A Generic Methodology for the Modular Verification of Security Protocol Implementations

Linard Arquint* , Malte Schwerhoff* , Vaibhav Mehta† , and Peter Müller*
*Department of Computer Science, ETH Zurich, Switzerland
†Princeton University, Princeton NJ 08544, USA
{linard.arquint, malte.schwerhoff, peter.mueller}@inf.ethz.ch
vaibhavm@princeton.edu

***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 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.**

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.

Several approaches for automating protocol verification have been proposed in the past. The vast majority of these targets protocol *models*, i.e., abstract descriptions of the cryptographic operations and message exchanges that constitute a protocol. Verification of protocol models is useful to show security of the protocol *design*, but does not guarantee that concrete protocol *implementations* are also secure. Common programming errors (such as missing bounds checks in the Heartbleed bug [5]) or incorrect implementations of the design (such as accidentally omitted protocol steps in the Matrix SDK [6]) may render the implementation insecure even if the protocol design is secure. Additionally, formal models may not always exist, or may not be in sync with evolving implementations.

One approach at closing this gap is *code generation* (e.g., [7]–[10]), where an executable implementation is automatically generated from a verified model. With this approach, verification is performed on the abstract protocol level, without the need to involve potentially intricate programming language semantics. A typical limitation of this approach, however, is that the generated code often exhibits sub-optimal performance, cannot interact easily with existing code, and cannot be changed manually afterwards (e.g., to optimize performance or upgrade dependencies) without risking to compromise security. The opposite direction is taken by *model extraction* (e.g., [11]–[15]), where security properties are verified against a protocol model that is extracted from code. To enable automatic model extraction, implementations are often severely restricted by coding disciplines. The same restriction applies to approaches with *executable models* (e.g., [16], [17]), 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.

This work. In this paper, 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., [18]–[22]), which makes it readily applicable. It is based on separation logic [23], [24], 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,

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

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

one can verify each method (or protocol participant) in isolation, without considering the implementations of other methods. Modularity is crucial for scalability, to reduce re-verification effort when the code evolves, and to provide strong guarantees for libraries whose clients are not known.

As is common in protocol verification, we explicitly model the global traces of a protocol, which allows us to express security properties in ways familiar to security experts. These traces are expressed and manipulated via *ghost code* [25], 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, i.e., that each sent message is correctly reflected in 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. Technically, we treat each participant of the protocol (including a Dolev-Yao attacker [26]) as a thread of a concurrent system, and the global trace as shared state among these threads. This approach allows us to leverage existing verification techniques for shared-data concurrency.

Our use of a global protocol trace is inspired by Bhargavan et al.'s DY* framework [9], [10], [27]. However, they perform code generation (from a functional implementation in F* [28] to OCaml and C), whereas we target implementations directly. For soundness, their approach requires a specific coding discipline (one method per protocol step) that needs to be enforced manually; our methodology requires no such restrictions. We also took inspiration from Dupressoir et al. [29], who model the protocol trace as a shared concurrent data structure. However, their verification technique depends heavily on the intricacies of the used language (e.g., C's volatile fields) and verifier (VCC [30]). In contrast, we present a generic methodology that applies to a wide range of languages and verification tools. Moreover, neither of the two approaches can prove injective agreement, which our approach enables.

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 notion of linear resources to prove injective agreement, i.e., the absence of replay attacks, which is difficult in other logics.
3) We developed a reusable Go library that facilitates the ghost code instrumentation to maintain 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 [19] to verify memory safety and security of Go implementations of Needham-Schroeder-Lowe (NSL) [3], [4] and WireGuard [1]. A prototype of the reusable library for C and the VeriFast verifier [18] shows that our approach relies on only a few common program verifier features.

The Go and C implementations of our reusable verification library and the case studies are available in our open-source artifact [31].

Outline. Sec. 2 introduces background on trace-based verification and our attacker model. In Sec. 3, we explain how we encode the global trace of the protocol 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, NSL and WireGuard. We explain the trust assumptions underlying our methodology 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 and 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 function applications over symbolic values, so-called *terms*, and encode the perfect cryptography assumption, e.g., that decryption succeeds if and only if the correct decryption key is used.

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 any message on the network. Furthermore, an attacker can 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 all its long-term state, such as long-term secret keys. Corrupting a participant session adds session-specific short-term state, e.g., ephemeral secret keys, or exchanged nonces.

Trace entries. The global trace is a sequence of events and each event corresponds to a high-level operation performed by a participant or the attacker. Each event has a name and takes event-specific arguments. E.g., event $\mathit{CreateNonce}(n)$

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 security properties. E.g., a protocol-specific event may express which nonces or keys a participant uses to communicate with a particular peer (cf. Sec. 4).

We use seven protocol-independent events, two of which correspond to operations that are integral to virtually every protocol: (1) a *create nonce* event records that a fresh nonce has been generated, and (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, for instance, 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 is satisfied 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, with symbolic terms as arguments. In order to verify concrete protocol implementations, we must relate this abstract view of the protocol to the concrete representation of the system state in the implementation. In particular, this state is distributed over the protocol participants, such that each participant has only a partial knowledge of the overall system state. In this section, we show how to reason locally about the implementations of the participants and how to relate their local operations to the global trace.

3.1. Safety Verification

To support realistic protocol implementations, our verification technique needs to handle programming concepts such as mutable heap data structures, asynchronous and concurrent execution as well as closures, some of which are necessary for efficient implementations, but difficult to reason about. To this end, we employ separation logic [23], [24], the de-facto standard for the modular verification of imperative code. Separation logic is supported by existing verifiers for many languages, including VeriFast [18] for C, Prusti [22] for Rust, and Gobra [19] for Go. All of them can be used in combination with our methodology.

We use separation logic to verify safety of each participant, that is, memory safety (e.g., the absence of buffer overflows), crash freedom, and the absence of standard concurrency errors (e.g., 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 separation-logic-based verification and orthogonal to the focus of this paper. However, as we will see in Sec. 4.1, separation logic also provides a simple way to prove injective agreement.

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, for instance, a message) to the byte array ra (representing, for instance, a number) is specified by relating the corresponding terms: $\exists x, r. xa = \gamma(x) \land ra = \gamma(r) \land r = h(x)$, 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. Enforcing a unique byte-level representation for every term (for instance, by preceding it with a tag) is not possible when targeting existing implementations with fixed message formats.

To solve this problem, we adopt the *pattern requirement* from Arquint et al. [32], 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. With the pattern requirement, 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

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

Figure 1. 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.

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. 1. 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 as well as the attacker, which we ensure as follows.

We model each participant and the attacker as a thread in a concurrent system and the global trace as a data structure that is shared among, and manipulated by, these threads. The concurrency reflects that participants have no control over the scheduling of other participants and that the attacker may perform its operations at any time. 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. The trace data structure provides two operations: appending an event, and reading the current state of the global trace. Fig. 2 illustrates how participants and the attacker interact with the global trace.

Regarding the global trace as a shared data structure in a concurrent execution allows us to employ any of the established verification techniques for concurrency reasoning. For concreteness, we use a *ghost lock* to reason about the shared trace. Reasoning about ghost locks is completely analogous to standard locks: the lock is associated with a lock invariant, which needs to be established when the lock is first created. This invariant may be assumed whenever the lock is acquired and must be proved to hold upon release. 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 additional thread interleavings. Since reasoning about ghost locks is analogous to standard locks, they are supported by separation logic program verifiers.

We use a single ghost lock to protect the global trace. Its lock invariant is the trace invariant (see Fig. 2). By

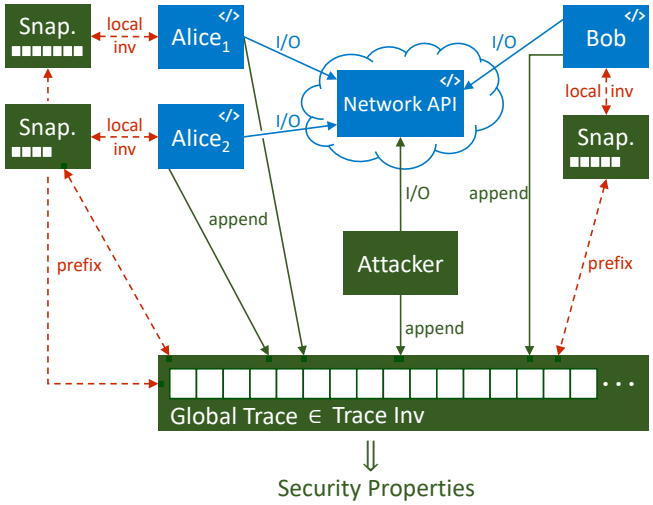

Figure 2. An overview of the main components of a protocol execution in our methodology. The blue boxes are components of the protocol implementation; green boxes denote ghost structures that are used for verification. Blue and green arrows denote actual and ghost method calls, respectively. The red 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 prove (global) security properties for a protocol by utilizing a trace invariant abstracting over all possible traces. We modularly verify for each protocol implementation (e.g., two and one implementations of Alice's and Bob's role, respectively) and the attacker that the trace invariant is maintained. We enable 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.

formulating this invariant in separation logic, it can express ownership of resources, which allows us to prove security properties that is out of scope 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 computing a hash, 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, we provide a library wrapper (see Sec. 5 for details) that performs these four steps[2]. 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 consider code that (1) acquires the ghost lock, (2) determines which operations the attacker could potentially perform based on its current

2. 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 3. 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.

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 by type-based approaches).

3.4. Local Snapshots

In order 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. 1), 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* of the global trace, that is, a local copy of the trace (see Fig. 2). 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.

The local snapshot of a participant is a ghost data structure that is owned by this participant. Therefore, in contrast to the shared global trace, the local snapshot is not directly affected by operations of other participants and the attacker. 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.

By making snapshots a local data structure of each participant, we can use standard sequential reasoning to relate the content of the 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. 3 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 and the attacker between line 1 and 2.

In summary, we have shown how we use separation logic to verify safety properties and prove that all participants and the attacker maintain a specified invariant over the global trace of events. All of theses proofs are modular and can be automated using existing separation-logic verifiers.

4. Proving Security Properties

Once we have verified that all participants and the attacker maintain the trace invariant, it remains to prove that this invariant implies the intended security properties. In this section, we illustrate this for two important 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 applicable to more complex properties such as forward secrecy, as demonstrated in Sec. 6.2.

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. 1. 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 turns 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) from Alice's and Bob's perspective, respectively. We focus on Alice's perspective in the following. We prove authentication for the Bob's perspective Sec. 6.2, where we also discuss authentication properties for WireGuard.

```
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 4. 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`, as well as 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` in 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 5. 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 $*$.

Non-injective Agreement. The fact that Alice agrees with Bob on the nonces na and nb, known as *non-injective agreement* [33], is shown in Fig. 4 (ignore the conjunct highlighted in blue for now). This trace-based property states that the *FinishA* event occurs on the trace (line 2) and that either *Respond* 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. 5 (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 (since it is part of the trace invariant). Since her local snapshot gets updated upon the receive-operation, the received message is also recorded on the local snapshot, such that Alice can retain the message invariant as part of her stable knowledge. Consequently, when Alice is about to add 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 only accessible to Alice and Bob (unless corrupted).

Once we established the trace invariant, it remains to show that. for all traces, the invariant from Fig. 5 implies non-injective agreement (Fig. 4). 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 trace-based verification technique for injective agreement in protocol implementations (see also Sec. 8).

The conjunct highlighted in blue in Fig. 4 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 need to strengthen our trace invariant to imply the highlighted property in Fig. 4. 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 requires 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. Consequently, the absence of an event on the local snapshot does *not* imply 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 linear resources (non-duplicable facts) to enforce event uniqueness. In separation logic, like in linear logic, conjoining a linear resource R with itself is equivalent to false. So if an assertion (such as an invariant) includes two resources R_1 and R_2, we can conclude that R_1 and R_2 must be different. We use this mechanisms as follows.

Conceptually, we tie event uniqueness to nonces because nonces are, by assumption of perfect cryptography, unique. Based on this idea, we enforce event uniqueness through the following machinery, which is implemented in our verification library (technical details follow in Sec. 5): 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, users state (via annotations) for which events they want to use the nonce (e.g., *FinishA*).

```
1  !t.AttackerKnows(s) || t.IsCorrupted({A, B})
```

Figure 6. Secrecy of a term s states that the attacker does not know s except after corrupting a participant that is allowed to read s (here A or B). We prove this property for all traces t satisfying the trace invariant.

The library call returns not only the fresh nonce (na), but also a linear resource per event type and nonce.

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`. When she appends the event $FinishA(\_, \_, na, \_)$ to the trace, the witness is transferred to the trace invariant, as expressed by the highlighted conjunct in Fig. 5. Due to the linearity of the resource, any attempt to append another $FinishA$ event with na would fail to verify because the trace invariant (which would then be equivalent to false) would not be preserved.

Consequently, the invariant from Fig. 5 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. 4).

In summary, 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 express 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. As shown in Fig. 6, 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 over the concrete attacker knowledge, we follow Bhargavan et al. [27], [34] 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 protocol session. In this case, we add the class of terms readable by the corrupted participant or protocol 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.

As a consequence of proactively enforcing secrecy labels, the secrecy labels of all terms in the (concrete) attacker knowledge are either public or contain a participant or protocol session that has been corrupted in the past. Hence, a term s with a non-public secrecy label is either not in the attacker knowledge or the attacker has corrupted a participant or protocol session that is allowed to read s. We prove this lemma 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, in particular attacker completeness (Sec. 2), that are proved only once and hold for all protocols. To enable verification of a wide range of protocols, the global trace is parametric in the events it can record, 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 [19], and one for the C verifier VeriFast [18]. 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 the technical solutions we used to implement it.

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 specified 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

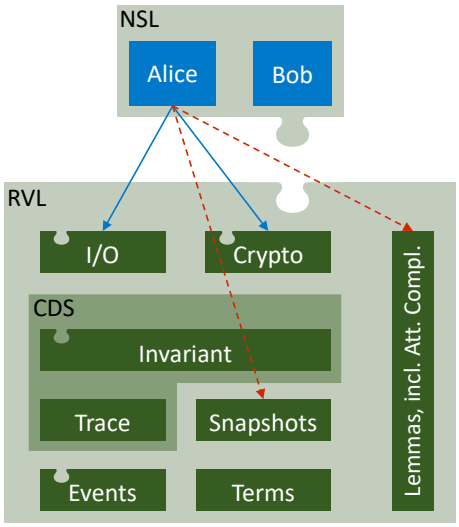

Figure 7. Structure of our reusable verification library (RVL). The library provides implementations for the abstractions used in our methodology: terms, events, the global trace, and local snapshots. Both the trace and all local snapshots are governed by the trace invariant. The trace is encapsulated 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. These include attacker completeness and also lemmas that are useful to verify participant implementations. 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.

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.

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 from the I/O and crypto APIs. These calls can be inlined in the participant implementation, such that the entire library can be removed from the executable program. Consequently, it does not cause any runtime overhead.

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

```
1  pred TraceInv[P](t: Trace) {
2    foreach e: Entry of t:
3      let pre = ... in // trace prefix up to e
4      match e {
5        case Send(msg):
6          MsgInv[P](msg, pre)
7        case PEvent(pe):
8          P::PEventInv(pe, pre)
9        ...
10     }
11 }
```

Figure 8. Excerpt of the parametric trace invariant (in separation logic), 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 interface instance `P`.

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 include 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 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 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

Library	LOC	LOS	Verification time [s]
Go/Gobra	210	6,601	123.0
C/VeriFast	326	2,340	0.9

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

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 implementations of our library. As explained above, the library consists mostly of ghost code; only around 3% 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 verifier 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 (Sec. 2 to 3), secrecy labels (Sec. 4), and terms with *equational theories*. Conceptually, events form an algebraic data type (ADT), as does the global trace (it is a functional Nil/Cons list). Labels and terms are also algebraic structures, but with additional properties (e.g., labels have a commutative join operator).

In the Gobra implementation of the library, we represent 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 emit axioms that ADT constructors are injective in their arguments, and that different constructors produce different events. For terms, on the other hand, we emit additional axioms to encode cryptographic equational theories, e.g., $g^{xy} = g^{yx}$, 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 (called "auto-lemmas") to express equational theories for terms.

Linear Resources. Our novel support for proving injective agreement (cf. Sec. 4) requires reasoning about the uniqueness of certain protocol-specific events. We enable this reasoning by building on separation logic's notion of *linear resources*. Some separation logic verifiers allow programs to declare custom resources; otherwise, one can introduce a (ghost) memory location and use this location as linear resource, since all separation logic verifiers support (exclusive) ownership of heap locations.

In Gobra, ownership of a single heap location is expressed as `acc(p)` (for a pointer `p`), in VeriFast as `p |-> _`. Since ownership is exclusive, we get `acc(p) && acc(q) ==> p != q` in Gobra (and analogously in VeriFast). Building on this, we can use separation logic's predicates [35] to construct linear resources with arbitrary parameters by mapping the parameter tuples injectively to a pointer, e.g., `acc(map(a1, a2))`, where `map` is an injective function. We use such resources 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 locks. 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 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 trivially satisfied in our library, where ghost locks are used only around ghost code.

Conceptually, snapshots are owned locally by each participant. To allow the trace invariant to relate the global trace and the snapshots, we technically share ownership of each snapshot between the participant and the shared concurrent data structure using separation logic's fractional ownership [36]. Fractional ownership is supported by both Gobra and VeriFast.

Case Studies	LOC	LOS	Verification time [s]
NSL	197	908	91.7
WireGuard	550	5,346	238.3

Figure 10. Lines of code (LOC) and lines of specification (LOS) (incl. ghost code) for the NSL and WireGuard case studies, together with the average verification time in Gobra. We have performed the measurements in the same way as in Fig. 9.

6. Case Studies

We apply our methodology to Go implementations of the NSL and the WireGuard VPN protocols that achieve strong security properties. This demonstrates that our methodology scales to real-world and interoperable protocol implementations. Our case studies are part of our artifact [31].

6.1. NSL

We have implemented the initiator and responder roles for the NSL protocol (cf. Fig. 1) in Go for the Gobra verifier. We implemented the core of the protocol as one method per participant; we also verified an alternative 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 in a struct and use an invariant to relate the struct fields via the term abstraction to their local snapshot and, thereby, to the global trace.

Fig. 11 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. 5). 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.

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 participants as Go routines (similar to threads) to demonstrate that key distribution (although not part of the protocol) can be modeled using our methodology. Fig. 10 reports the size of the implementation and specification, as well as the verification time in Gobra. The specification and verification time exclude our reusable verification library and the alternative initiator implementation, but includes 362 LOS to instantiate the library.

```
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 11. The struct used for Alice's local state in the NSL implementation, 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.

6.2. 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 has received a lot of attention lately. It has not only been integrated into the Linux kernel. The protocol's core, the WireGuard protocol, has been well studied [37], [38]. The WireGuard protocol performs an authenticated key exchange. 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 have taken the existing Go implementation from Arquint et al. [39], which is based on WireGuard's official Go implementation [40]. Advanced VPN features such as DDoS protection are not part of the implementation. Nevertheless, the implementation is interoperable with other WireGuard implementations and supports tunneling IP packets via the established VPN connection to and from the

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

Figure 12. Strong (unhighlighted-only parts) 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.

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. Thanks to our reusable verification library's parametric design, instantiating the library with the existing networking library was straightforward. We have annotated the cryptographic functions from the original implementation with suitable postconditions to express our cryptographic assumptions.

The implementations of the initiator and responder roles are located in different Go packages. We have factored out role-independent specifications into packages that are imported by both roles. E.g., one such package defines the WireGuard-specific events and trace invariant. We report on statistics after presenting the security properties that we prove for the implementations.

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 [10], [41], [42], 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 tree properties hold: (1) The attacker does not know s (line 1), (2) the actor or its peer has been corrupted before completing the handshake at timepoint j (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 session state of the corrupted participant. This session state contains 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. Session keys of sessions established in the past, i.e., before time point j, 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. [43], generalizing the more widely known notion of key compromise impersonation (KCI).

In WireGuard, we prove strong forward secrecy for the two session keys from the initiator's perspective after completing the handshake and after receiving each transport message. In contrast, the responder achieves only weak instead of strong forward secrecy at the end of the handshake. We prove strong forward secrecy for the session keys from the responder's perspective after receiving the first transport message and after sending each transport message. The forward secrecy property 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 only in a way that they can be read 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 (session 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 respond do not agree on the nonces.

In WireGuard, we prove injective agreement on both

```
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 conjunct with a blue background indicates the `Commit` event's uniqueness requirement for the given `m`.

session keys with AKC security from the initiator's perspective at the end of the handshake. From the responder's perspective, we prove the same agreement property after receiving the first transport message, i.e., confirming the session keys.

Discussion. Finding a strong enough trace invariant to prove the presented security properties was challenging. 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 (or if suitable an over-approximation thereof).

Moreover, the invariant for protocol-specific events has to be strong enough to prove injective agreement with KCI resistance. Our reusable verification library facilitates strengthening the proven authentication property from non-injective to injective agreement (with or without KCI resistance) because it suffices to express the uniqueness witness for each protocol-dependent event. Therefore, we were able to focus on finding a suitable invariant to prove non-injective agreement with KCI resistance. Afterwards, we expressed each event's uniqueness witness to strengthen the authentication property, which required less than 40 additional LOS.

Verifying the initiator and responder implementations takes ca. 4 minutes (cf. Fig. 10), which includes proving WireGuard-specific lemmas. Overall, the verified WireGuard codebase (excluding the reusable verification library) consists of 550 LOC and 5,346 LOS, whereof 984 LOS instantiate the reusable verification library with protocol-specific events and a sufficiently strong trace invariant.

7. Trust Assumptions

Our methodology allows us to prove strong security properties for implementations of security protocols. Like with all verification techniques, these proof 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 [44].

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. Note that omittance of all other events 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 to the attacker in case of corruption).

8. Related Work

Much prior work on the verification of cryptographic protocols exists, and surveys such as [45]–[47] 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. [29] use VCC [30] to verify memory safety, non-injective agreement, and (via an external argument in Coq) weak secrecy, of two protocols implemented in C: RPC [34] and Otway-Rees [48]. 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); and they account for principal corruption, but not session corruption (we account for both). Polikarpova et al. [49] 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, more goal-directed 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 but stateful subset of TPM 2.0.

Vanspauwen et al. [50], like us, use a separation-logic-based verifier (VeriFast [18]), but they do not model a global trace (which we do). 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) that uses PolarSSL's cryptographic primitives. Unlike us, they do not consider session corruption.

Arquint et al. [32] suggest a two-step approach: First, a protocol *model* is verified via Tamarin [51]. If successful, a separation logic predicate (one per participant) with I/O specifications [52] is generated, specifying which interactions (sending and receiving messages) 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 from a sequential Tamarin model) may prevent verifying corresponding implementations.

Bhargavan et al. [27] suggest DY*: a framework for verifying protocols implemented in F* [28], 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 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 such as 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 [9]. 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 KCI resistance. 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., [34], [53], [54]) uses the F7 type checker [53] to verify security of functional programs written in F#, but does not integrate equational theories, and has limited support for reasoning about mutable state. Moreover, global protocol traces are not modelled, and security properties therefore only implicitly stated.

Küsters et al. [55] 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 [56] and ASPIER [57] (for C), and JavaSec [58] (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 as in the case of the Heartbleed bug [5]).

Prior work on verifying properties of WireGuard (our main case study) includes [10], [37], [38], [41], [59], [60], but 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. However, the necessary proofs are significantly harder to automate, e.g., due to probabilistic reasoning, and we are not aware of tools for modularly verifying computational security properties of realistic implementations. Recently, first separation logics for probabilistic reasoning have been proposed (e.g., [61]–[63]), but we are not aware of automated verifiers for such logics.

9. Conclusions

We presented a methodology for the 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. Moreover, separation logic allows us to specify resources in the trace

invariant to express uniqueness of protocol-specific events. This is key to modularly proving injective agreement; to the best of our knowledge, our work is the first verification technique for protocol implementations to achieve that.

We have instantiated our methodology for two languages and separation logic verifiers. Our case studies on NSL and WireGuard demonstrate that our methodology handles existing and interoperable implementations of protocols achieving strong security properties, such as forward secrecy and injective agreement with AKC security.

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.

Acknowledgements. 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.

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. Lecture Notes in Computer Science, 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 (1)*. 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, "An in-depth symbolic security analysis of the ACME standard," in *CCS*. ACM, 2021, pp. 2601–2617.

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

[11] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 1, pp. 5:1–5:61, 2008.

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

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

[14] 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.

[15] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 483–502.

[16] 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.

[17] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in WebAssembly," in *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 1256–1274.

[18] 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. Lecture Notes in Computer Science, vol. 6617. Springer, 2011, pp. 41–55.

[19] 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 (1)*, ser. Lecture Notes in Computer Science, vol. 12759. Springer, 2021, pp. 367–379.

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

[21] 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.

[22] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The Prusti project: Formal verification for Rust," in *NFM*, ser. Lecture Notes in Computer Science, vol. 13260. Springer, 2022, pp. 88–108.

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

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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. Lecture Notes in Computer Science, vol. 5674. Springer, 2009, pp. 23–42.

[31] ModularVerification. A generic methodology for the modular verification of security protocol implementations. [accessed 02-December-2022]. [Online]. Available: https://github.com/ModularVerification/SecurityProtocolImplementations

[32] L. Arquint, F. A. Wolf, J. Lallemand, C. Sprenger, R. Sasse, S. Wiesner, D. Basin, and P. Müller, "Sound verification of security protocols: From design to interoperable implementations," in *S&P*. IEEE, 2023 (to appear).

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

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

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

[36] J. Boyland, "Checking interference with fractional permissions," in *SAS*, ser. Lecture Notes in Computer Science, vol. 2694. Springer, 2003, pp. 55–72.

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

[38] 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.

[39] L. Arquint, F. A. Wolf, J. Lallemand, C. Sprenger, R. Sasse, S. Wiesner, D. Basin, and P. Müller. Tamarin model & verified Go implementation of the WireGuard VPN key exchange protocol. [accessed 21-November-2022]. [Online]. Available: https://github.com/SoundVerification/wireguard

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

[41] 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.

[42] 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.

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

[44] 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 *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 983–1002.

[45] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," in *IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 777–795.

[46] 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.

[47] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *POST*, ser. Lecture Notes in Computer Science, vol. 7215. Springer, 2012, pp. 3–29.

[48] M. Abadi and R. M. Needham, "Prudent engineering practice for cryptographic protocols," *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 6–15, 1996.

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

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

[51] 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.

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

[53] 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.

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

[55] 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.

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

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

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

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

[60] 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.

[61] 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.

[62] 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.

[63] 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.