A\}_{pk_B})$. Since $\{1, na, A\}_{pk_B}$ is a protocol message, we can apply the pattern requirement to derive the required information about m: $\exists na.\, m = \{1, na, A\}_{pk_B}$.

3.3 Global Trace Encoding

As explained in Sec. 2, we use a global trace of events, verify invariants over this trace, and finally prove that the invariants imply the intended security properties. For this approach to be sound, the global trace has to include all relevant events performed by the protocol participants and the attacker, which we ensure as follows.

We model each participant instance potentially participating in a protocol session, and the attacker, as a thread in a concurrent system. Each thread maintain its own (mutable) local state, which may contain short-term, session-specific data and long-term data that is shared by all instances of the participant. Multiple instances of the same protocol role are modeled as threads that execute the same code. Soundness of separation logic ensures that any verified property holds for all possible interleavings between the threads, that is, for all possible interactions between the participant instances and the attacker. Moreover, since separation logic is modular, it verifies the implementation of each participant in isolation, independent of the other threads potentially running in the system (assuming only that their implementations are also verified). Consequently, the verified properties hold for an arbitrary, unbounded number of participant instances.

Separation logic achieves thread-modular reasoning by ensuring that different threads operate on *disjoint* memory, which prevents data races and eliminates interference between threads (see below for shared state). The proof rule for parallel composition (Par in Fig. 1) illustrates this approach. The threads C_1 and C_2 can be verified independently. They operate on the heap locations for which they obtain permissions via their preconditions P_1 and P_2, resp. Separation logic's *separating conjunction* $*$ in the precondition of the parallel composition expresses that the permissions in P_1 and P_2 are disjoint. Note that we show the rule for a structured parallel composition statement for simplicity; our technique also supports dynamic thread creation.

Each thread needs to manipulate its own local data structures and the global trace data structure that is shared among all threads. To support mutable shared state, we can use any of the established verification techniques for concurrency reasoning. For concreteness, we use a global *lock*, which is associated with a lock invariant that needs to be established when the lock is first created. This invariant may then be assumed whenever the lock is acquired and must be proved to hold upon release. Proof rule With in Fig. 1 illustrates this reasoning for a critical section C that is protected by the lock r. I_r is the invariant associated with lock r, as specified by $r : I_r$ in the proof context. Conceptually, a lock owns the permissions expressed in I_r and temporarily lends these permissions to a thread on entering the critical section.

Since the global trace exists only for the purpose of verification, we model it as *ghost state* and all operations on it as *ghost operations*; both are erased during compilation. Consequently, the lock protecting this ghost data structure can also be erased. Reasoning about ghost locks is completely analogous to standard locks (and supported by separation logic program verifiers). However, since a ghost lock is erased during compilation, it does not ensure mutual exclusion. Therefore, any non-ghost operation performed between an acquire and a release must be *atomic* to ensure that erasing the ghost lock does not create thread interleavings that were not considered during verification.

The trace data structure provides two operations: appending an event, and reading the current state of the global trace. Fig. 3 illustrates how participants and the attacker interact with the global trace. The lock invariant for the global trace is the trace invariant. By formulating this invariant in separation logic, it can express ownership of heap locations and other resources, which allows us to prove security properties that are out of reach for existing invariant-based related work, as we will see in Sec. 4.1.

Participants must record all protocol-relevant operations on the global trace. That is, to perform an operation such as sending a message or creating a nonce, they must (1) acquire the ghost lock, (2) perform the operation, (3) append the corresponding event to the trace, and (4) release the ghost lock (and at this point prove that the trace invariant is preserved). For each relevant operation,

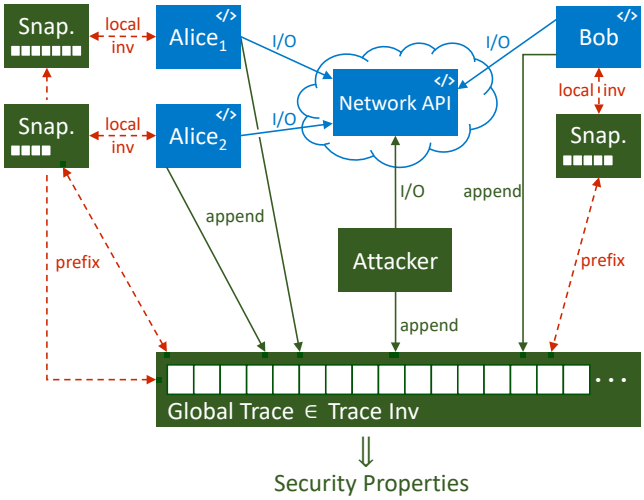

Figure 3: An overview of the main components of a protocol execution in our methodology. The blue boxes are components of the protocol implementation; green boxes denote ghost structures that are used for verification. Blue and green arrows denote actual and ghost method calls, resp. The red dashed arrows denote invariants relating different data structures. Participants and the attacker send and receive messages by interacting with the network. The attacker can perform additional I/O operations such as instructing the network to drop or modify messages. All protocol-relevant operations (including I/O operations) are recorded on a global trace. We verify (global) security properties by proving modularly that each protocol implementation (e.g., two and one implementations of Alice's and Bob's role, resp.) and the attacker maintain a trace invariant, and that the trace invariant implies the security properties. We enable the verification of participants by relating participant-local state with the trace via local ghost state that contains a participant's local snapshot, i.e., its last observed version of the trace.

we provide a library wrapper (see Sec. 5 for details) that performs these four steps[3]. Preconditions on the library functions ensure that the performed operation indeed preserves the trace invariant. Since the trace invariant (and, hence, the preconditions) contain protocol-specific properties, our library is parametric in the invariant (cf. Sec. 5). To ensure that *all* relevant operations are recorded on the trace, it then suffices to perform a simple syntactic check that relevant operations are performed only via the wrapper library.

The attacker is handled similarly. We model it as code that (1) acquires the ghost lock, (2) determines which operations the attacker could potentially perform based on its current attacker knowledge (which is recorded on the trace), (3) non-deterministically chooses any of these operations and appends the corresponding event to the trace, and (4) releases the ghost lock (and at this point proves that the trace invariant is preserved). Verifying this code ensures that all possible attacker operations preserve the trace invariant. In other words, the invariant may state only those properties that are valid under our attacker model, a property we call *attacker completeness* (sometimes referred to as *attacker typability*).

Participant and session corruption are two of the possible attacker operations in step 2 above. In both cases, step 3 adds all symbolic terms possibly present in the participant's (long-term or short-term) state to the attacker knowledge, and step 4 checks that the invariant about the attacker knowledge is maintained.

3.4 Local Snapshots

To prove that a protocol-relevant operation preserves the trace invariant, we frequently need to relate the arguments of the operation to earlier events on the trace. For example, when sending the first message of the NSL protocol (Fig. 2), Alice has to show that the message invariant holds. The message invariant specifies that *na* is a nonce, i.e., requires a prior *CreateNonce*(*na*) event on the trace.

Discharging such proof obligations requires that participants retain information about their prior operations on the global trace. Since the global trace is a shared data structure that may grow between any two accesses, participants may soundly hold on to those facts that are *stable* under extensions of the trace. For instance, if a *CreateNonce*(*na*) is present on the trace at some program point, it will also be present in all future versions of the trace.

We represent the stable information of a participant by maintaining in each participant a *local snapshot* (i.e., a local copy) of the global trace (see Fig. 3). Since the global trace may evolve by actions of other participants and the attacker, the local snapshot of a participant is generally a prefix of the global trace. Whenever a participant performs a protocol-relevant operation, we update its local snapshot to the current global trace. The trace invariant ensures that the local snapshots of all participants are prefixes of the global trace, and that these updates are the *only* modifications of local snapshots.

With this design, local snapshots need to be owned by the participants (to ensure their values are retained across operations of other threads), and they must *also* be owned by the ghost lock (to allow the lock invariant to relate the local snapshots to the global trace). To express this notion of shared ownership, we use fractional permissions [35], which are supported by many separation logics. Conceptually, fractional permissions allow one to split a permission into several fractions; a non-zero fraction permits read access, whereas the full permission is required for write access. Separating conjunction *adds* the fractions in both conjuncts and yields false if the sum for any location exceeds a full permission.

We split the permission to a local snapshot into two halves: One half is part of the trace invariant and lets it express properties of the local snapshot. The other half remains with the corresponding participant and enables the participant to retain information about the global trace. After acquiring the ghost lock, a participant temporarily obtains exclusive permission to its local snapshot by adding the half it holds with the half from the trace invariant (through the precondition $P * I_\tau$ in rule With in Fig. 1) and can, thus, update the local snapshot.

By letting each participant retain a non-zero permission to its snapshot, we can rule out interference from other threads and, thus, use standard sequential reasoning to relate the content of the

[3]To avoid any runtime overhead, calls to this wrapper library could be inlined (and ghost code is erased in any case).

```
1  na /*@, naT @*/ := CreateNonce(/*@ s @*/)
2  //@ assert s.NonceOccurs(naT)
```

Figure 4: Excerpt from a NSL implementation for Alice creating a nonce and demonstrating how to relate local state with the global trace. `//@` and `/*@ ... @*/` mark ghost code that is used for verification only. We omit the nonce's secrecy label (Sec. 4.2) for simplicity.

local snapshot to the concrete data structures maintained by the participant (via local invariants) and to prove the presence of an event on the snapshot. The example in Fig. 4 illustrates that. Line 1 invokes the library function `CreateNonce`. Its regular result `na` is the generated nonce; the additional ghost result `naT` is the corresponding term. `CreateNonce` takes the caller's local snapshot `s` as ghost argument, which allows the function to update the snapshot and to express in its postcondition the existence of the create-nonce event on the updated local snapshot. This postcondition allows the caller to prove the assertion in line 2, without having to consider any interleaving operations by other participants or the attacker.

4 PROVING SECURITY PROPERTIES

In this section, we show how to define a trace invariant that lets us verify two important security properties, authentication and secrecy. Authentication means that two protocol participants are indeed communicating with each other and (depending on the particular authentication property) agree on some common values. Secrecy holds if confidential data remains unknown to the attacker. While we focus here on the proof techniques for these two standard properties, our methodology is also applicable to more complex properties such as forward secrecy, as demonstrated in Sec. 6.3.

4.1 Authentication

To prove authentication, we use protocol-specific events to record additional information beyond the exchanged messages, so that authentication properties can be expressed in a familiar way: as correspondence between these events. In this subsection, we show how to use trace invariants expressed in separation logic to prove two strong and common authentication properties: non-injective and injective agreement.

We illustrate our methodology using the NSL example from Fig. 2. We prove authentication using four protocol-specific events: Before sending the first message, Alice creates event *Initiate*(*Alice*, *Bob*, *na*) to record that she wants to communicate with Bob, and use the nonce *na* in the current protocol session. After receiving the first and before sending the second message, Bob in turn creates event *Respond*(*Alice*, *Bob*, *na*, *nb*), indicating the communication partners and used nonces. Finally, the events *FinishA* and *FinishB*, with the same parameters as *Respond*, indicate successful completion of the protocol (i.e., runtime checks such as nonce comparisons succeeded) for Alice and Bob, resp. We focus on Alice's perspective in the following. We prove authentication for Bob's perspective in Sec. 6.3, where we also discuss authentication properties for WireGuard.

Non-injective Agreement. The fact that Alice agrees with Bob on the nonces *na* and *nb*, known as *non-injective agreement* [36], is specified in Fig. 5 (ignore the conjunct highlighted in blue for now).

```
1  let commit = FinishA(A,B,na,nb) in
2  t.Occurs(commit) ⟹
3  let prefix, i = t.GetPrefix(commit) in
4  (prefix.Occurs(Respond(A,B,na,nb)) &&
5    !(∃A',B',nb',i'. i != i' &&
6      t.OccursAt(FinishA(A',B',na,nb'),i'))
7  ) || prefix.IsCorrupted({A, B})
```

Figure 5: Non-injective (white background) and injective (all lines) agreement from Alice's perspective with Bob on the nonces `na` and `nb`. `t.Occurs(e)` yields whether event `e` occurs on trace `t`; `t.GetPrefix(e)` returns `t`'s prefix up to and including the most recent occurrence of `e`, and the index of that occurrence (i.e., the length of `prefix` minus 1). `t.OccursAt(e,i)` expresses that event `e` occurs at index `i` on trace `t`.

```
1  match ev {
2    case FinishA(A, B, na, nb):
3      UniWit(FinishA, na) &&
4      (prefix.Occurs(Respond(A, B, na, nb)) ||
5        prefix.IsCorrupted({A, B}))
6    ...
7  }
```

Figure 6: A simplified fragment of the trace invariant for NSL-specific events. This invariant is universally quantified over the events `ev` occurring on the trace; `prefix` is the trace prefix up to event `ev`. The invariant expresses that whenever a *FinishA* event occurs on the trace, a *Respond* event with matching arguments must *previously* occur, unless one of the participants has been corrupted. The highlighted line includes a separation logic resource to express that the *FinishA* event is unique w.r.t. to the nonce *na*, which allows us to prove injective agreement. The conjunction `&&` is interpreted as separation logic's separating conjunction `*`.

This trace-based property states that if a *FinishA* event occurs on the trace (line 2) then either a *Respond* event with matching arguments occurs earlier on the trace (line 4) or one of the participants has been corrupted before an agreement was reached (line 7).

To prove agreement for NSL, we include the NSL-specific property from Fig. 6 (ignore line 3 for now) into the trace invariant. It states that for every *FinishA* event, a corresponding *Respond* event occurred prior on the trace, or one of the participants has been corrupted. Maintaining this invariant requires us to show the occurrence of a suitable *Respond* event (or of corruption) when Alice creates the *FinishA* event.

We discharge this proof obligation by extending the trace invariant with a message invariant for NSL's second message, which requires that the *Respond* event occurs on the trace or the message comes from the attacker. Hence, an implementation for Bob has to create a *Respond* event before sending the second message. When Alice receives the message, she may assume its message invariant (as part of the trace invariant). Since her local snapshot gets updated upon the receive-operation, the received message is recorded on the local snapshot and the message invariant becomes part of Alice's stable knowledge. So when Alice adds the *FinishA* event to the trace, she knows that either the *Respond* event occurs on the trace, or the second NSL message comes from the attacker. In the latter case,

Alice can derive that corruption must have occurred because the attacker was able to construct a message containing the nonce *na*, which is accessible only to Alice and Bob (unless corrupted).

Once we established the trace invariant, it remains to show that for all traces, the invariant from Fig. 6 implies non-injective agreement (Fig. 5). This proof is a standard entailment check, which is performed automatically by program verifiers.

Injective Agreement. The stronger property *injective agreement* holds only for implementations that detect if the attacker replays messages from other protocol sessions. If successful, such a replay attack could trick participants into reusing outdated nonces (in general, key material), thereby weakening security. Proving injective agreement modularly is challenging; to the best of our knowledge, we present here the first invariant-based verification technique for injective agreement in protocol implementations (see also Sec. 8).

The highlighted conjunct in Fig. 5 strengthens non-injective to injective agreement by mandating that there is no second *FinishA* event on the trace with the same nonce na. The uniqueness of the event/nonce-pair enforces a one-to-one correspondence between *Respond* and *FinishA* events and, thus, excludes replay attacks.

To prove injective agreement, we strengthen our trace invariant to imply this property. We could in principle include a conjunct that specifies uniqueness by quantifying over the indexes into the trace. However, such an invariant would be difficult to maintain *modularly*. The necessary proof obligation for adding a *FinishA* event would require that no such event with the same first nonce already exists on the trace. However, each participant has only *partial* information about the trace stored in its local snapshot. So even if we proved the absence of an event on the local snapshot, we could not conclude its absence on the trace, such that the proof obligation cannot be discharged.

To obtain a modular verification technique for injective agreement, we leverage separation logic's permissions to encode arbitrary linear resources (non-duplicable facts). Due to the meaning of separating conjunction, $p \mapsto \_ \star p \mapsto \_$ is equivalent to false (because the permissions of the two conjuncts are not disjoint). That is, the points-to assertion $p \mapsto \_$ is a *non-duplicable (i.e., unique) resource*. We can use this fact to model the uniqueness of an event by representing the event as a separation logic permission. We use this mechanism as follows.

Conceptually, we tie event uniqueness to nonces because nonces are, by assumption of perfect cryptography, unique. When a protocol-specific event is declared, it can be specified as unique w.r.t. a specific nonce parameter. E.g., in NSL, event *FinishA* is unique w.r.t. its third parameter *na*. Subsequently, when a nonce is generated via a call to our verification library, a program annotation states for which events this nonce will be used (e.g., *FinishA*). The library call returns not only the fresh nonce (*na*), but also a linear resource for each indicated event type (technical details follow in Sec. 5).

This resource—called an event's *uniqueness witness*—then needs to be given up when the corresponding event is appended to the trace. That is, ownership of the resource is transferred from the participant to the ghost lock by conjoining the resource to the trace invariant. E.g., for NSL, Alice obtains the witness $UniWit(FinishA, na)$ when creating nonce *na*. This witness is transferred to the trace invariant when she appends the event $FinishA(\_, \_, na, \_)$ to the trace, as expressed by the highlighted conjunct in Fig. 6. Due to the linearity of the resource, any attempt to append another *FinishA* event with *na* would fail to verify because the required witness cannot be provided a second time, which would be necessary to preserve the trace invariant.

Consequently, the invariant from Fig. 6 implies that the *FinishA* event with *na* is unique, which allows a standard separation logic verifier to prove the highlighted conjunct in the definition of injective agreement (Fig. 5).

Our discussion shows how the combination of a global trace and local snapshots allows us to prove authentication modularly, and how we can leverage the expressive power of separation logic to specify a trace invariant that lets us prove injective agreement.

4.2 Secrecy

Secrecy of a term *s*, e.g., a key or a nonce, states that the attacker does not learn this term except when corrupting one of the protocol participants that know the term. We can express secrecy as a property of the global trace because we can extract both the attacker knowledge and corruption events from the trace.

Instead of directly reasoning about the concrete attacker knowledge, we follow Bhargavan et al. [9, 37] by over-approximating the concrete attacker knowledge to classes of terms that the attacker (possibly) knows. This over-approximation enables modular reasoning about secrecy: we impose proof obligations that prevent secrets from being leaked to the attacker by checking for every send operation that the sent message belongs to a class already known to the attacker. For instance, if a participant tried to send a (unencrypted) secret term over the network, the send operation would be rejected by the verifier. Consequently, sending a message does not change the over-approximated attacker knowledge. This knowledge is extended only when the attacker corrupts a participant or session. In this case, we add the class of terms readable by the corrupted participant or session to the knowledge.

We classify terms based on their allowed recipients by assigning them a *secrecy label*. Secrecy labels range from public (i.e., everyone including the attacker) over a set of participants to a set of particular protocol sessions. The latter is useful to classify ephemeral private keys, e.g., in our WireGuard case study, because only a participant running a particular protocol session is allowed to read these keys.

By proactively enforcing secrecy labels, we ensure that the (concrete) attacker knowledge may contain only public terms and terms whose secrecy label contains a participant or protocol session that is allowed to read the term and that has been corrupted in the past. We prove this property once and for all as part of our reusable verification library (cf. Sec. 5).

5 REUSABLE VERIFICATION LIBRARY

We implement our methodology as a reusable verification library, which significantly reduces the verification effort per protocol: the library encapsulates the global trace and provides a convenient API for common network and cryptographic operations that automates trace updates. In addition, the library provides various lemmas, such as attacker completeness (Sec. 2), which are proved once and hold for all protocols. To enable verification of a wide range of protocols,

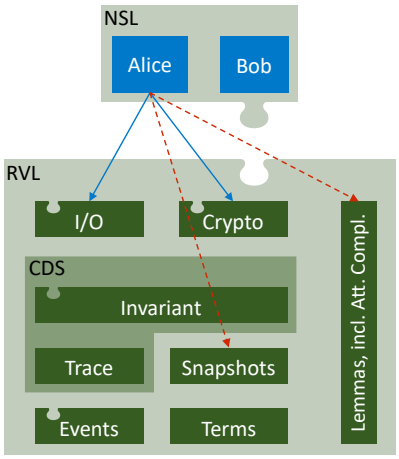

Figure 7: Structure of our reusable verification library (RVL). The library provides implementations for the abstractions used in our methodology: terms, events, the global trace, and local snapshots. Both the trace and all local snapshots are governed by the trace invariant. The trace is encapsulated inside a concurrent data structure (CDS) that permits shared access. The APIs for I/O and cryptographic operations apply these operations and also register the corresponding events on the trace. The RVL also provides several lemmas that have been proved for all protocols, e.g., attacker completeness. Many components of the library are parametric to accommodate protocol-specific events and invariants (and the corresponding preconditions for the I/O and crypto API). We indicate parametric components using a tab symbol near the top of the box. The parameters are supplied for a concrete protocol (here, NSL), as indicated by the tab at the bottom of the box.

the global trace is parametric in the events it records, and the trace invariant is parametric to account for protocol-specific properties.

To demonstrate that our methodology is widely applicable, we developed a library for the Go verifier Gobra [22], and one for the C verifier VeriFast [21]. Both library implementations are available in our open-source artifact [31]. In this section, we give an overview of the library and highlight some of its technical solutions.

5.1 Overview

In the following, we describe the library's structure and components, explain how the library can be instantiated for different protocols, and provide data on its size and verification time.

Components. Fig. 7 illustrates the structure of our library (lower box) and a protocol implementation that uses it (upper box). The library provides the abstractions introduced in Sec. 3: terms and events abstract over concrete data structures (e.g., byte arrays) and participant operations, respectively. Events are recorded on the global trace, whose content is constrained by the trace invariant. The concurrent data structure (CDS) fully encapsulates the trace, to govern shared access and maintain the invariant. Local snapshots are prefixes of the global trace, which also satisfy the trace invariant, but are owned locally by the protocol participants.

The library also provides a convenient API for common network I/O and cryptographic operations: each function performs the corresponding concrete operation (e.g., sending a message or creating a nonce) and also adds the corresponding event to the trace. Suitable preconditions ensure that the operation preserves the trace invariant; they lead to proof obligations for clients using the API. Clients typically discharge these with the help of stable knowledge about the trace, which is recorded in their local snapshots.

In terms of cryptographic operations, our library currently offers asymmetric encryption, authenticated encryption with associated data (AEAD), signatures, and modular exponentiation, but can easily be extended by additional cryptographic operations. As a reference, adding the latter two features and proving the corresponding lemmas took about two person days.

Note that almost the entire library consists of ghost code that is used for verification, but will be erased by the compiler. The only non-ghost operations are the calls to the underlying I/O and crypto libraries. This has two important consequences. First, these calls can be inlined in the participant implementation, such that the entire library can be removed from the executable program and does not cause any runtime overhead. Second, existing protocol implementations do not have to be modified to use the library. The library provides a convenient way to systematically annotate an implementation with ghost code and proof obligations, but other forms of annotations are also possible.

Parametricity. As we discussed earlier, some events and aspects of the trace invariant (and consequently the preconditions of the I/O and crypto API) are protocol-specific. To capture them, we designed our library to be parametric, such that clients using the library can instantiate it for a given protocol.

Despite being parametric, our library nonetheless provides lemmas that are proven once and for all protocols, in particular, *attacker completeness* (Sec. 2) and the *secrecy lemma* (Sec. 4.2). Attacker completeness can be proved once and for all because the library is not parametric in the kinds of term abstractions it provides. Secrecy directly follows from the protocol-independent parts of the trace invariant, which enforce for all protocols that implementations do not leak secrets to the attacker, i.e., messages have to be public. The library provides also several utility lemmas (e.g., that event existence is a stable trace property) that can be used when verifying a participant implementation.

Fig. 8 shows a small excerpt of our trace invariant. The parameter `P` provides protocol-specific events and invariants. Besides various properties of the entire trace (not shown in the figure), the trace invariant also includes event-specific invariants. We show here the invariants for *Send* events and protocol-specific events. A *Send* event requires the message invariant, which itself can be parameterized by library clients. We prove that the generic part of the message invariant is weak enough to be preserved by the attacker; it states, in particular, that the terms occurring in the message do not leak secrets. The protocol-specific part of the message invariant may constrain only encrypted data and must allow the possibility that the encrypted data was fabricated by the attacker out of terms in the attacker knowledge. This ensures that it is maintained by

```
pred TraceInv[P](t: Trace) {
  foreach e: Entry of t:
    let pre = ... in // trace prefix up to e
    match e {
      case Send(msg):
        MsgInv[P](msg, pre)
      case PEvent(pe):
        P::PEventInv(pe, pre)
      ...
    }
}
```

Figure 8: Excerpt of the parametric trace invariant, defined via pattern matching over individual trace entries. All cases may refer to earlier events on the trace via the prefix parameter `pre`. The case for a *Send* event enforces the message invariant, which is partly defined by the library, but itself parametric. A `PEvent` represents any protocol-specific event `pe`. The corresponding case of the trace invariant comes entirely from the protocol parameter `P`.

Library	LOC	LOS	Verification time [s]
Go/Gobra	83	6,932	126.1
C/VeriFast	343	3,837	0.8

Figure 9: Lines of code (LOC) and lines of specification (LOS) (incl. ghost code) for the library code, together with the average verification times in Gobra and VeriFast.

all attacker actions. For a protocol-specific event, the invariant is supplied entirely by the parameter `P`. In the following, we explain how this parameter is represented in our library implementations.

In the Go implementation of the library, we achieve parametricity by using Go interfaces. In particular, the *generic protocol interface* declares mathematical functions (e.g., *isUnique* to indicate that an event is unique), separation logic predicates (e.g., protocol-specific event invariants), and lemmas. Clients may then supply different implementations of this interface with different definitions for these functions, predicates, and lemmas. Gobra checks via suitable proof obligations that any concrete implementation satisfies key properties specified in the interface (e.g., that protocol-specific invariants provide uniqueness witness resources for unique events). These properties can thus soundly be assumed while verifying the parametric library. Analogously, parametricity w.r.t. events is enabled by declaring an *Event* interface that protocol-specific events extend.

In VeriFast, we use its generic types (e.g., for events), abstract mathematical functions (e.g., *isUnique*), and abstract lemmas (e.g., that the event invariant is stable) to achieve parametricity and verify the library once for all protocols. When verifying implementations of a particular protocol, these abstract functions and lemmas are concretized by providing function and lemma definitions via an automated syntactic transformation. We prove that these definitions are not present while verifying the library, that is, we indeed verify the parametric version of the library, not a concrete instantiation.

Statistics. Fig. 9 shows the size and verification time for the two verified implementations of our library. As explained above, the library consists mostly of ghost code; only around 1% is executable code. All methods and lemmas together are verified in ca. 2 minutes. The library for VeriFast is currently less complete than the one for Gobra, and lacks several useful lemmas, which explains the smaller amount of ghost code. It verifies in 1 second (VeriFast is usually faster than Gobra, but provides less automation). We have measured the verification times by averaging over 30 runs on a 2020 Apple Mac mini with M1 processor and macOS Ventura 13.0.1. Since the library is verified once for all protocols, this effort does not have to be repeated when verifying a concrete protocol implementation.

5.2 Technical Solutions

In the following, we summarize the features of a verification technique and tool required to implement the main abstractions (e.g., terms, events, global traces) provided by our library.

Custom Mathematical Theories. Verification techniques frequently represent information as values of mathematical theories, such as sets, tuples, sequences, etc. In contrast to the corresponding data types of a programming language, these values are immutable and their operations have a direct representation in the verification logic, which simplifies reasoning.

We use mathematical theories to represent the abstractions we use in specifications and ghost code: events, the global trace, secrecy labels, and terms with equational theories. Conceptually, events form an algebraic data type (ADT), as does the global trace (a functional list). Labels and terms are also algebraic structures, but with additional properties (e.g., labels have a commutative join operator).

The Gobra implementation of the library represents all these structures as uninterpreted functions with appropriate axioms (analogous to how custom theories are encoded to SMT solvers). E.g., for the ADT of events, we define axioms that ADT constructors are injective in their arguments, and that different constructors produce different events. For terms, we define additional axioms to encode cryptographic equational theories, e.g., $g^{x\,y} = g^{y\,x}$, where g^x denotes Diffie-Hellman exponentiation with generator g. VeriFast supports ADTs natively, which we use to represent events and the global trace. For labels and terms, we again use uninterpreted functions and axioms ("auto-lemmas") to express equational theories.

Linear Resources. Our novel support for proving injective agreement (cf. Sec. 4) requires reasoning about the uniqueness of certain protocol-specific events. For this purpose, we introduce (ghost) memory locations and use separation logic's (exclusive) permissions to these locations as linear resources. Separation logic predicates [38] allow us to construct linear resources with arbitrary parameters by mapping the parameter tuples injectively to a heap location. We use such predicates to represent the uniqueness witnesses from Sec. 4.

Concurrency Reasoning. As discussed in Sec. 3.3, we model the global trace as a *concurrent* data structure. Our approach is compatible with any verification technique that is able to reason about shared accesses to such a data structure and to maintain an invariant over it. Moreover, to encode local snapshots (cf. Sec. 3.4), we require support for reasoning about properties that are stable under concurrency, which are offered by separation logic verifiers.

We model the global trace as a data structure that is protected by a ghost lock. Neither Gobra nor VeriFast support ghost locks directly, but both offer standard locks. Reasoning about ghost locks and standard locks is almost identical, with one exception: Any non-ghost operations performed between acquiring and releasing

```
1  struct Alice  {
2    SkA: byte[]
3    PkB: byte[]
4    Na:  byte[]
5    Nb:  byte[]
6    /*@ Step: uint @*/
7    ...
8  }
9
10 /*@ pred LocalInvariant(a: Alice) {
11   ∃naT,nbT.
12   ... && // memory omitted
13   (a.Step == 2 ==>
14     UniWit(FinishA, naT)) &&
15   (a.Step >= 2 ==>
16     γ(naT) == a.Na &&
17     a.Snap().NonceOccurs(naT)) &&
18   (a.Step >= 3 ==>
19     γ(nbT) == a.Nb &&
20     a.Snap().Occurs(FinishI(A, B, naT, nbT)))
21 } @*/
```

Figure 10: The struct used for Alice's local state in the Go implementation of NSL, and an excerpt from the local invariant that relates this state to Alice's local snapshot and, thereby, to the global trace. The `Step` field is a ghost field that is used to track Alice's progress in the protocol.

a ghost lock must be atomic (because the lock will be erased by the compiler, so it does not actually provide mutual exclusion). This property is satisfied in our library.

6 CASE STUDIES

We applied our methodology to Go implementations of the NSL public key protocol, signed Diffie-Hellman (DH) key exchange, and the WireGuard VPN protocol, and prove strong security properties. We also verified a C implementation of NSL, and obtained the same security properties as for the Go implementation. Our case studies (included in our artifact [31]) thus demonstrate the portability of our methodology across different protocols, programming languages, and verifiers, and its scalability to realistic, interoperable implementations. In this section, we first summarize each of the case studies and then discuss our experiences.

6.1 Needham-Schroeder-Lowe

We used Gobra to verify a Go implementation of the initiator and responder roles for the NSL protocol (cf. Fig. 2), and likewise VeriFast for a C implementation thereof. We implemented the core of the protocol as one method per participant; we also verified an alternative Go implementation of the initiator that contains one method per message to demonstrate that verification is not sensitive to the code structure. Both protocol roles store their program state locally and use an invariant to relate the local state via the term abstraction to their local snapshot and, thereby, to the global trace.

Fig. 10 illustrates the interplay between the local state and the local snapshot for the initiator, Alice. Alice manages her program state in a struct `Alice`. The local invariant in lines 10–21 relates Alice's local state to her local snapshot (and, thus, indirectly to the global trace). This invariant expresses ownership of the heap locations for the struct fields, which is omitted in the figure. More importantly, it specifies properties about the struct fields depending on Alice's progress within the protocol execution, which we keep track of via the `Step` field. E.g., Alice is in Step 2 after creating the nonce *naT* and sending the first message. In this case, the invariant includes the uniqueness witness (line 14), which allows Alice to create the *FinishI* event in a later protocol step. The invariant relates the concrete nonce field `Na` to its term representation `naT` using the concretization function γ (line 16). This term is used in the events on the global trace. In particular, the *CreateNonce* event for `naT` must occur on Alice's local snapshot `a.Snap()` (line 17) and, thus, on the global trace. Once Alice's protocol run has reached the final Step 3, it adds the *FinishI* event to the trace. The invariant reflects this by stating that the event is on the local snapshot (line 20). Knowledge about *FinishI*'s existence on the trace entails (via the trace invariant) properties about the *Respond* event created by Bob (recall Fig. 6). This knowledge, together with *FinishI*'s uniqueness witness (now stored in the trace invariant), allows us to prove injective agreement with Bob as explained in Sec. 4.1.

$$
\begin{aligned}
&M1. \quad A \to B : g^x \\
&M2. \quad B \to A : \{\!|\langle 0, B, A, g^x, g^y\rangle|\!\}_{sk_B} \\
&M3. \quad A \to B : \{\!|\langle 1, A, B, g^y, g^x\rangle|\!\}_{sk_A}
\end{aligned}
$$

Figure 11: The signed DH key exchange protocol, where g^x and g^y are DH public keys and $\{\!|m|\!\}_{sk}$ denotes cryptographically signing a payload m with a secret key sk.

We prove for all participant implementations that they achieve (at the end of a protocol execution) injective agreement on, and secrecy for, both nonces *na* and *nb*. Additionally, we verify initialization code that creates an empty trace, generates public/private key pairs for the participants, and spawns two participant instances as Go routines (similar to threads) to demonstrate that key distribution (although not part of the protocol) can be modeled using our methodology.

6.2 Signed Diffie-Hellman

In the signed DH key exchange (cf. Fig. 11), Alice and Bob each generate a DH secret key x and y, respectively. By transmitting the corresponding (signed) DH public keys g^x and g^y, they agree on the shared key $g^{x\,y}$ after a successful protocol run.

We prove secrecy for, and injective agreement on, the shared key. The proof is similar to the proof for NSL, which allowed us to reuse substantial parts. One noticable difference is that proving that both participants derive the *same* shared key requires the equational theory for Diffie-Hellman exponentiation. Our reusable verification library provides such custom theories, as discussed in Sec. 5.2. Another difference is that the nonces x and y are not directly part of the protocol messages (in contrast to *na* and *nb* in NSL), but are existentially quantified in the message invariants. A participant instance can determine the values of these existentially-quantified variables after receiving a protocol message, by connecting the message invariant to its own Diffie-Hellman secret key.

6.3 WireGuard

As our main case study, we have picked the WireGuard VPN protocol as a real-world protocol achieving even stronger security properties than NSL. WireGuard is a modern, open-source, and cross-platform VPN that uses state-of-the-art cryptography and is part of the Linux kernel. The WireGuard protocol, which performs an authenticated key exchange, has been analyzed rigorously [39, 40]. It consists of a handshake and transport phase. During the handshake phase, the protocol participants agree on two session keys k_{IR} and k_{RI}, one per direction, that are used to symmetrically encrypt VPN packets in the transport phase.

Implementation. We used the existing Go implementation from Arquint et al. [41], whose memory safety proof we reused. Thanks to our reusable verification library's parametric design, instantiating our library with the concrete networking library used by the WireGuard implementation was straightforward and only required annotating cryptographic functions with suitable postconditions.

Arquint et al.'s implementation is a subset of WireGuard's official Go implementation. It omits advanced VPN features such as DDoS protection, session key renewal, and support for multiple concurrent VPN connections. Moreover, their implementation reduces concurrency (which we partly re-introduced, as we discuss below), and replaces a message buffer pool by single-use buffers. Our technique could handle the removed features with additional effort that is mostly orthogonal to our methodology. For instance, the implementation of DDoS protection collects metrics (which does not pose a challenge for program verification) and uses a slightly different handshake (whose verification is analogous to the standard handshake; the differences are not relevant for authenticity and secrecy). Supporting multiple VPN connections requires slightly more complex data structures, as do buffer pools, which can easily be handled in separation logic. Despite these simplifications, the implementation is interoperable with other WireGuard implementations and supports tunneling IP packets via the established VPN connection to and from the operating system. Since each IP packet is encrypted using a distinct counter value, a new handshake must be performed before the counter reaches its upper limit, which is not yet implemented. Instead, the implementation stops forwarding IP packets at that point.

Our case study goes substantially beyond of Arquint et al.'s, which focuses on connecting Tamarin to code-level verification, and proves weak forward secrecy and non-injective agreement in the presence of long-term key corruption. We additionally consider session corruption, i.e., the possibility for an attacker to obtain ephemeral key material, and prove *strong* forward secrecy and *injective agreement with actor key compromise (AKC) security*. Furthermore, we have re-introduced (from the official WireGuard implementation) and verified the ability to send and receive transport messages in the initiator *concurrently*. This change increases TCP throughput compared to Arquint et al.'s implementation by a factor of 180, which illustrates how important such code optimizations are for real-world protocol implementations. The initiator verified in our work reaches 72% of the official implementation's throughput; the additional concurrency needed to close the remaining performance gap, requires standard concurrency reasoning in separation logic, which is supported by our methodology.

```
1 !t.AttackerKnows(s) ||
2   t.GetHs(ASess, PSess).IsCorrupted({A, P}) ||
3   t.IsSessionCorrupted({ASess, PSess})
```

Figure 12: Strong (without highlighted part) and weak forward secrecy (entire property) for a session key `s` on trace `t`. `A` and `P` identify the actor and peer that derive the key in their protocol sessions `ASess` and `PSess`, respectively. `t.GetHs(ASess, PSess)` returns a prefix of `t` up to and including the corresponding handshake's completion from the actor's perspective. The key is protected against (future) participant corruption after the handshake's completion.

Security Properties. Since the session keys are based on ephemeral as well as long-term key material that is contributed by both protocol participants, WireGuard achieves strong security properties. In particular, we prove forward secrecy and injective agreement, both with actor key compromise (AKC) security. While WireGuard optionally incorporates a pre-shared symmetric key into the handshake to increase security, we prove all security properties in this section without considering this pre-shared key, i.e., we treat the pre-shared key as a term known to the attacker. In the following, we call the initiator *actor* and the responder *peer* when proving a property from the initiator's perspective, and vice versa for the responder's perspective.

Forward secrecy protects sessions against future corruption of the long-term secret keys. I.e., an attacker cannot compute the session keys of an already established session after learning the long-term secret keys. However, sessions that get established after corrupting the long-term secret keys are not protected because the attacker can impersonate participants by knowing their secret keys. The literature distinguishes between weak and strong forward secrecy. We were able to reuse formalizations from existing work [11, 42, 43], which are phrased as trace-based security properties and, thus, directly supported by our methodology.

Weak forward secrecy for a session key s (cf. entire Fig. 12) holds if at any point in time, one of the following three properties hold: (1) The attacker does not know s (line 1), (2) the actor or its peer has been corrupted before completing the handshake (line 2), or (3) the actor's or peer's session has been corrupted (line 3). In the last case, the attacker gets to read the long-term and short-term state of the corrupted participant, that is, the long-term secret key and also the session keys if the session is established. Hence, the attacker either directly obtains the session keys if the session is already established or otherwise uses the long-term secret key to impersonate the actor or its peer while establishing a session in the future. The session keys of all other sessions remain secret.

Compared to weak forward secrecy, session keys satisfying *strong forward secrecy* are additionally protected against corrupting the actor, i.e., the highlighted actor is removed from line 2 in Fig. 12. In particular, having access to the actor's long-term secret key does not allow the attacker to obtain the established session keys. This resilience has been formalized as actor key compromise (AKC) by Basin et al. [44], generalizing the more widely known notion of key compromise impersonation (KCI).

From the initiator's perspective, WireGuard guarantees strong forward secrecy for the two session keys once the handshake has

```
let commit = Commit(A,P,ASess,PSess,m) in
let running = Running(A,P,ASess,PSess,m) in
t.Occurs(commit) ⟹
let prefix, i = t.GetPrefix(commit) in
(prefix.Occurs(running) &&
  !(∃A',P',ASess',PSess',i'. i != i' &&
    t.OccursAt(Commit(A',P',ASess',PSess',m),i'))
) || prefix.IsCorrupted({P})
  || prefix.IsSessionCorrupted({ASess})
```

Figure 13: Injective agreement with AKC security on a term `m` from the actor `A`'s perspective with a peer `P`. The highlighted conjunct indicates the `Commit` event's uniqueness requirement for the given `m`.

Case Studies	LOC	LOS	Verification time [s]
Go/Gobra			
NSL	197	924	97.6
Signed DH	225	888	119.0
WireGuard	557	5,815	268.1
C/VeriFast			
NSL	300	1,014	5.0

Figure 14: Lines of code (LOC) and lines of specification (LOS) (incl. ghost code) for our case studies, together with the average verification times in Gobra and VeriFast. We performed the measurements in the same way as in Fig. 9.

been completed. In contrast, the responder guarantees only weak forward secrecy by the end of the handshake, but achieves strong forward secrecy after receiving the first transport message. We verified strong forward secrecy at the appropriate points in the protocol for both roles.

The responder's forward secrecy guarantee is strengthened by receiving and successfully processing the first transport message because this message acts as a key confirmation. I.e., the responder checks that it derived the same session key k_{IR} as the initiator, which allows the responder to detect AKC attacks. Based on strong forward secrecy for the session keys, we further prove that the VPN payloads are treated with the same level of secrecy. This induces proof obligations that a participant sends VPN payloads to the network in a way that they can be read only by participants allowed to read the session keys (e.g., by encrypting the VPN payloads with one of the session keys).

Confirming the session keys not only enables strong forward secrecy for the session keys but also provides additional authentication guarantees: *Injective agreement with AKC security* (cf. Fig. 13) states that (1) an actor A agrees with a peer P on a term m with a one-to-one correspondence between the *Commit* and *Running* events unless (2) the actor's session or (3) the peer's (short-term or long-term) state has been corrupted. In particular, corrupting the actor is not sufficient to satisfy this property. In contrast, the NSL protocol only satisfies injective agreement *without* AKC security (as presented in Sec. 4.1) from the initiator's perspective because having access to the initiator's secret key enables the attacker to decrypt the second message, obtain the nonces na and nb, and construct a modified second message containing na and nb' with $nb \neq nb'$. Thus, there is no correspondence between *Commit* and *Running* events in the case of actor key compromise because the initiator and responder do not agree on the nonces.

6.4 Discussion

For each case study, Fig. 14 reports the size of the implementation and its specification, along with the verification time. We exclude the alternative NSL initiator implementation in Go, and the reusable verification library (recall Fig. 9). However, the specifications do include the (ghost) code instantiating our reusable verification library: it amounts to 374, 370, and 1,077 LOS in Gobra for NSL, DH, and WireGuard, respectively, and 391 LOS in VeriFast for NSL.

Overall, the annotation overhead for Gobra ranges between 3.9 and 10.4 LOS per line of code, and is in the typical range for modular program verification. For example, Wolf et al. [22] report a ratio of 2.7 for a small example using concurrency in Gobra. VST-Floyd [45], a separation logic-based verifier based on Coq, reports an average ratio of 13.9 for small C programs. Both works verify only memory safety and functional properties, but do not include any (arguably much more complex) security properties, whereas our numbers include safety and security. As another data point, Arquint et al. [20] prove security properties for WireGuard in Gobra with a ratio of 6.5, in addition to a Tamarin model of 350 lines and a Tamarin oracle implemented in Python, which ensures that Tamarin's proof search terminates. Counting the Tamarin model and oracle as specification, the overall ratio is 7.1 (and requires the use of three different languages).

The main challenge in our case studies was finding a sufficiently strong trace invariant to prove the presented security properties. For WireGuard, we had to find suitable message invariants such that the secrecy labels for the derived session keys k_{IR} and k_{RI} are sufficiently strong to prove weak and strong forward secrecy. These secrecy labels are related to the message invariants because the session keys are derived by an eightfold application of key derivation functions (KDFs) that factors in long-term and ephemeral, i.e., session-specific, Diffie-Hellman key material that is either locally generated or received from the peer. Thus, each KDF application results in a new key with a secrecy label that depends on the secrecy labels of the input key material. To keep the annotations related to the secrecy labels in the implementation to a minimum, we have implemented a lemma for each KDF application that proves the result's secrecy label.

Moreover, the invariant for protocol-specific events has to be strong enough to prove injective agreement with AKC resilience. Our reusable verification library enables strengthening the proven authentication property from non-injective to injective by adding the uniqueness witness for each protocol-specific event. This allowed us to focus on finding a suitable invariant for non-injective agreement with AKC resilience first, and then strengthen the authentication property, which required less than 40 additional LOS.

After completing the proofs for sequential code, we re-introduced concurrency to the initiator's transport phase (recall Sec. 6.3), which entailed only minimal proof changes and was done in an afternoon. This demonstrates that our separation-logic-based methodology enables security proofs that are robust w.r.t. nontrivial code changes.

7 TRUST ASSUMPTIONS AND SOUNDNESS

Our methodology allows us to prove strong security properties for implementations of security protocols. Like with all verification techniques, these proofs rely on several assumptions about the implementation and the execution environment.

We rely on the soundness of the used program verifier. Since our methodology is compatible with standard separation logic verifiers, we can mitigate this assumption by using a mature tool.

As is standard for symbolic cryptography, we assume perfect cryptographic operations (e.g., absence of hash collisions, or that ciphertexts do not leak any information). We also do not verify that the implementations of the cryptographic primitives are functionally correct; while this is orthogonal to our work, our methodology could be combined with verified libraries like EverCrypt [46].

Furthermore, we assume that all *output* operations, i.e., sending messages, are reflected on the global trace by corresponding events, which is the case when using the I/O operations provided by our verification library. However, if an implementation uses, e.g., inline assembly or third-party libraries to send messages to the network, the global trace has to reflect these messages nonetheless. Omitting any other event does not affect soundness, only completeness.

Lastly, we assume that the protocol terms corresponding to the byte arrays in a participant's *initial* state, and those obtained from operations outside of our library (e.g., read from a config file), are readable at least by that participant according to the terms' secrecy labels (recall Sec. 4.2). Otherwise, it would not be sufficient to model corruption of a participant by adding the class of terms readable by that participant to the attacker knowledge; the attacker could learn even more. For all terms a participant can obtain by interacting with our verification library (e.g., receiving messages, generating nonces, applying encryption), we prove in our library (via corresponding lemmas) that a participant can read these terms (and thus the terms leak as expected to the attacker in case of corruption).

We sketch soundness of our methodology in App. A by showing that the global trace reflects all relevant protocol steps and, thus, any security property proved for the trace indeed holds for the protocol implementation. For this purpose, we define a core programming language covering all protocol-relevant operations (e.g., network I/O, cryptographic primitives), and those relevant for modeling an attacker (e.g., corruption). The language's operational semantics supports thread-local state and explicitly maintains a shared global trace. The thread-local state models the state of each participant and is manipulated via assignments in the participants' implementations. In contrast, the global trace is maintained automatically by our semantics and extended whenever a relevant protocol operation is executed. We then define a Hoare logic to enable modularly verifying each participant implementation. The logic natively supports our methodology's local snapshots and the global trace. We prove that this logic is sound w.r.t. the operational semantics using a standard rule induction. Thereby, we obtain the guarantee that locally-verified participants, if composed with the attacker to a concurrent system, maintain the global trace invariant in all possible interleavings. As one would expect, obtaining this global guarantee turned out to be the most challenging step in the soundness proof. Formally connecting our dedicated Hoare logic to a standard separation logic is straightforward, based on the encoding discussed throughout the paper (using the heap to store the thread-local state, a ghost lock to synchronize access, and a lock invariant to constrain the trace and all local snapshots).

8 RELATED WORK

Much prior work on the verification of cryptographic protocols exists, and surveys [47–49] provide an extensive overview of the field. We focus on *modular verification of symbolic security properties*, and discuss the most closely related work first: techniques for verifying security of *realistic protocol implementations*.

Dupressoir et al. [32] use VCC [33] to verify memory safety, non-injective agreement, and (via an external argument in Coq) weak secrecy, of two protocols implemented in C: RPC and Otway-Rees. To our knowledge, they are the first to encode a global protocol trace ("log") as a concurrent data structure. We generalize this idea to separation logic to make it much more widely applicable, because their encoding relies on C's `volatile` fields and a VCC-specific program logic, neither of which are (widely) available in other languages and verifiers. Moreover, since their logic (unlike separation logic) does not provide linear resources, proving injective agreement would require a nontrivial extension of their work. Their set-based trace encoding prevents proving, e.g., forward secrecy (which we do); they account for principal corruption, but not session corruption (we account for both). Polikarpova et al. [50] extend this work by incorporating stepwise refinement to formally connect a model to an existing implementation, all encoded in VCC. This refinement decomposes the verification into smaller steps, but incurs additional overhead. Moreover, they remove the need for external arguments when proving weak secrecy. They verify the latter, and a variant of authentication, for a small stateful subset of TPM 2.0.

Vanspauwen et al. [51], like us, use a separation-logic-based verifier (VeriFast [21]), but they do not model a global trace. Consequently, properties that are commonly expressed over a trace potentially need to be assembled from individual assertions. They propose an extended symbolic model that strengthens attackers by permitting byte-wise manipulations, such as splitting and reconcatenating byte sequences, in addition to the usual symbolic manipulations. Our attacker operates on terms (standard for symbolic cryptography) but we could adapt their extension. They specify PolarSSL's API using this extended model, and then verify secrecy and non-injective agreement of an NSL-implementation (and a few less complex protocols). Unlike us, they do not consider session corruption.

Arquint et al. [20] suggest a two-step approach: First, a protocol *model* is verified via Tamarin [52]. If successful, a separation logic predicate (one per participant) with I/O specifications [53] is generated, specifying which I/O operations preserve the security properties of the model. Second, existing implementations of the protocol can be verified against these predicates; if successful, the implementation is guaranteed to satisfy the model's properties. This two-step workflow achieves tool reuse—Tamarin, and suitable separation logic verifiers—but requires expertise in two different fields of formal reasoning, and the existence of a Tamarin protocol model. Moreover, limitations of Tamarin (e.g., difficulties when verifying protocols with loops), and of the I/O specifications (unclear how to generate specifications suitable for a concurrent

implementation) may prevent verifying corresponding implementations. Similar limitations apply to Sprenger et al.'s work [34], which connects protocol models verified in Isabelle/HOL [54] via I/O specifications to separation logic verifiers.

Bhargavan et al. [9] suggest DY*: a framework for verifying protocols implemented in F* [12], a functional language that enables type-system-based proofs, e.g., using monadic effects and refinement types. DY* introduces the idea of a parametric library for reducing the per-protocol proof effort; an idea we adopted. DY*'s type system is tailored to F*, whereas our methodology supports a wide range of languages and tools. Moreover, by building on separation logic, we are able to prove stronger properties, in particular, injective agreement. Our methodology can be applied directly to existing implementations, as we demonstrate in the WireGuard case study. In contrast, DY* supports code generation, but additionally requires a hand-written (and partly protocol-specific) runtime wrapper [10]. Included in DY*'s case study is the first automated verification of Signal [2] that proves forward and post-compromise security over an unbounded number of protocol messages. Our main case study is WireGuard, for which we prove, also for an unbounded number of messages, forward secrecy and injective agreement with AKC resilience. Soundness of DY*'s global protocol trace depends on a specific coding discipline (one method per protocol step) that is not automatically enforced. If missed, the attacker is accidentally restricted, and security properties can be proven incorrectly.

An earlier line of work (e.g., [37, 55, 56]) verifies security of functional programs written in F# using the F7 type checker [55], but does not integrate equational theories, and has limited support for mutable state. Moreover, this work does not model the global protocol traces and, thus, states security properties only implicitly.

Küsters et al. [57] share our goal of reusing existing program analyzers and suggest an approach that enables non-interference checkers to establish computational indistinguishability results for sequential programs. To account for closed-system assumptions (typically made by such checkers) in the presence of an attacker-controlled environment, they restrict interaction with the latter to static, exception-free methods, and primitive (i.e., value) types. How to extend their approach to trace-based properties and concurrent programs remains unclear.

Several security property verifiers exist that (unlike us) do not reuse existing program analyzers, e.g., Csur [58] and ASPIER [59] (for C), and JavaSec [60] (for Java). However, to reduce development costs, such domain-specific tools typically only implement semantics of a restricted language subset and, e.g., assume crucial properties such as memory safety (which may render implementations insecure, e.g., due to buffer overflows).

Prior work [11, 39, 40, 42, 61, 62] on verifying properties of WireGuard (our main case study) is concerned with verifying models of the protocol, not existing implementations.

Finally, a large body of work is concerned with mechanizing the verification of computational (rather than symbolic) properties; see aforementioned surveys for details. This line of work establishes stronger guarantees by making weaker, more realistic cryptographic assumptions. For instance, Owl [63] allows one to verify computational security of protocols written in a dedicated language. Like in our work, their proofs are automated and compositional. However, due to probabilistic reasoning, verifying computational security is generally more challenging than reasoning about symbolic terms, and we are not aware of tools for modularly verifying computational security properties of existing implementations. Recently, the first separation logics for probabilistic reasoning have been proposed [64–66], but we are not aware of automated verifiers for such logics.

9 CONCLUSIONS

We presented a methodology for the modular verification of security protocol implementations. It enables proving strong security properties for realistic protocol implementations in the presence of a network-controlling attacker. By employing separation logic, we support efficient implementations using heap data structures, side effects, concurrency, etc. Encapsulating the global trace in a concurrent ghost data structure and our use of invariants over local snapshots allow our methodology to support arbitrary code structures and data representations, which is crucial for targeting existing implementations. Separation logic also allows us to specify resources in the trace invariant to express uniqueness of protocol-specific events, which is key to modularly proving injective agreement.

We have instantiated our methodology for Go and C and two corresponding verifiers. Our case studies on NSL, signed DH, and WireGuard demonstrate that our methodology handles existing and interoperable implementations of protocols with strong security properties, such as forward secrecy and injective agreement.

For future work, we plan to integrate our methodology with formally-verified cryptographic libraries to further reduce our trust assumptions. It would also be interesting to advance towards the computational model of cryptography by combining our work with probabilistic separation logic.

ACKNOWLEDGMENTS

We thank the Werner Siemens-Stiftung (WSS) for their generous support of this project. We are grateful to Ralf Sasse for the helpful discussions and feedback on an earlier draft of this paper; Thibault Dardinier, for his suggestions for the soundness proof; Hugo Queinnec, for his critical questions; and the anonymous reviewers for their helpful feedback that helped us sharpen our contributions.

REFERENCES

[1] J. A. Donenfeld, "WireGuard: Next generation kernel network tunnel," in *NDSS*. The Internet Society, 2017.
[2] M. Marlinspike and T. Perrin. The X3DH key agreement protocol (revision 1). [Online]. Available: https://www.signal.org/docs/specifications/x3dh/
[3] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR," in *TACAS*, ser. LNCS, vol. 1055. Springer, 1996, pp. 147–166.
[4] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, no. 12, pp. 993–999, 1978.
[5] CVE, "CVE-2014-0160," 2013. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2014-0160
[6] ——, "CVE-2021-40823," 2021. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2021-40823
[7] D. Pozza, R. Sisto, and L. Durante, "Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus," in *AINA*. IEEE Computer Society, 2004, pp. 400–405.
[8] D. Cadé and B. Blanchet, "From computationally-proved protocol specifications to implementations," in *ARES*. IEEE Computer Society, 2012, pp. 65–74.
[9] 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 *EuroS&P*. IEEE, 2021, pp. 523–542.

[10] ——, "An in-depth symbolic security analysis of the ACME standard," in *CCS*. ACM, 2021, pp. 2601–2617.

[11] S. Ho, J. Protzenko, A. Bichhawat, and K. Bhargavan, "Noise*: A library of verified high-performance secure channel protocol implementations," in *S&P*. IEEE, 2022, pp. 107–124.

[12] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin, "Dependent types and multi-monadic effects in F*," in *POPL*. ACM, 2016, pp. 256–270.

[13] 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.

[14] N. O'Shea, "Using Elyjah to analyse Java implementations of cryptographic protocols," in *FCS-ARSPA-WITS-2008*, 2008.

[15] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Computational verification of C protocol implementations by symbolic execution," in *CCS*. ACM, 2012, pp. 712–723.

[16] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *EuroS&P*. IEEE, 2017, pp. 435–450.

[17] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *S&P*. IEEE Computer Society, 2017, pp. 483–502.

[18] R. Sisto, P. B. Copet, M. Avalle, and A. Pironti, "Formally sound implementations of security protocols with JavaSPI," *Formal Aspects Comput.*, vol. 30, no. 2, pp. 279–317, 2018.

[19] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in WebAssembly," in *S&P*. IEEE, 2019, pp. 1256–1274.

[20] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. A. Basin, and P. Müller, "Sound verification of security protocols: From design to interoperable implementations," in *SP*. IEEE, 2023, pp. 1077–1093.

[21] 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*, ser. LNCS, vol. 6617. Springer, 2011, pp. 41–55.

[22] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of Go programs," in *CAV*, ser. LNCS, vol. 12759. Springer, 2021, pp. 367–379.

[23] S. Blom and M. Huisman, "The VerCors tool for verification of concurrent programs," in *FM*, ser. LNCS, vol. 8442. Springer, 2014, pp. 127–131.

[24] J. F. Santos, P. Maksimovic, D. Naudziuniene, T. Wood, and P. Gardner, "JaVerT: JavaScript verification toolchain," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 50:1–50:33, 2018.

[25] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 147:1–147:30, 2019.

[26] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *CSL*, ser. LNCS, vol. 2142. Springer, 2001, pp. 1–19.

[27] J. C. Reynolds, "Separation Logic: A logic for shared mutable data structures," in *LICS*. IEEE Computer Society, 2002, pp. 55–74.

[28] J. Filliâtre, L. Gondelman, and A. Paskevich, "The spirit of ghost code," in *CAV*, ser. LNCS, vol. 8559. Springer, 2014, pp. 1–16.

[29] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[30] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[31] L. Arquint, M. Schwerhoff, V. Mehta, and P. Müller, "A generic methodology for the modular verification of security protocol implementations," Sep. 2023, artifact containing the reusable verification libraries and the case studies. [Online]. Available: https://doi.org/10.5281/zenodo.8330913

[32] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," in *CSF*. IEEE Computer Society, 2011, pp. 3–17.

[33] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *TPHOLs*, ser. LNCS, vol. 5674. Springer, 2009, pp. 23–42.

[34] 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.

[35] J. Boyland, "Checking interference with fractional permissions," in *SAS*, ser. LNCS, vol. 2694. Springer, 2003, pp. 55–72.

[36] G. Lowe, "A hierarchy of authentication specification," in *CSFW*. IEEE Computer Society, 1997, pp. 31–44.

[37] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *POPL*. ACM, 2010, pp. 445–456.

[38] M. J. Parkinson and G. M. Bierman, "Separation Logic and abstraction," in *POPL*. ACM, 2005, pp. 247–258.

[39] B. Dowling and K. G. Paterson, "A cryptographic analysis of the WireGuard protocol," in *ACNS*, ser. LNCS, vol. 10892. Springer, 2018, pp. 3–21.

[40] B. Lipp, B. Blanchet, and K. Bhargavan, "A mechanised cryptographic proof of the WireGuard virtual private network protocol," in *EuroS&P*. IEEE, 2019, pp. 231–246.

[41] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. Basin, and P. Müller, "Sound verification of security protocols: From design to interoperable implementations," Aug. 2022, Tamarin model & verified Go implementation of the WireGuard VPN key exchange protocol. [Online]. Available: https://doi.org/10.5281/zenodo.7409524

[42] G. Girol, L. Hirschi, R. Sasse, D. Jackson, C. Cremers, and D. A. Basin, "A spectral analysis of Noise: A comprehensive, automated, formal analysis of Diffie-Hellman protocols," in *USENIX Security Symposium*. USENIX Association, 2020, pp. 1857–1874.

[43] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of TLS 1.3," in *CCS*. ACM, 2017, pp. 1773–1788.

[44] D. A. Basin, C. Cremers, and M. Horvat, "Actor key compromise: Consequences and countermeasures," in *CSF*. IEEE Computer Society, 2014, pp. 244–258.

[45] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel, "Vst-floyd: A separation logic tool to verify correctness of C programs," *J. Autom. Reason.*, vol. 61, no. 1-4, pp. 367–422, 2018.

[46] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Z. Béguelin, "EverCrypt: A fast, verified, cross-platform cryptographic provider," in *S&P*. IEEE, 2020, pp. 983–1002.

[47] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," in *S&P*. IEEE, 2021, pp. 777–795.

[48] M. Avalle, A. Pironti, and R. Sisto, "Formal verification of security protocol implementations: a survey," *Formal Aspects Comput.*, vol. 26, no. 1, pp. 99–123, 2014.

[49] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *POST*, ser. LNCS, vol. 7215. Springer, 2012, pp. 3–29.

[50] N. Polikarpova and M. Moskal, "Verifying implementations of security protocols by refinement," in *VSTTE*, ser. LNCS, vol. 7152. Springer, 2012, pp. 50–65.

[51] G. Vanspauwen and B. Jacobs, "Verifying protocol implementations by augmenting existing cryptographic libraries with specifications," in *SEFM*, ser. LNCS, vol. 9276. Springer, 2015, pp. 53–68.

[52] B. Schmidt, S. Meier, C. Cremers, and D. A. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *CSF*. IEEE Computer Society, 2012, pp. 78–94.

[53] W. Penninckx, B. Jacobs, and F. Piessens, "Sound, modular and compositional verification of the input/output behavior of programs," in *ESOP*, ser. LNCS, vol. 9032. Springer, 2015, pp. 158–182.

[54] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[55] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," in *CSF*. IEEE Computer Society, 2008, pp. 17–32.

[56] K. Bhargavan, C. Fournet, and N. Guts, "Typechecking higher-order security libraries," in *APLAS*, ser. LNCS, vol. 6461. Springer, 2010, pp. 47–62.

[57] R. Küsters, T. Truderung, and J. Graf, "A framework for the cryptographic verification of Java-like programs," in *CSF*. IEEE Computer Society, 2012, pp. 198–212.

[58] J. Goubault-Larrecq and F. Parrennes, "Cryptographic protocol analysis on real C code," in *VMCAI*, ser. LNCS, vol. 3385. Springer, 2005, pp. 363–379.

[59] S. Chaki and A. Datta, "ASPIER: An automated framework for verifying security protocol implementations," in *CSF*. IEEE Computer Society, 2009, pp. 172–185.

[60] J. Jürjens, "Security analysis of crypto-based Java programs using automated theorem provers," in *ASE*. IEEE Computer Society, 2006, pp. 167–176.

[61] J. A. Donenfeld and K. Milner. Formal verification of the WireGuard protocol. [Online]. Available: https://www.wireguard.com/papers/wireguard-formal-verification.pdf

[62] N. Kobeissi, G. Nicolas, and K. Bhargavan, "Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols," in *EuroS&P*. IEEE, 2019, pp. 356–370.

[63] J. Gancher, S. Gibson, P. Singh, S. Dharanikota, and B. Parno, "Owl: Compositional verification of security protocols via an information-flow type system," in *SP*. IEEE, 2023, pp. 1130–1147.

[64] J. Tassarotti and R. Harper, "A Separation Logic for concurrent randomized programs," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 64:1–64:30, 2019.

[65] K. Batz, B. L. Kaminski, J. Katoen, C. Matheja, and T. Noll, "Quantitative Separation Logic: A logic for reasoning about probabilistic pointer programs," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 34:1–34:29, 2019.

[66] G. Barthe, J. Hsu, and K. Liao, "A probabilistic Separation Logic," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 55:1–55:30, 2020.

[67] V. Vafeiadis, "Concurrent separation logic and operational semantics," in *MFPS*, ser. Electronic Notes in Theoretical Computer Science, vol. 276. Elsevier, 2011, pp. 335–351.

A SOUNDNESS PROOF SKETCH

Intuitively, we argue soundness of our methodology by showing that, given a distributed system of verified protocol implementations and an arbitrary attacker, the systems' set of possible executions is a subset of the executions permitted by the verification trace invariant, which in turn is a subset of the executions that satisfy the desired security properties. To achieve this, we define a minimal but concurrent programming language with primitives for security-relevant operations such as sending messages or creating nonces, and a corresponding operational semantics (Sec. A.1) that reflects these operations on a global (i.e. system-wide shared) trace. We then define an axiomatic semantics (Sec. A.2) parameterized with a trace invariant that we prove sound w.r.t. the operational semantics. I.e., we show that the axiomatic proof rules enforce the trace invariant. Since the global trace maintained by the operational semantics reflects all relevant protocol steps, and because our axiomatic semantics is proven sound, we can conclude that the aforementioned trace inclusion holds (Sec. A.3). In each subsection, we additionally relate the semantics defined for the proof sketch with the verification performed by an off-the-shelf separation-logic verifier (such as Gobra) against our reusable verification library.

A.1 Language and Operational Semantics

On a high-level, we consider a distributed system consisting of multiple components: either instances of a protocol implementation, i.e. participants, or the attacker. Our programming language does not support user-defined shared variables or a heap, and each participant executes its commands in its own local state. However, security-relevant commands additionally mutate the global trace to reflect the performed operation.

Consequently, our system's configurations comprise a local configuration per component, and the global trace τ. A local configuration for a protocol participant i is characterized by its local command C_i and local state σ_i. The local configuration for the attacker is similar, but additionally contains a knowledge set k_a that stores all symbolic terms that are known to the attacker.

Definition 1. ***Local program states.*** *Local program states, ranged over by σ, are total functions from local variables (in the set PVars) to values (in the set PVals).*

$$PStates \triangleq PVars \longrightarrow PVals$$

We define our programming language such that it directly works with symbolic terms instead of bytes, which avoids having to complicate the semantics to reflect the orthogonal issue of mapping between the bytes and terms.

Definition 2. ***System configurations.*** *A configuration of our distributed system has the shape*

$$\langle\langle C_1, \sigma_1\rangle, \cdots, \langle C_n, \sigma_n\rangle, \langle C_a, \sigma_a\rangle, k_a, \tau\rangle$$

where $\langle C_i, \sigma_i\rangle$ denotes the local command and local state of participant i, $\langle C_a, \sigma_a\rangle$ denotes the local command and local state of the attacker a, k_a is the attacker's knowledge set and τ denotes the system's global trace.

Observe that the attacker's knowledge set k_a is not part of the attacker's local configuration $\langle C_a, \sigma_a\rangle$, even though only commands executed by the attacker possibly modify k_a. By using the same shape for local configurations of participants and the attacker, both can apply the same operational semantics rules, e.g., for sequential composition.

Definition 3. ***Programming language.*** *We consider the following programming language, where C ranges over commands, x and $\vec{x}$ over variables and lists of variables in the set PVars, respectively, and e over expressions (modeled as total functions from PStates to PVals):*

$$\begin{aligned} C \triangleq\ & \textbf{skip} \mid C; C \mid \textbf{if}\ (e)\ \{C\}\ \textbf{else}\ \{C\} \mid \textbf{while}\ (e)\ \{C\} \mid \\ & x := e \mid \mathit{send}(e) \mid x := \mathit{recv}() \mid x := \mathit{nonce}() \mid \\ & x := \mathit{hash}(e) \mid x := \mathit{pk}(e) \mid x := \mathit{enc}(e, e) \mid x, x := \mathit{dec}(e, e) \mid \\ & \mathit{drop}(e) \mid \mathit{learn}(e) \mid x := \mathit{choose}() \mid \mathit{corrupt}(e) \mid \\ & \textbf{fork}\ (\vec{x})\ \{C\} \end{aligned}$$

Besides standard commands, such as sequential composition and assignment, the programming language provides several commands essential for protocol implementations: for sending and receiving a network message, for generating a nonce, hashing a term, generating a public key corresponding to a given secret key (*pk*), and encrypting and decrypting a term with a key. Additionally, the programming language provides commands only available to the attacker: dropping a message from the network, adding the value of a local variable to the attacker knowledge (*learn*), nondeterministically obtaining a term from the attacker knowledge (*choose*), and corrupting the state of specific participant (each participant has a unique id/index).

Finally, fork starts a new thread executing the provided command, which corresponds to spawning a new participant or the attacker. The new thread operates on its own local state, which initially maps the variables in $\vec{x}$ to the same values as the state in which the fork command is executed. This command is used to bootstrap the distributed system, as discussed in Sec. A.3.

The expression language comprises symbolic terms for booleans and integers, and the usual operations thereon. We assume well-typed programs, e.g., that if-conditions are of type boolean.

Definition 4. ***Operational semantics*** *Fig. 15 defines the small-step operational semantics for our programming language.*

The rules for standard commands such as sequential composition and conditionals, are as expected, and we will thus only discuss non-standard aspects of our programming language.

Global trace. Recall from Sec. 3.3 that in our verification methodology (as implemented in Gobra), we use a concurrent ghost data structure with ghost locks to manage the global trace. In our operational semantics, we instead represent the trace as the dedicated element τ in the system's state. Irregardless of the technical implementation we must ensure three crucial properties: (1) Each operation may only append a single trace events. In our methodology, this is checked via a suitable proof obligation upon lock release; in our operational semantics, each rule adds at most one event. (2) To ensure monotonicity, the trace may only grow. Checked upon lock release in our methodology; in our operational semantics, no rule shortens the trace. (3) Each single operation must preserve the trace invariant. Checked upon lock release in our methodology; in our operational semantics, this is part of the soundness theorem (cf. Thm. 1).

$$\frac{\langle C_i, \langle \sigma_i, \tau \rangle\rangle \to \langle C_i', \langle \sigma_i', \tau' \rangle\rangle}{\langle \cdots, \langle C_i, \sigma_i \rangle, \cdots, k_a, \tau \rangle \to \langle \cdots, \langle C_i', \sigma_i' \rangle, \cdots, k_a, \tau' \rangle} \text{(Local)} \qquad \frac{\langle C_a, \langle \sigma_a, k_a, \tau \rangle\rangle \to \langle C_a', \langle \sigma_a', k_a', \tau' \rangle\rangle}{\langle \cdots, \langle C_a, \sigma_a \rangle, k_a, \tau \rangle \to \langle \cdots, \langle C_a', \sigma_a' \rangle, k_a', \tau' \rangle} \text{(Attacker)}$$

$$\frac{}{\langle \textbf{skip}, \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma, \tau \rangle\rangle} \text{(Skip)} \qquad \frac{\langle C_1, \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma', \tau' \rangle\rangle}{\langle C_1; C_2, \langle \sigma, \tau \rangle\rangle \to \langle C_2, \langle \sigma', \tau' \rangle\rangle} \text{(Seq1)} \qquad \frac{\langle C_1, \langle \sigma, \tau \rangle\rangle \to \langle C_1', \langle \sigma', \tau' \rangle\rangle}{\langle C_1; C_2, \langle \sigma, \tau \rangle\rangle \to \langle C_1'; C_2, \langle \sigma', \tau' \rangle\rangle} \text{(Seq2)}$$

$$\frac{}{\langle \textbf{if } (e) \ \{C_1\} \textbf{ else } \{C_2\}, \langle \sigma, \tau \rangle\rangle \to \langle C_1, \langle \sigma, \tau \rangle\rangle} \text{(If1)}^{e(\sigma_i)=True()} \qquad \frac{}{\langle \textbf{if } (e) \ \{C_1\} \textbf{ else } \{C_2\}, \langle \sigma, \tau \rangle\rangle \to \langle C_2, \langle \sigma, \tau \rangle\rangle} \text{(If2)}^{e(\sigma_i)\neq True()}$$

$$\frac{}{\langle \textbf{while } (e) \ \{C\}, \langle \sigma, \tau \rangle\rangle \to \langle \textbf{if } (e) \ \{C; \textbf{while } (e) \ \{C\}\} \textbf{ else } \{\textbf{skip}\}, \langle \sigma, \tau \rangle\rangle} \text{(While)}$$

$$\frac{}{\langle x \coloneqq e, \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[x \mapsto e(\sigma)], \tau \rangle\rangle} \text{(Assign)}$$

$$\frac{}{\langle send(e), \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[snap \mapsto \tau + Send(e(\sigma))], \tau + Send(e(\sigma)) \rangle\rangle} \text{(Send)}$$

$$\frac{}{\langle x \coloneqq recv(), \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[x \mapsto v], \tau \rangle\rangle} \text{(Recv)}^{v \in msgs(\tau)}$$

$$\frac{}{\langle x \coloneqq nonce(), \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[x \mapsto v, snap \mapsto \tau + Nonce(v)], \tau + Nonce(v) \rangle\rangle} \text{(NonceGen)}^{fresh(v,\tau)}$$

$$\frac{}{\langle x \coloneqq hash(e), \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[x \mapsto Hash(e(\sigma))], \tau \rangle\rangle} \text{(Hash)} \qquad \frac{}{\langle x \coloneqq pk(e), \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[x \mapsto Pk(e(\sigma))], \tau \rangle\rangle} \text{(Pk)}$$

$$\frac{}{\langle x \coloneqq enc(e_1, e_2), \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[x \mapsto Enc(e_1(\sigma), e_2(\sigma))], \tau \rangle\rangle} \text{(Enc)}$$

$$\frac{}{\langle x, ok \coloneqq dec(e_1, e_2), \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[x \mapsto v, ok \mapsto True()], \tau \rangle\rangle} \text{(DecSucc)}^{\exists v. e_2(\sigma)=Enc(Pk(e_1(\sigma)),v)}$$

$$\frac{}{\langle x, ok \coloneqq dec(e_1, e_2), \langle \sigma, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[ok \mapsto False()], \tau \rangle\rangle} \text{(DecFail)}^{\forall v. e_2(\sigma)\neq Enc(Pk(e_1(\sigma)),v)}$$

$$\frac{}{\langle drop(e), \langle \sigma, k, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[snap \mapsto \tau + Drop(e(\sigma))], k, \tau + Drop(e(\sigma)) \rangle\rangle} \text{(Drop)}$$

$$\frac{}{\langle learn(e), \langle \sigma, k, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma, k \cup \{e(\sigma)\}, \tau \rangle\rangle} \text{(Learn)} \qquad \frac{}{\langle x \coloneqq choose(), \langle \sigma, k, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma[x \mapsto v], k, \tau \rangle\rangle} \text{(Choose)}^{v \in k}$$

$$\frac{}{\langle \cdots, \langle C_i, \sigma_i \rangle, \cdots, \langle corrupt(i); C_a', \sigma_a \rangle, k_a, \tau \rangle \to \langle \cdots, \langle C_i, \sigma_i \rangle, \cdots, \langle C_a', \sigma_a \rangle, k_a \cup val(\sigma_i), \tau + Corrupt(i, val(\sigma_i)) \rangle\rangle} \text{(Corrupt)}$$

$$\frac{}{\langle \cdots, \langle \textbf{fork } (x_1, \cdots, x_n) \ \{C\}; C', \sigma_i \rangle, \cdots, k_a, \tau \rangle \to \langle \cdots, \langle C', \sigma_i \rangle, \cdots, \langle C, [x_1 \mapsto \sigma_i(x_1), \cdots, x_n \mapsto \sigma_i(x_n), snap \mapsto \sigma_i(snap)] \rangle, k_a, \tau \rangle} \text{(Fork)}$$

Figure 15: Small-step semantics. Since expressions are functions from states to values, $e(\sigma)$ denotes the evaluation of expression e in state σ. $\sigma[x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$ denotes state update: a state that, for all i, $1 \leq i \leq n$, yields v_i for x_i, and the value in σ for all other variables. Appending to a trace is denoted by +, e.g., $\tau + Nonce(v)$. $\langle \epsilon, \langle \sigma, \tau \rangle\rangle$ denotes a terminal state.

Local snapshots. Recall from cf. Sec. 3.4 that each participant has a trace snapshot, which enables participants to keep local invariants of trace prefixes. To enable corresponding assertions in our program logic (Sec. A.2), our operational semantics provide a local variable *snap* that is treated special in two ways: local states σ map *snap* to a sequence of trace events (not to a value in *PVals*), and program commands may not use (in particular, modify) *snap* (a straightforward syntactical constraint).

Projecting system configurations. The Local and Attacker rule project project a system configuration down to a participant- and attacker-local configuration, respectively. Besides Corrupt and Fork, all other rules then operate on either a participant- or attacker-local configuration, depending on whether a command can be executed by participants and the attacker, or only by the attacker.

Network messages. All operations modifying the network state, i.e., sending and dropping a message, are recorded on the global trace τ, and we can thus compute the set of receivable messages as follows:

Definition 5. ***Messages on the network.***

$$msgs(\tau) \triangleq \{m \mid \forall m.\, Send(m) \in \tau \land Drop(m) \notin \tau\}$$

Consequently, function $msgs(\tau)$ occurs in rule Recv's side-condition to constrain the set of messages to receive from Without loss of generality, this side-condition implies that we consider only non-blocking traces, i.e., where *recv*() is invoked when $msgs(\tau)$ is non-empty.

Nonce freshness. NonceGen rule's side condition captures our perfect cryptography assumption that generated nonces are always fresh.

Definition 6. ***Freshness of nonces.*** *Since all previously generated nonces have been recorded on the trace τ, we can define freshness of a nonce v on the global trace τ as follows:*

$$fresh(v, \tau) \triangleq v \notin \{n \mid \forall n, l.\, Nonce(n, l) \in \tau\}$$

Corruption. The Corrupt rule expresses that the attacker knowledge is extended by all terms in the state σ_i of the corrupted participant i. The attacker can make use of these newly learnt terms by executing $x := choose()$ that non-deterministically picks a term in the attacker knowledge and assigns it to the local variable x.

Note that the Corrupt rule requires command *corrupt*() to be followed by another command, skip. Baking in sequential composition avoids the need for further sequential rules that only differ in the kind of configuration (system vs. local) they operate on. Rule Fork is defined analogously.

Forking. Rule Fork extends the system configuration by another local configuration with the forked command to execute and the new thread's initial state. This new state maps the variables x_1 to x_n and *snap* to the same value as σ_i, i.e., the state in which the fork command is executed, which enables, e.g., the sharing of public keys.

A.2 Program Logic

We now present a program logic that enables local reasoning about each participant, while guaranteeing that the trace invariant is maintained even when composing arbitrarily many verified participants and the attacker to a distributed system. We first present several auxiliary definitions and lemmas, and then the logic's proof rules.

Definition 7. ***Trace prefix.*** *We define the following predicate over two traces expressing that τ_1 is a prefix of τ_2*

$$prefix(\tau_1, \tau_2) \triangleq \exists p.\, \tau_1 + p = \tau_2$$

where p is a possibly empty sequence of trace events.

Lemma 1. ***Prefix reflexivity.***

$$\forall \tau.\, prefix(\tau, \tau)$$

Proof sketch. Pick p to be the empty sequence in Def. 7. □

Lemma 2. ***Prefix transitivity.***

$$\forall \tau_1, \tau_2, \tau_3.\, prefix(\tau_1, \tau_2) \land prefix(\tau_2, \tau_3) \implies prefix(\tau_1, \tau_3)$$

Proof sketch.

$$\begin{aligned} & prefix(\tau_1, \tau_2) \land prefix(\tau_2, \tau_3) \\ & \overset{\text{def}}{\iff} \exists p_1, p_2.\, \tau_1 + p_1 = \tau_2 \land \tau_2 + p_2 = \tau_3 \\ & \implies \exists p_1, p_2.\, \tau_1 + p_1 + p_2 = \tau_3 \\ & \overset{\text{def}}{\iff} prefix(\tau_1, \tau_3) \end{aligned}$$

where we pick in the last step p in Def. 7 to be $p_1 + p_2$. □

Inspired by Vafeiadis [67], we express the semantics of judgements in our logic in terms of configuration safety, which we define next. Intuitively, $safe_n(i, C, \sigma, Q, \tau)$ expresses that it is safe to execute command C, as the ith component of the distributed system, and for n execution steps starting in a state σ; and if the command is fully executed, the predicate Q holds in the resulting final state. Furthermore, if new threads have been forked as part of executing C then it is safe to execute these forked components, too. Since we are ultimately interested in the effects on the global trace τ, configuration safety includes trace invariant ρ maintenance. A judgement $\models [P]\ C\ [Q]$ then expresses that it is safe to execute the command C for an arbitrary number of execution steps and from any initial state satisfying the predicate P.

Definition 8. ***Configuration safety.***

$safe_0(i, C, \sigma, Q, \tau)$ holds always.

$safe_{n+1}(i, C, \sigma, Q, \tau)$ holds if and only if

(i) $C = \epsilon \implies Q(\sigma)$ *and*

(ii)
$$\begin{aligned} \forall \vec{C}, \vec{C'}, \vec{\sigma}, \vec{\sigma'}, k_a, k_a', \tau'.\, & i \le |\vec{C}| = |\vec{\sigma}| \le |\vec{C'}| = |\vec{\sigma'}| \land \\ & \vec{C}_i = C \land \vec{C}_i' \neq C \land \vec{\sigma}_i = \sigma \land \\ & \rho(\tau) \land prefix(snap(\sigma), \tau) \land \\ & \langle \overrightarrow{\langle C, \sigma \rangle}, k_a, \tau \rangle \to \langle \overrightarrow{\langle C', \sigma' \rangle}, k_a', \tau' \rangle \\ \implies & \rho(\tau') \land prefix(\tau, \tau') \land prefix(snap(\vec{\sigma}_i'), \tau') \land \\ & k_a \subseteq k_a' \land safe_n(i, \vec{C}_i', \vec{\sigma}_i', Q, \tau') \land \\ & \left(\bigwedge_{|\vec{C}| < j \le |\vec{C'}|} safe_n(j, \vec{C}_j', \vec{\sigma}_j', True(), \tau') \land prefix(snap(\vec{\sigma}_j'), \tau') \right) \end{aligned}$$

$$\frac{}{\vdash [P]\ \ skip\ \ [P]}\ (\textsc{Skip}) \qquad \frac{\vdash [P]\ \ C_1\ \ [R] \qquad \vdash [R]\ \ C_2\ \ [Q]}{\vdash [P]\ \ C_1; C_2\ \ [Q]}\ (\textsc{Seq}) \qquad \frac{P \models P' \quad Q' \models Q \quad \vdash [P']\ \ C\ \ [Q']}{\vdash [P]\ \ C\ \ [Q]}\ (\textsc{Cons})$$

$$\frac{\vdash [e \wedge P]\ \ C_1\ \ [Q] \qquad \vdash [\neg e \wedge P]\ \ C_2\ \ [Q]}{\vdash [P]\ \ \textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \ [Q]}\ (\textsc{If}) \qquad \frac{\vdash [e \wedge P]\ \ C\ \ [P]}{\vdash [P]\ \ \textbf{while}\ (e)\ \{C\}\ \ [\neg e \wedge P]}\ (\textsc{While}) \qquad \frac{}{\vdash [P[e/x]]\ \ x := e\ \ [P]}\ (\textsc{Assign})$$

$$\frac{}{\vdash [ext(Send(e), snap) \wedge \forall p.\, P[snap + p + Send(e)/snap]]\ \ send(e)\ \ [P]}\ (\textsc{Send}) \qquad \frac{}{\vdash [\forall x.\, P]\ \ x := recv()\ \ [P]}\ (\textsc{Recv})$$

$$\frac{}{\vdash [ext(Nonce(x), snap) \wedge \forall p, x.\, P[snap + p + Nonce(x)/snap]]\ \ x := nonce()\ \ [P]}\ (\textsc{NonceGen})$$

$$\frac{}{\vdash [P[Hash(e)/x]]\ \ x := hash(e)\ \ [P]}\ (\textsc{Hash}) \qquad \frac{}{\vdash [P[Pk(e)/x]]\ \ x := pk(e)\ \ [P]}\ (\textsc{Pk}) \qquad \frac{}{\vdash [P[Enc(e_1, e_2)/x]]\ \ x := enc(e_1, e_2)\ \ [P]}\ (\textsc{Enc})$$

$$\frac{}{\vdash [\forall x.\, P[True()/ok][e_2/Enc(Pk(e_1), x)] \wedge P[False()/ok]]\ \ x, ok := dec(e_1, e_2)\ \ [P]}\ (\textsc{Dec})$$

$$\frac{}{\vdash [ext(Drop(e), snap) \wedge \forall p.\, P[snap + p + Drop(e)/snap]]\ \ drop(e)\ \ [P]}\ (\textsc{Drop})$$

$$\frac{}{\vdash [P]\ \ learn(e)\ \ [P]}\ (\textsc{Learn}) \qquad \frac{}{\vdash [\forall x.\, P]\ \ x := choose()\ \ [P]}\ (\textsc{Choose})$$

$$\frac{}{\vdash [(\forall v.\, ext(Corrupt(e, v), snap)) \wedge (\forall p, v.\, P[snap + p + Corrupt(e, v)/snap])]\ \ corrupt(e)\ \ [P]}\ (\textsc{Corrupt})$$

$$\frac{fv(R) \subseteq \vec{x} \quad P \models R \qquad \vdash [R]\ \ C\ \ [True()] \qquad \vdash [P]\ \ C'\ \ [Q]}{\vdash [P]\ \ \textbf{fork}\ (\vec{x})\ \{C\}; C'\ \ [Q]}\ (\textsc{Fork})$$

Figure 16: The proof rules.

where $|\vec{V}|$ and $\vec{V}_i$ denote the length and element at index i of a vector V, resp., and $\overrightarrow{\langle C, \sigma \rangle}$ is syntactic sugar for $\langle \vec{C}_1, \vec{\sigma}_1 \rangle \cdots \langle \vec{C}_{|\vec{C}|}, \vec{\sigma}_{|\vec{\sigma}|} \rangle$.

Definition 9. ***Validity.***

$$\models [P]\ \ C\ \ [Q] \triangleq \forall n, i, \sigma, \tau.\, P(\sigma) \implies safe_n(i, C, \sigma, Q, \tau)$$

Executing zero steps is vacuously safe. Executing $n + 1$ steps is safe (i) if the command is already fully executed and the predicate Q satisfied; and otherwise (ii) if there is a transition to $\vec{C}_i'$ that maintains the trace invariant ρ, the necessary monotonicity properties (on snapshot, trace, and the attacker's knowledge set), and allows continued safe execution of all components (i.e., of commands $\vec{C}'$) in the system, including newly forked ones (the last, iterated conjunct in the definition).

Fig. 16 shows the proof rules for our logic. Our assertion language is a first-order logic (for brevity not a separation logic) with the usual logical connectives and quantifiers, and access to local program variables. Pre- and postconditions can therefore refer to the local snapshot, but they *cannot* refer to the global trace. The latter corresponds to our methodology (recall Sec. 3.4), where pre- and postconditions also cannot directly express properties about the trace because access to it is governed by our library's ghost lock. Instead, properties about the global trace, such as the existence of a particular trace event, must always be expressed via the local snapshot. This ensures that pre- and postconditions are stable under potential environment interference, which is needed to prove our proof rules sound.

Similar to the discussion of the operational semantics, we discuss only non-standard proof rules. Proof rules corresponding to commands that modify the global trace, e.g., Send, enforce that the trace invariant is maintained under potential environment interference. For this purpose, we define an extensibility predicate specifying that appending a trace n event to an arbitrary extension of a trace τ maintains the trace invariant.

Definition 10. ***Extensibility.*** *A trace τ is* extensible *by a trace event n if the trace invariant ρ is maintained for any possible trace τ', given that τ is a prefix thereof:*

$$ext(n, \tau) \triangleq \forall \tau'.\, prefix(\tau, \tau') \wedge \rho(\tau') \implies \rho(\tau' + n)$$

Recall from Fig. 15 that commands modifying the global trace, e.g., send, also update the local snapshot to the most recent version of the trace. Analogous to the proof rule for assignments, the proof rules for trace-modifying command thus require that the syntactically substituted postcondition $\forall p.\, P[snap + p + n/snap]$ holds in the state before executing the command, where n is a trace event (e.g., $Send(e)$). The quantified p accounts for all possible trace

extensions that could have been made by the environment since the local snapshot was last updated, and thus accounts for arbitrary interleavings of participants and the attacker.

For the sake of presentation we have omitted additional assumptions that are available when discharing preconditions of snapshot-updating commands: e.g., in proof rule NonceGen we may additionally use nonce freshness, and in proof rule Recv we may assume that a received message was previously sent and not dropped in the meantime.

Theorem 1. ***Soundness of proof rules.***

$$\textit{If } \vdash [P]\ C\ [Q] \textit{ then } \models [P]\ C\ [Q]$$

We proof this theorem in the usual way, by structural induction on the shape of the proof tree given by the theorem's left-hand side of the implication. We proceed by a case distinction on the last rule applied, and may assume the theorem (i.e., our induction hypothesis) for this rule's premises. In our proof sketch we focus on a few interesting cases – send, sequential composition, and fork – and we present these cases further down, as individual lemmas. Send is interesting because it illustrates a trace-updating proof rule, for which we have to show that the trace invariant is maintained. The challenge for sequential composition is to show that our definition of configuration safety allows us to prove that *each* transition in the system maintains the trace invariant. The Fork proof rule is of interest because it is the only command that extends the system configuration with additional components.

We begin by sketching the proofs for several auxiliary lemmas about configuration safety that will be useful later on.

Lemma 3. *The empty command satisfies configuration safety given that the predicate Q holds.*

$$\forall n, i, \sigma, Q, \tau.\, Q(\sigma) \implies \mathit{safe}_n(i, \epsilon, \sigma, Q, \tau)$$

Proof sketch. We show for arbitrary n, i, σ, τ, and assuming $Q(\sigma)$, that $\mathit{safe}_n(i, \epsilon, \sigma, Q, \tau)$ holds. Case (i) from the definition of *safe* holds straightforwardly. Case (ii) is satisfied because there is no transition starting in command ϵ and resulting in a different command. Hence, this case vacously holds. □

Lemma 4. *A command C satisfying configuration safety for n execution steps is safe to execute for fewer execution steps.*

$$\forall m, n, i, C, \sigma, Q, \tau.\, m \le n \land \mathit{safe}_n(i, C, \sigma, Q, \tau) \implies \mathit{safe}_m(i, C, \sigma, Q, \tau)$$

Proof sketch. Straightforward induction on m. □

Next, we present soundness lemmas for the aforementioned interesting proof rules: send, sequential composition, and fork.

Send. Soundness for the proof rule Send directly follows from the following safety lemma:

Lemma 5.

$$\forall n, i, \sigma, Q, \tau.\, \mathit{ext}(\mathit{Send}(e(\sigma)), \mathit{snap}(\sigma)) \land (\forall p.\, Q[\mathit{snap} + p + \mathit{Send}(e)/\mathit{snap}](\sigma)) \implies \mathit{safe}_n(i, \mathit{send}(e), \sigma, Q, \tau)$$

Proof sketch. We prove this lemma by induction on n using the following induction hypothesis:

$$\mathit{IH}(n) \triangleq \forall i, \sigma, Q, \tau.\, \mathit{ext}(\mathit{Send}(e(\sigma)), \mathit{snap}(\sigma)) \land (\forall p.\, Q[\mathit{snap} + p + \mathit{Send}(e)/\mathit{snap}](\sigma)) \implies \mathit{safe}_n(i, \mathit{send}(e), \sigma, Q, \tau)$$

In the base case ($n = 0$), $\mathit{safe}_0(i, \mathit{send}(e), \sigma, Q, \tau)$ holds by definition. For the induction step, we assume $\mathit{IH}(n)$ and show that $\mathit{IH}(n+1)$ holds. I.e., we further assume $\mathit{ext}(\mathit{Send}(e(\sigma)), \mathit{snap}(\sigma))$ and $\forall p.\, Q[\mathit{snap} + p + \mathit{Send}(e)/\mathit{snap}](\sigma)$ for arbitrary i, σ, Q, and τ. We have to prove that $\mathit{safe}_{n+1}(i, \mathit{send}(e), \sigma, Q, \tau)$ holds. Case (i) from the definition of *safe* holds trivially because $\mathit{send}(e) \neq \epsilon$. To prove case (ii), we assume the implication's left-hand side and show that the right-hand side holds. In particular, we consider a transition that executes command $\mathit{send}(e)$. According to the operational semantics, only the transition rule Local with an application of the Send rule in its premise is applicable and modifies the command in the ith component's configuration. This allows us to conclude that the considered transition must have the following shape:

$$\langle \overrightarrow{\langle C, \sigma \rangle}, k_a, \tau \rangle \to \langle \overrightarrow{\langle C', \sigma' \rangle}, k_a', \tau' \rangle$$

where

$$|\vec{C'}| = |\vec{C}| \land k_a' = k_a \land \tau' = \tau + \mathit{Send}(e(\sigma)) \land$$
$$\vec{C_i'} = \epsilon \land \vec{\sigma_i'} = \vec{\sigma_i}[\mathit{snap} \mapsto \tau + \mathit{Send}(e(\sigma))] \land$$
$$(\forall j.\, i \neq j \implies \vec{C_j'} = \vec{C_j} \land \vec{\sigma_j'} = \vec{\sigma_j})$$

and $\rho(\tau) \land \mathit{prefix}(\mathit{snap}(\vec{\sigma_i}), \tau)$ holds. We have to prove that (1) $\rho(\tau')$, (2) $\mathit{prefix}(\tau, \tau')$, (3) $\mathit{prefix}(\mathit{snap}(\vec{\sigma_i'}), \tau')$, (4) $k_a \subseteq k_a'$, and (5) $\mathit{safe}_n(i, \vec{C_i'}, \vec{\sigma_i'}, Q, \tau')$ hold. Note that no additional local configurations have been added by this command because $|\vec{C'}| = |\vec{C}|$ holds. (1) follows directly by definition of Def. 10. (2) holds by choosing $p = \mathit{Send}(e(\sigma))$ as witness in Def. 7. (3) holds by reflexivity (cf. Lemma 1). (4) holds because the attacker knowledge is unchanged. Finally, (5) follows from Lemma 3 via the following derivation to obtain $Q(\vec{\sigma_i'})$:

$$\forall p.\, Q[\mathit{snap} + p + \mathit{Send}(e)/\mathit{snap}](\vec{\sigma_i}) \implies Q[\tau + \mathit{Send}(e)/\mathit{snap}](\vec{\sigma_i}) \iff Q(\vec{\sigma_i}[\mathit{snap} \mapsto \tau + \mathit{Send}(e(\vec{\sigma_i}))]) \iff Q(\vec{\sigma_i'})$$

where the implication is justified by the fact that $\mathit{prefix}(\mathit{snap}(\vec{\sigma_i}), \tau)$ holds. □

Seq. In the case where the last rule applied in our proof tree is Seq, we may assume the induction hypothesis for the rule's premises, i.e., $\models [P]\ S_1\ [R]$ and $\models [R]\ S_2\ [Q]$. Soundness for this case, i.e., showing $\models [P]\ S_1; S_2\ [Q]$, then follows from the following safety lemma:

Lemma 6.

$$\forall n, i, S_1, S_2, \sigma_1, R, Q, \tau.\, \mathit{safe}_n(i, S_1, \sigma_1, R, \tau) \land (\forall m, \sigma_2, \tau'.\, m \le n \land R(\sigma_2) \implies \mathit{safe}_m(i, S_2, \sigma_2, Q, \tau')) \implies \mathit{safe}_n(i, S_1; S_2, \sigma_1, Q, \tau)$$

```
1  func main(num_initiators, num_responders int) {
2    ... // initialization code
3    while (num_initiators > 0) {
4      fork (initiator_args) {
5        initiator(initiator_args)
6      }
7      num_initiators := num_initiators - 1
8    }
9    while (num_responders > 0) {
10     fork (responder_args) {
11       responder(responder_args)
12     }
13     num_responders := num_responders - 1
14   }
15   fork() { attacker() }
16 }
```

Figure 17: Sketch of a program C_{system} bootstrapping the distributed system by first executing sequential initialization code to, e.g., generate public/private keypairs and then forking several instances of an initiator and responder implementation and the highly non-deterministic attacker implementation. `initiator_args` and `responder_args` are abbreviations for a list of arguments that are passed to the initiator and responder implementations, respectively. E.g., the initiator's public/private keypair and the responder's public key might constitute `initiator_args`.

Proof sketch. We perform induction on n using the following induction hypothesis:

$$\begin{aligned} IH(n) \triangleq \forall i, S_1, S_2, \sigma_1, R, Q, \tau, . \\ & safe_n(i, S_1, \sigma_1, R, \tau) \wedge \\ & (\forall m, \sigma_2, \tau'. m \leq n \wedge R(\sigma_2) \implies safe_m(i, S_2, \sigma_2, Q, \tau')) \\ & \implies safe_n(i, S_1; S_2, \sigma_1, Q, \tau) \end{aligned}$$

The base case ($n = 0$) holds by definition. In the induction step, we may assume $IH(n)$ to prove $IH(n+1)$. For arbitrary i, S_1, S_2, σ_1, R, Q, and τ we assume the left-hand side, i.e., $safe_{n+1}(i, S_1, \sigma_1, R, \tau)$ and $\forall m, \sigma_2, \tau'. m \leq n+1 \wedge R(\sigma_2) \implies safe_m(i, S_2, \sigma_2, Q, \tau')$. It remains to prove that $safe_{n+1}(i, S_1; S_2, \sigma_1, Q, \tau)$ holds. The proof proceeds similarly to the proof of Lemma 5 except that in case (ii) the Local rule's premise is fulfilled by an application of either the Seq1 or Seq2 rule:

- Case Seq1: According to this transition's premise, there exists a transition $\langle S_1, \langle \sigma_i, \tau \rangle\rangle \to \langle \epsilon, \langle \sigma_i', \tau'' \rangle\rangle$ for some τ''. Thus, we obtain by definition of $safe_{n+1}(i, S_1, \sigma_i, R, \tau)$ that $\rho(\tau'')$, $prefix(\tau, \tau'')$, $prefix(snap(\sigma_i'), \tau'')$, $k_a \subseteq k_a'$, and $safe_n(i, \epsilon, \sigma_i', R, \tau'')$ hold. We distinguish two cases, namely $n = 0$ and $n > 0$. In the first case, we obtain by definition $safe_0(i, S_2, \sigma_i', Q, \tau'')$. In the second case, we obtain by definition of $safe_n(i, \epsilon, \sigma_i', R, \tau'')$ that $R(\sigma_i')$ holds. Therefore, we can instantiate m, σ_2, and τ' with n, σ_i', and τ'', respectively, in the quantifier above. Thus, we obtain $safe_n(i, S_2, \sigma_i', Q, \tau'')$. This concludes the proof for both cases $n = 0$ and $n > 0$ showing that $safe_{n+1}(i, S_1; S_2, \sigma_i, Q, \tau)$ holds.
- Case Seq2: This transition's premise specifies that a transition $\langle S_1, \langle \sigma_i, \tau \rangle\rangle \to \langle S_1', \langle \sigma_i', \tau'' \rangle\rangle$ for some τ'' exists. We apply the definition of $safe_{n+1}(i, S_1, \sigma_i, R, \tau)$ to obtain $\rho(\tau'')$, $prefix(\tau, \tau'')$, $prefix(snap(\sigma_i'), \tau'')$, $k_a \subseteq k_a'$, and $safe_n(i, S_1', \sigma_i', R, \tau'')$. By applying the induction hypothesis for n, we obtain $safe_n(i, S_1'; S_2, \sigma_i', Q, \tau'')$. Thus, we showed $safe_{n+1}(i, S_1; S_2, \sigma_i, Q, \tau)$.

□

Fork. Soundness of the Fork proof rule follows from the following safety lemma:

Lemma 7.

$$\begin{aligned} & \forall n, i, \vec{x}, S_1, S_2, \sigma_1, Q, \tau. \, safe_n(i, S_1, \sigma_1, Q, \tau) \wedge \\ & \qquad (\forall j, \sigma_2. \, [\sigma_1 \sim \sigma_2]^{\vec{x} \cup snap} \implies safe_n(j, S_2, \sigma_2, True(), \tau)) \\ & \implies safe_n(i, \textbf{fork } (\vec{x}) \ \{S_2\}; S_1, \sigma_1, Q, \tau) \end{aligned}$$

where $[\sigma_1 \sim \sigma_2]^{\vec{x} \cup snap}$ denotes that σ_1 maps the variables in $\vec{x}$ and variable snap to the same values as σ_2 does.

Proof sketch. We perform induction on n and use the following induction hypothesis:

$$\begin{aligned} IH(n) \triangleq \forall i, \vec{x}, S_1, S_2, \sigma_1, Q, \tau, . \\ & safe_n(i, S_1, \sigma_1, Q, \tau) \wedge \\ & (\forall j, \sigma_2. \, [\sigma_1 \sim \sigma_2]^{\vec{x} \cup snap} \implies safe_n(j, S_2, \sigma_2, True(), \tau)) \\ & \implies safe_n(i, \textbf{fork } (\vec{x}) \ \{S_2\}; S_1, \sigma_1, Q, \tau) \end{aligned}$$

For $n = 0$, $safe_0(i, \textbf{fork } (\vec{x}) \ \{S_2\}; S_1, \sigma_1, Q, \tau)$ holds by definition. In the induction step, we assume $IH(n)$ to show $IH(n+1)$. We assume the left-hand side, i.e.,

$$safe_{n+1}(i, S_1, \sigma_1, Q, \tau) \wedge \tag{1}$$

$$(\forall j, \sigma_2. \, [\sigma_1 \sim \sigma_2]^{\vec{x} \cup snap} \implies safe_{n+1}(j, S_2, \sigma_2, True(), \tau)) \tag{2}$$

and seek to show $safe_{n+1}(i, \textbf{fork } (\vec{x}) \ \{S_2\}; S_1, \sigma_1, Q, \tau)$. Similar to the proof of Lemma 5, the interesting case is (ii) in which we only consider the inference rule Fork. Based on the operational semantics, we obtain

$$\begin{aligned} & (\vec{C_i'} = S_1) \wedge (\sigma_1 = \vec{\sigma_i} = \vec{\sigma_i'}) \wedge (|\vec{C'}| = |\vec{C}| + 1) \wedge \\ & (\vec{C'}_{|\vec{C'}|} = S_2) \wedge \left[\sigma_1 \sim \vec{\sigma'}_{|\vec{C'}|}\right]^{\vec{x} \cup snap} \wedge \\ & (\forall j. \, 1 \leq j \leq |\vec{C}| \implies \vec{\sigma_j'} = \vec{\sigma_j}) \wedge \\ & (\forall j. \, 1 \leq j \leq |\vec{C}| \wedge i \neq j \implies \vec{C_j'} = \vec{C_j}) \end{aligned}$$

Since the attacker knowledge k_a and global trace τ remain unchanged by the application of this inference rule, we have to prove that (a) $safe_n(i, S_1, \sigma_1, Q, \tau)$ and (b) $safe_n(|\vec{C'}|, S_2, \vec{\sigma'}_{|\vec{C'}|}, True(), \tau)$ hold. (a) follows from applying Lemma 4 to $safe_{n+1}(i, S_1, \sigma_1, Q, \tau)$. Since (2)'s left-hand side is satisfied for $\sigma_2 = \vec{\sigma'}_{|\vec{C'}|}$, we obtain $safe_{n+1}(|\vec{C'}|, S_2, \vec{\sigma'}_{|\vec{C'}|}, True(), \tau)$ by instantiating the quantifier j with $|\vec{C'}|$. We also obtain (b) by applying Lemma 4. □

A.3 Trace Inclusion

We can now show the desired trace inclusion (recall Sec. A), which directly follows from Thm. 1.

Theorem 2. *If we boostrap the distributed system from a single component, with no precondition, a trace invariant that holds for the empty trace, and an initial attacker knowledge set, then the trace*

invariant always holds, regardless of how many transitions are performed, and additional components (participants and the attacker) are forked.

$$\forall C, \vec{C'}, Q, \sigma, \vec{\sigma'}, k'_a, \tau' . \vdash [\mathit{True}()]\ C\ [Q] \land \rho(\emptyset) \land \langle\langle C, \sigma\rangle, k_a^{init}, \emptyset\rangle \rightarrow^* \langle\overrightarrow{\langle C', \sigma'\rangle}, k'_a, \tau'\rangle \implies \rho(\tau')$$

where k_a^{init} is the initial attacker knowledge consisting of all public terms.

Proof sketch. We prove this theorem by first applying soundness of our proof rules (Thm. 1) and expanding Def. 8 because all states σ satisfy $\mathit{True}()$. We proceed by induction over the length of transition sequences. Since the trace invariant holds initially, is maintained by each transition, and each command in every component of the system satisfies configuration safety, we obtain $\rho(\tau')$ for every trace τ' that is possible after executing n transitions, where n is the induction variable. □

Thm. 2 implies the following trace inclusion property where ϕ is a security property implied by the trace invariant ρ, i.e., $\rho \models \phi$:

$$\forall C, Q . \vdash [\mathit{True}()]\ C\ [Q] \land \rho(\emptyset) \implies \mathit{Tr}(C) \subseteq \mathit{Tr}(\rho) \subseteq \mathit{Tr}(\phi)$$

where $\mathit{Tr}(C)$ denotes the set of all traces that result from executing arbitrary many transitions according to the small-step operational semantics. $\mathit{Tr}(\rho)$ and $\mathit{Tr}(\phi)$ are the sets of traces satisfying ρ and ϕ, respectively. I.e., $\mathit{Tr}(\rho) = \{\tau \mid \forall \tau . \rho(\tau)\}$ and $\mathit{Tr}(\phi)$ analogously.

In our verification case studies we prove Thm. 2 in three steps: in step 1, we once-and-forall verify our reusable verification library, including a most-general attacker implementation (an iterated non-deterministic choice between all executable commands) against a partially abstract (thus sufficiently general) trace invariant. I.e., the judgement $\vdash [true]\ C_a\ [true]$ we obtain for the attacker holds for all possible attackers and protocol-specific instantiations of this abstract trace invariant. In step 2, we implement each participant in its own program C_i; verifying these effectively yield a judgement $\vdash [P_i]\ C_i\ [Q_i]$ per participant.

In step 3, we combine these separate judgements for the protocol participants and the attacker by constructing a program C_{system} that first performs some sequential initialization code and then forks several instances of protocol participants and the attacker, as illustrated in Fig. 17. By taking the number of participant instances as unconstraint input parameters, we obtain a result for unboundedly-many instances. Functions and non-deterministic choices are straightforward extensions to our programming language. The initialization code's purpose is to establish the participants' preconditions. E.g., in our NSL case study we implement initialization code that generates public-private keypairs and passes the relevant keys to the individual protocol participants. In the case of WireGuard, the corresponding initialization code remains an assumption, which is typical for security protocol verification and corresponds to assuming that there exists a mechanism to authentically distribute public keys.