A Cryptographic Investigation of Secure Scuttlebutt

Oleksandra Lapiha
École Normale Supérieure, Symbolic Software

September 2019

Abstract

Scuttlebutt is a framework for developing decentralized applications which can achieve scalability, availability and secure communications despite not requiring a central trusted party. The Scuttlebutt protocol describes three sub-protocols for its handshake, invite system and private messaging, all of which claim to achieve ambitious security goals.

We present a comprehensive analysis of the Scuttlebutt sub-protocols in the symbolic model using the automated verifier ProVerif, and the first computational proofs of Scuttlebutt's security using the CryptoVerif proof assistant.

1 Introduction

When presenting itself as a protocol and not as an application, Scuttlebutt describes itself as *"a protocol for building decentralized applications that work well offline and that no one person can control. Because there is no central server, Scuttlebutt clients connect to their peers to exchange information. This guide describes the protocols used to communicate within the Scuttlebutt network."* [1, 2]

To a more general audience, Scuttlebutt is a *"decentralized secure gossip platform"*. It offers a framework for building social network[1], media sharing applications[2] and even correspondence chess platforms[3] that are scalable, decentralized and that achieve certain security guarantees on their communication, not limited only to confidentiality and authentication, but extending to forward secrecy and identity hiding (per the definitions in §2.3)

Scuttlebutt's security goals are spread across three Scuttlebutt sub-protocols, each with a specific purpose inside a complete Scuttlebutt application.

1. **Scuttlebutt's handshake protocol** achieves an authenticated key exchange (AKE) between two parties, Alice and Bob. This key is later used to encrypt a stream of content payloads (referred to as a *"feed"*). Scuttlebutt's handshake
2. **Scuttlebutt's invite protocol** allows members of a Scuttlebutt network or application (referred to as a *"pub"*) to delegate and manage access to said *"pub"*, by generating and communicating secret invitation tokens that can be consumed by the desired new participant.

[1] `https://github.com/ssbc/patchwork`
[2] `https://github.com/ssbc/patchbay`
[3] `https://github.com/Happy0/ssb-chess`

3. **Scuttlebutt's private messaging protocol** allows for the easy instantiating of secure, end-to-end encrypted group chats within Scuttlebutt applications, accommodating up to seven participants.

Not only does Scuttlebutt offer a suite of complex protocols, but it also intends for them to be used across an unbounded number of unspecified use cases, which is unusual for a protocol suite and which is certainly not the case for TLS, Signal or other popular protocols. Despite this, the only existing prior formal analysis of Scuttlebutt appears to pertain to its handshake. Cas Cremers and Dennis Jackson's modeling of Scuttlebutt's handshake [3] using the Tamarin automated symbolic model verifier [4] was furthermore not specifically intended to be a comprehensive analysis of Scuttlebutt, but rather only employs Scuttlebutt as an example to show how Tamarin can catch protocol weaknesses based on small subgroup attacks [5] that affect certain Elliptic Curve Diffie-Hellman primitives [6].

In this work, we present the first *comprehensive* analysis of all of Scuttlebutt's major sub-protocols in the symbolic models. We employ the ProVerif [7, 8] symbolic prover, and show that it can also detect the same small subgroup attacks that Tamarin was able to identify in Scuttlebutt's handshake. Furthermore, we employ the CryptoVerif [9] proof assistant and produce the first game-based computational proofs on certain elements of Scuttlebutt's handshake and invite sub-protocols.

Contributions. In this work, we provide the following contributions:

- Formal description of Scuttlebutt's sub-protocols. (§3)
- Comprehensive modeling of Scuttlebutt's three sub-protocols (handshake, invite and private messaging) in the symbolic model using ProVerif, and analysis against each sub-protocol's claimed security goals. (§4)
- Modeling of Scuttlebutt's handshake and invite redemption protocols in CryptoVerif with game-based proof for certain authentication properties. (§5)

All models and supporting materials for this work may be found at the following source code repository:

`https://github.com/olapiha/scuttlebutt`

2 Preliminaries

In this section, we cover the preliminaries to our analysis of the Scuttlebutt protocol, going through an introductory comparison of formal verification in the symbolic versus computational model. We then cover our notation for cryptographic primitives, as well as the concrete security properties that we assume for these primitives.

2.1 Formal Verification: A Primer

There are two different styles in which protocols have classically been modeled and in this work we employ both of them. *Symbolic* models were developed by the security protocol verification community for ease of automated analysis. Cryptographers, on the other hand, prefer to use *computational* models and do their proofs by hand. Here we briefly outline their differences [10] in terms of the two tools we will use.

ProVerif analyzes symbolic protocol models, whereas CryptoVerif verifies computational models. The input languages of both tools are similar. For each protocol role (e.g. client or server) we write a *process* that can send and receive messages over public channels, trigger security events and store messages in persistent databases.

In ProVerif, messages are modeled as abstract terms. Processes can generate new nonces and keys, which are treated as atomic opaque terms that are fresh and unguessable. Functions map terms to terms. For example, encryption constructs a complex term from its arguments (key and plaintext) that can only be deconstructed by decryption (with the same key). The attacker is an arbitrary ProVerif process running in parallel with the protocol, which can read and write messages on public channels and can manipulate them symbolically.

In CryptoVerif, messages are concrete bitstrings. Freshly generated nonces and keys are randomly sampled bitstrings that the attacker can guess with some probability (depending on their length). Encryption and decryption are functions on bitstrings to which we may associate standard cryptographic assumptions such as IND-CCA. The attacker is a probabilistic polynomial-time CryptoVerif process running in parallel.

Authentication goals in both ProVerif and CryptoVerif are written as correspondences between events: for example, if the client triggers a certain event, then the server must have triggered a matching event in the past. Secrecy is treated differently in the two tools; in ProVerif, we typically ask whether the attacker can compute a secret, whereas in CryptoVerif, we ask whether it can distinguish a secret from a random bitstring.

The analysis techniques employed by the two tools are quite different. ProVerif searches for a protocol trace that violates the security goal, whereas CryptoVerif tries to construct a cryptographic proof that the protocol is equivalent (with high probability) to a trivially secure protocol. ProVerif is a push-button tool that may return that the security goal is true in the symbolic model, or that the goal is false with a counterexample, or that it is unable to conclude, or may fail to terminate. CryptoVerif is semi-automated, it can search for proofs but requires human guidance for non-trivial protocols.

We use both ProVerif and CryptoVerif for their complementary strengths. CryptoVerif can prove stronger security properties of the protocol under precise cryptographic assumptions, but the proofs require more work. ProVerif can quickly analyze large protocols to automatically find attacks, but a positive result does not immediately provide a cryptographic proof of security. This complimentary usage of both symbolic and computational analysis has proven successful in recent analysis of TLS 1.3 [11] and Signal [12], among other use cases.

2.2 Cryptographic Primitives

In this work, we use the following notation for cryptographic primitives:

- H stands for a collision resistant hash function, which produces a hash for an input message.
- ENC is a symmetric encryption function that takes as input a message and a key, and outputs a ciphertext.
- SIGN is a digital signature function that takes as input a message and a secret key, and produces a signature for this message.
- SIGNVERIF takes as input the signature, the public key of a signer and a corresponding message, and outputs a boolean value `TRUE` if and only if the signature is correct.
- AEADENC is a symmetric authentication encryption with associated data (AEAD) function. It takes as input a plaintext, a key, and, optionally, associated data to be authenticated.

- HMAC is a keyed secure hash function.
- Function HMACVERIF takes the key, a message and an HMAC on the input and outputs `TRUE` if and only if the HMAC corresponds to the one generated by the message-key pair.
- INVITEVERIF is a supplementary function that checks a property of an invite code against the table of codes. Its properties depend on a specific implementation of the Scuttlebutt protocol.

Scuttlebutt uses the [13] cryptographic library as a provider for its primitives:

- `sha256` is used as an instance of SHA-2 family of hash functions.
- `nacl_auth` uses HMAC-SHA-512-256.
- `nacl_scalarmult` is a multiplication of an element by a scalar in the group of Curve25519 [6].
- `nacl_sign_detached` is an implementation of the Ed25519 signature scheme [14].
- `nacl_secret_box` is based on schemes XSalsa20 and Poly1305.

Below, we discuss the security properties and assumptions of these primitives in more detail.

2.2.1 SHA-2

SHA-256 is a member of SHA-2 family of hash functions designed by the United States National Security Agency. SHA-256 outputs a 256-bit long hash and is believed to offer roughly 128 bits of security. Collision attacks are known for reduced number of rounds on SHA-256 [15], but no attacks are presently known on the full algorithm.

SHA-2 is also included in the National Institute of Standards and Technology (NIST) Federal Information Processing Standards (FIPS) [16].

2.2.2 HMAC-SHA-512-256

HMAC-SHA-512-256 is a keyed secure hash function. The construction used in NaCl was formally defined in the work of Krawczyk, Bellare and Canetti [17].

Definition 1. *Suppose that we have a hash function* H *that applies a simple compression function to blocks of data of size* B*. Let opad be outer padding, consisting of repeated* B *bytes valued* $0x5c$*. Let ipad be the block-sized inner padding, consisting of repeated bytes valued* $0x36$*. Let* K' *be a secret key adjusted with padding to be of size* B*. The* HMAC *of a message* m *is defined as follows:*

$$\mathsf{HMAC}(K, m) = \mathsf{H}\left(\left(K' \oplus opad\right) \mid \mathsf{H}\left(\left(K' \oplus ipad\right) \mid m\right)\right)$$

$$K' = \begin{cases} \mathsf{H}(K) & K \text{ is larger than block size} \\ K & otherwise \end{cases}$$

In NaCL, the hash function used for keyed hashing is SHA-512. The suffix "-256" means that the output is the restricted to the first 256 bits of HMAC-SHA-512. The security guaranties of HMAC appear to be strong: Bellare [18] that HMAC is a PRF under the sole assumption that its ompression function is a PRF.

2.2.3 Ed25519

Ed25519 [14] is the EdDSA signature scheme using SHA-512 and Curve25519. EdDSA in turn is an elliptic curve signature scheme. The security of the signature relies on the Discrete-logarithm assumption and collision resistance of the hash function.

2.2.4 ChaCha20

NaCL [13] uses as a primitive the stream cipher XSalsa20 which is a version of Salsa20 with the nonce extended to 192 bits. XSalsa20 is provably secure if Salsa20 is secure with no additional assumptions [19].

There have been many attacks on a reduced number of rounds in Salsa20 but the best known attack breaks 8 of 20 rounds. In the work of Mouha and Preneel [20], it has been proved that 15-round of Salsa20 is 128-bit secure against differential cryptanalysis. The conjecture is that the Salsa20 output blocks, for a uniform random secret key k, are indistinguishable from independent uniform random 64-byte strings.

2.2.5 Poly1305

Poly1305 is a message authentication code and is used along with a stream cipher to achieve an AEADENC construction. Assuming the stream cipher is a PRF, Poly1305 is provably secure.

2.3 Security Goals

This analysis performed in this work stretches across all three of Scuttlebutt's subprotocols. In order to give the reader an understanding of the security goals that are taken into consideration, we provide Scuttlebutt's security goals, as they are described in the Scuttlebutt protocol specification [1]:

- **Mutual authentication.** *"The client must know the server's public key before connecting. The server learns the client's public key during the handshake. After a successful handshake the peers have verified each other's public keys."*
- **Shared secret agreement.** *"The handshake produces a shared secret that can be used with a bulk encryption*[4] *cipher [sic] for exchanging further messages."*
- **Server can abort before confirming identity.** *"Once the client has proven their identity the server can decide they don't want to talk to this client and disconnect without confirming their own identity."*
- **Identity hiding.** *"A man-in-the-middle cannot learn the public key of either peer."*
- **Network identifier hiding.** *"Both peers need to know a key that represents the particular Scuttlebutt network they wish to connect to, however a man-in-the-middle can't learn this key from the handshake."*
- **Network confirmation.** *"If the handshake succeeds then both ends have confirmed that they wish to use the same network."*
- **Replay attack resistance.** *"Past handshakes cannot be replayed. Attempting to replay a handshake will not allow an attacker to discover or confirm guesses about the participants' public keys."*

[4] "Bulk encryption" does not appear to differ from regular authenticated encryption.

- **Forward secrecy.** *"Handshakes provide forward secrecy. Recording a user's network traffic and then later stealing their secret key will not allow an attacker to decrypt their past handshakes."*

3 Secure Scuttlebutt

Scuttlebutt's security goals are spread across three Scuttlebutt sub-protocols, each with a specific purpose inside a complete Scuttlebutt application:

1. **Scuttlebutt's handshake protocol** achieves an authenticated key exchange (AKE) between two parties, Alice and Bob. This key is later used to encrypt a stream of content payloads (referred to as a *"feed"*). Scuttlebutt's handshake
2. **Scuttlebutt's invite protocol** allows members of a Scuttlebutt network or application (referred to as a *"pub"*) to delegate and manage access to said *"pub"*, by generating and communicating secret invitation tokens that can be consumed by the desired new participant.
3. **Scuttlebutt's private messaging protocol** allows for the easy instantiating of secure, end-to-end encrypted group chats within Scuttlebutt applications for up to 7 participants.

In this section, we go into these protocols in more detail.

3.1 Handshake Protocol

Scuttlebutt uses its own "Secret Handshake key exchange" [2] protocol in order to establish an authenticated shared secret between a client (here referred to as Alice) and a server (here referred to as Bob). The AKE occurs over a 2-RTT synchronous protocol that we illustrate in Figure 1.

Both parties have long-term identity key pairs, and the protocol assumes that Alice has prior knowledge of Bob's long-term public key g^B. During the key exchange, both parties will generate ephemeral key pairs unique to this session, which helps provide forward secrecy. Alice's long-term public key is never sent to Bob as plaintext but encrypted using a temporary session secret derived from the principals' long-term and ephemeral key pairs. This provides identity hiding for Alice with respect to a network observer.

An interesting implementation detail lies in the authenticated encryption nonce always being set to 0. This is not likely to be dangerous given the usage of ephemeral key pairs from both principals that are unique to each session.

In our analysis, we attempt to verify the attainability of the goals mentioned above and to verify in which contexts and scenarios they are attainable.

3.2 Invite Protocol

In order for a client (here Alice) to join the Pub and make her messages seen by its followers, she needs to use an invite code. The invite contains the Pub's long-term public key as well as a newly generated invite secret key. Invite redemption is achieved through a 2-RTT synchronous protocol shown in Figure 2.

During the key exchange, both parties will generate ephemeral key pairs unique to this session, which makes replay attacks impossible. The invite key pair is never sent to Bob as plaintext but encrypted using a temporary session secret derived from the principals' long-term and ephemeral key pairs.

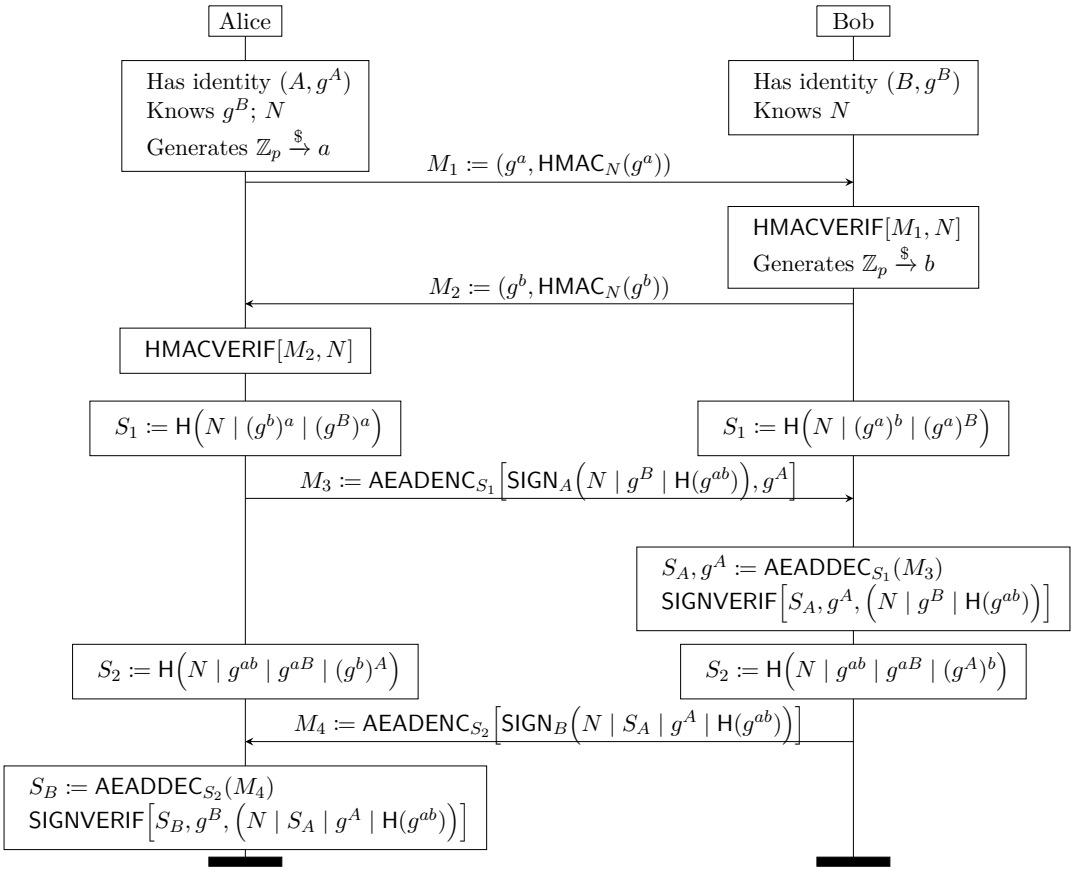

Figure 1: Secure Scuttlebutt's Authenticated Key Exchange (AKE) phase. Here, Bob acts as the server; Alice is assumed to have a pre-authenticated copy of Bob's long-term public key g^B before initializing the session. The AKE attempts to accomplish identity hiding with respect to Alice, key compromise impersonation resistance, and forward secrecy.

The protocol uses the same technique as the handshake, but instead of Alice's identity, it is her invite code that is being mutually authenticated.

3.3 Private Messaging

Scuttlebutt offers a mechanism for private messaging, where users can maintain secure, end-to-end group conversations accommodating up to seven participants. To send a private message to number of recipients, the client (Alice) posts an encrypted message to her feed. If one of the intended recipients sees Alice's message, they are then able to decrypt it. The scheme combines symmetric and asymmetric encryption as described in Figure 3.

For encryption, Alice generates a new key pair unique to each session and derives a pairwise shared secret with every recipient. The secrets are used only to encrypt the header, which contains the number of recipients and a symmetric key, used to encrypt the message itself. Embedding the number of recipients within the payload makes it impossible for the active attacker to change the number of recipients or to alter the message for different clients.

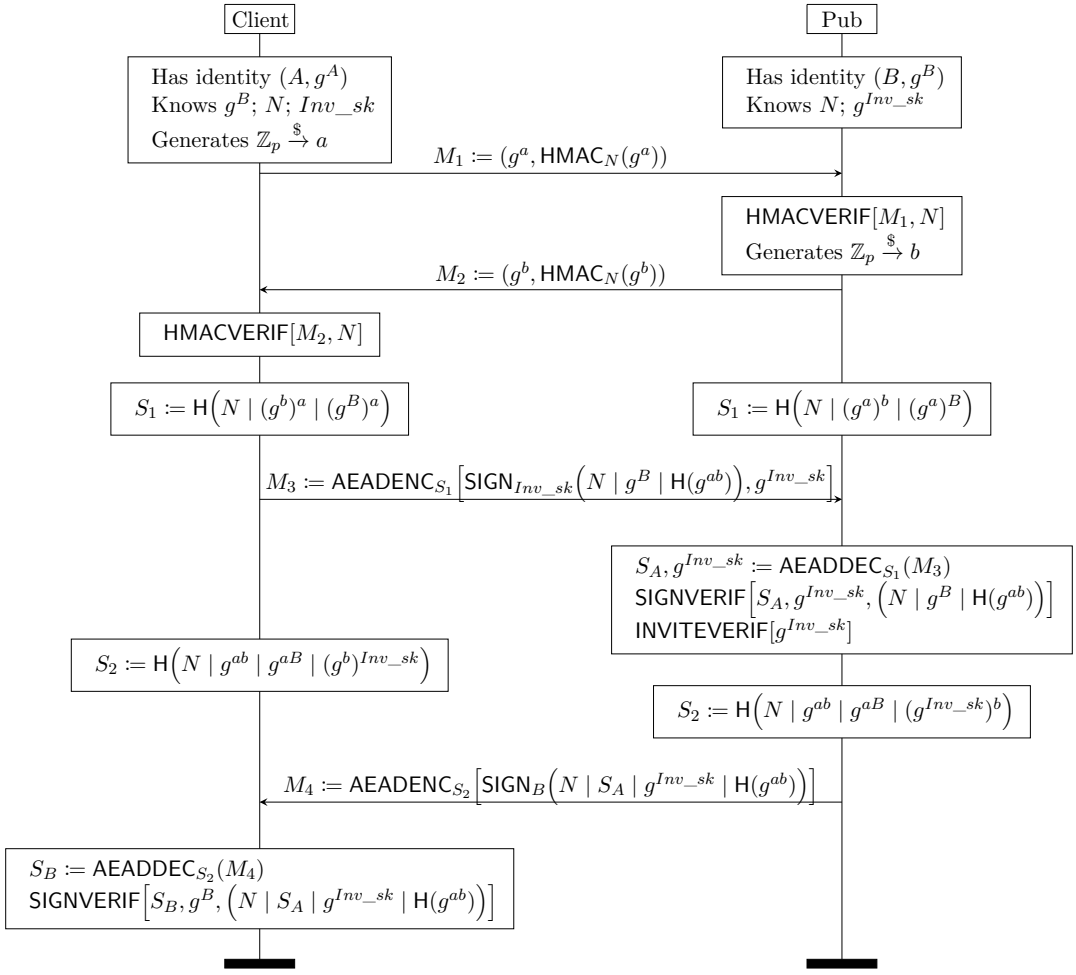

Figure 2: Secure Scuttlebutt's invite redeeming phase. Here, Pub acts as the server; Alice is assumed to have an invite before initializing the session. Invite redemtion protocol attempts to accomplish invite code hiding and its validity check.

4 Symbolic Verification with ProVerif

ProVerif is an automatic symbolic protocol verifier. It supports a wide range of cryptographic primitives, defined by rewrite rules or by equations [10]. In ProVerif, protocols are expressed using the applied pi-calculus [21], a language for modeling security protocols with a syntax inspired by standard ML. Let us examine how each component of a cryptographic protocol ends up being represented in the symbolic model (also referred to as the Dolev-Yao model [22]).

In the symbolic model, cryptographic primitives are represented as "perfect black-boxes"; a hash function, for example, cannot be modeled to be vulnerable to a length extension attack (which the hash function SHA-1 is vulnerable to, for example.) Encryption primitives are perfect pseudorandom permutations. Hash functions are perfect one-way maps. It remains possible to build complex primitives such as authenticated encryption with associated data (AEAD) and also to model interesting use cases, such as a Diffie-Hellman exponentiation that obtains a shared secret that is a low order element on the elliptic curve. However, in the latter case, such constructions cannot be based on an algebraic structure commonly considered

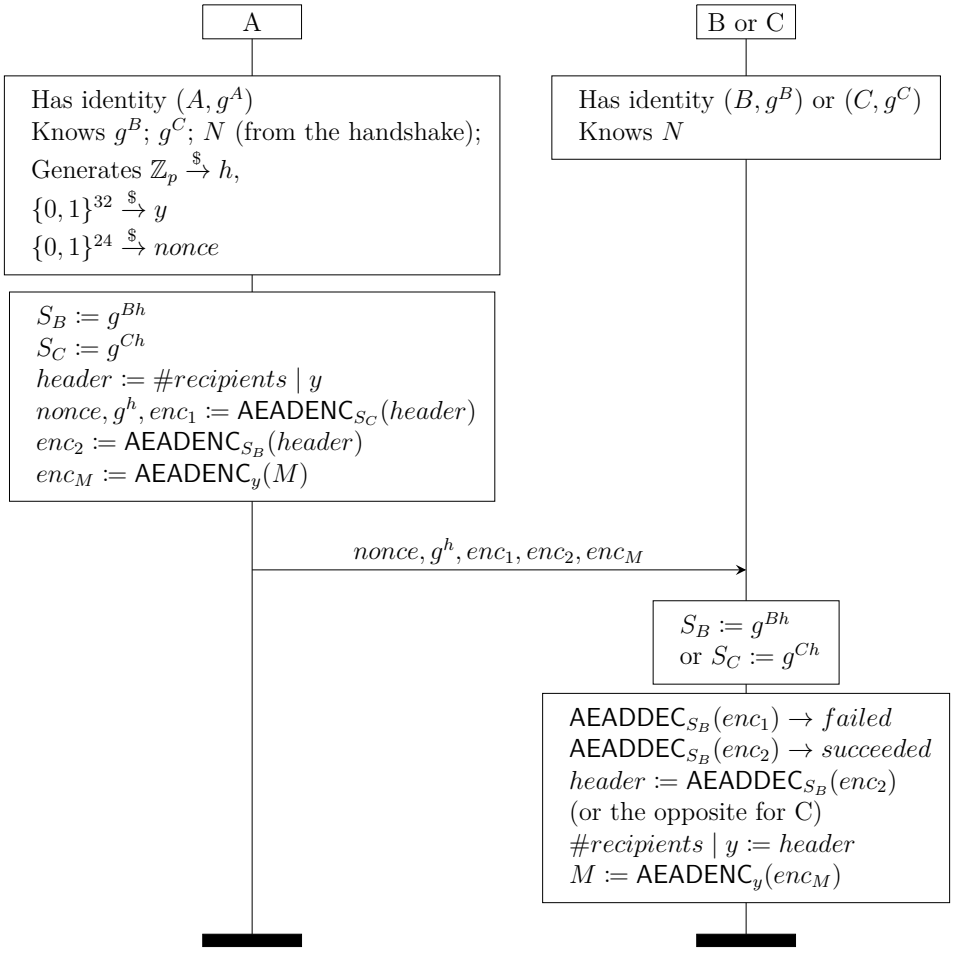

Figure 3: Private messaging in Scuttlebutt.

when modeling these scenarios, due to limitations of the equations that can be modeled in ProVerif(e.g. associativity of the group operation cannot be modeled)

When describing protocols for symbolic verification in ProVerif, the following constructions frequently come into play:

- **Protocol functions.** `let` and `letfun` declarations allow us to chain state transformations or processes in an interface that resembles functions in standard ML.
- **Principals.** Principals and their roles are inferred from how the protocol messages get constructed and exchanged on the network: after all of the protocol's constructions and functionalities are defined, a "top-level" process expresses the sequence in which operations occur and messages are broadcasted.

Informally, protocol verification in the symbolic model is concerned with modeling every possible execution of the described protocol and ensuring that queries hold in each of the resulting scenarios.

Symbolic verification is useful because it is fully automated whereas verification in the computational model tends to be semi-automated. It analyzes protocol messages as they are exchanged over the network under a simulated active attacker which will attempt to look for a contradiction to the query by delaying, dropping

and injecting messages. That said, ProVerif does not always terminate. ProVerif may sometimes produce an *"I don't know"* answer with more elaborate security queries: however, the effect of this is ProVerif finding false positives. ProVerif cannot miss a contradiction to a query should that contradiction exist. Deriving sound cryptographic proofs using symbolic analysis is still an open problem for real-world protocols [23].

A ProVerif script Σ is divided into two major parts:

1. $\Delta_1 \ldots \Delta_n$, a sequence of declarations which encapsulates all types, free names, queries, constructors, destructors, equations, pure functions and processes. Queries define the security properties to prove. Destructors and equations define the properties of cryptographic primitives.
2. P, the top-level process which then effectively employs $\Delta_1 \ldots \Delta_n$ as its toolkit for constructing a process flow for the protocol.

In processes, the replication $!P$ represents an unbounded number of copies of P in parallel.

Events are used for recording that certain actions happen (e.g. a message was sent or received), in order to use that information for defining security properties.

Phases model a global synchronization: processes initially run in phase 0; then at some point processes of phase 0 stop and processes of phase 1 run and so on. For instance, the protocol may run in phase 0 and some keys may be compromised after the protocol run by giving them to the adversary in phase 1.

More information on ProVerif is available in the ProVerif manual [24].

4.1 Modeling the Primitives

In this work, we model three subprotocols within Scuttlebutt: handshake, invite redemption, and private messaging. All three protocols share the same set of primitives:

4.1.1 SHA-256

SHA-256 is modeled as a perfect one way function:

$$\mathsf{fun\ hash(bitstring)}:\ \mathsf{bitstring}.$$

Since the destructor for this term is not defined, the attacker cannot obtain a preimage.

4.1.2 HMAC-SHA-512-256

HMAC-SHA-512-256 is modeled as a perfect keyed hash function. To verify the *hmac* we use the destructor *checkhmac*, equivalently we could have also used an equality test inside the top-level process:

$$\begin{aligned}&\mathsf{fun\ hmac(bitstring,\ hmac\_key)}:\ \mathsf{bitstring}.\\&\mathsf{reduc\ forall}\ m:\ \mathsf{bitstring},\ k:\ \mathsf{hmac\_key};\\&\mathsf{checkhmac(hmac}(m,\ k),\ m,\ k) =\ \mathsf{true}.\end{aligned}$$

4.1.3 Ed25519

Ed25519 is modeled as a detached signature which does not reveal a messge or the public key of the signer, but has an unexpected behaviour: If the key is *"weak"* (i.e. is an element of low order subgroup), then the signature verifies for any message. For the signature verification we provide a rewrite rule that returns a boolean value $true$ whenever the parameters are constructed correctly and $false$ otherwise.

fun get_pk(sk) : pk.
fun weak(sk) : sk.
fun sign(bitstring, sk) : bitstring.
fun checksign(bitstring, pk, bitstring) : bool
reduc forall m : bitstring, k : sk;
checksign(sign(m, k), get_pk(k), m) = true
otherwise forall $m1$: bitstring, $m2$: bitstring, k : sk;
checksign(sign($m1$, weak(k)), get_pk(weak(k)), $m2$) = true
otherwise forall $str1$: bitstring, $pkey$: pk, $str2$:
bitstring;
checksign($str1$, $pkey$, $str2$) = false.

4.1.4 XSalsa20-Poly1305

XSalsa20 and Poly1305 are modeled as perfect authenticated encryption with associated data. We skip the associated data entry in our model since it is not used in the protocol. We also create a zero nonce constant, as it is used by both parties, without trasmitting it over the network. We also allow incrementing and decrementing the nonce. For private messaging, the nonce is created as a fresh value for every session.

fun incnonce(nonce) : nonce.
reduc forall n : nonce;
decnonce(incnonce(n)) = n.
fun aeadenc(sym_key, nonce, bitstring) : bitstring.
fun aeaddec(sym_key, nonce, bitstring) : bitstring
reduc forall k : sym_key, n : nonce, m : bitstring;
aeaddec(k, n, aeadenc(k, n, m)) = (true, m)
otherwise forall k : sym_key, n : nonce, str : bitstring;
aeaddec(k, n, str) = (false, failmsg).

4.1.5 Curve25519

Curve25519 key derivation is defined as a classic exponentiation function with underlying commutativity equation. We then add custom handling for weak keys, as we did for Ed25519 in §4.1.3. If a weak key is given as one of the inputs, the derived shared secret is equal to a public constant that is known to the attacker.

```
const zero : sym_key [data].
fun exp(pk, sk) : sym_key.
equation forall x : sk, y : sk;
exp(get_pk(x), y) = exp(get_pk(y), x).
fun dhexp(pk, sk) : sym_key
reduc forall b : sk, a : sk;
dhexp(get_pk(weak(a)), b) = zero
otherwise forall b : sk, a : sk;
dhexp(get_pk(a), weak(b)) = zero
otherwise forall b : sk, a : sk;
dhexp(get_pk(a), b) = exp(get_pk(a), b).
```

4.2 Modeling the Handshake

We model the princpals Alice and Bob as two sub-processes that are executed in parallel in the main process and that communicate over a public channel. The active attacker has full control over the communication chanel, and so can read all trasmitted messages as well as insert any value at any point in time.

We model the key exchange with respect to the specifications given by its authors and primitives defined above. We also insert the events in order to run the correspondence queries.

Let us first describe the syntax for the queries in ProVerif. To model the security properties stated by the protocol designers, we need confidentiality eries and correspondence queries. In order to query for the confidentiality of value v (optionally before transition to the phase n) within symbolic model we write:

$$\text{query attacker}(v)[\text{phase } n].$$

The correpondence queries should be of the form:

$$\text{query } x_1 : t_1, \ldots, x_m : t_m;\ q_1 ==> q_2$$

where q_1, q_2 declares a conjunction or disjuction of events that can involve only the variables $x_1, \ldots, x_n$. $t_1, \ldots, t_n$ is a declaration of the types of the variabes.

We use phases in the model to test for the forward secrecy. In phase 0, parties perform an unbounded number of sessions where they derive a shared secret and transmit an encrypted message. In phase 1, Bob's secret key is leaked to the attacker, but the parties do not communicate anymore.

4.3 Modeling the Handshake: Queries and Results

We were able to verify the following queries for Scuttlebutt's handshake protocol:

$$\text{query attacker}(m).$$

m is a private constant message that Alice encrypts and sends to Bob after the key exchange is complete. The secrecy of the message implies the secrecy of the shared secret. We do not specify a phase, so the attacker can use his potential knowledge of Bob's secret key to try to decrypt previous sessions.

$$\text{query attacker}(\text{get\_pk}(skA))$$

The term get_pk(skA) stands for Alice's public key. It should not be learnt by the attacker in any phase.

$$\text{query attacker(get\_pk}(skB)\text{) phase } 0.$$

The term get_pk(skA) stands for Bob's public key. It is revealed to the attacker in phase 1, but it should remain secret before it is explicitly leaked.

$$\text{query attacker}(N).$$

The network identifier can be public or private depending on the choice of the developers. If it is private, it should not be learned by the attacker based on the protocol execution.

$$\begin{aligned}&\text{query } x : \text{pk},\ y : \text{hmac\_key};\\ &\text{event } (\text{ServerEndSession}(x,\ y)) ==> \text{event}(\text{ClientAccept}(x)) \ \&\&\\ &\text{event}(\text{ClientConfirmNetwork}(y)).\end{aligned}$$

This is an authentication query from Bob to Alice. Whenever Bob is able to complete the protocol, Alice should be able to confirm Bob's identity as well as their chosen network identifier.

$$\begin{aligned}&\text{query } x : \text{pk},\ y : \text{hmac\_key};\\ &\text{event}(\text{ClientEndSession}(x,\ y)) ==> \text{event}(\text{ServerAccept}(x)) \ \&\&\\ &\text{event}(\text{ServerConfirmNetwork}(y)).\end{aligned}$$

This is an authentication query from Alice to Bob. Whenever Alice is able to complete the protocol, Bob should be able to confirm Alice's identity as well as their chosen network identifier.

$$\begin{aligned}&\text{query } pkSender :\ \text{pk},\ mes : \text{bitstring};\\ &\text{inj} - \text{event}(\text{Receivedmessage}(pkSender,\ mes)) ==>\\ &\text{inj} - \text{event}(\text{Sentmessage}(pkSender,\ mes)).\end{aligned}$$

This is an authentication query for messages sent from Alice to Bob after the handshake is completed.

All the queries above are true (meaning the desired security properties hold) when the network identifier is chosen to be secret.

If the network identifier is public, as for example in the main Scuttlebutt network, there exists an attack that was previously documented by Cas Cremers and Dennis Jackson in their recent result [3]. It relies on the unexpected behaviour of the signature scheme whenever the key is weak, and on the weak shared secrets in the Diffie-Hellman key exchange on the Curve25519. Cremers and Jackson notice that the attacker can complete the handshake with Alice without prior knowledge of Bob's public key.

In our model, if the network identifier is public, the attacker in able to break two queries: the authentication of the message sent from Alice to Bob, as well as the authentication of Alice to Bob during the handshake:

- The attacker impersonates Alice and chooses his longterm and ephemeral key pairs to be weak.
- The attacker derives a shared secret with Bob which is now a constant value (i.e. it does not depend on Bob's public key anymore).

- The attacker cannot sign Bob's public key, which he needs to send as part of the third message of the protocol, so instead, the attacker signs their own term m. Since the key is weak, the resulting signature still passes verification.
- The final derived shared secret in this case is also a constant, so it is know by the attacker, rendering it usable to encrypt messages to Bob.

A simple fix for this attack, which was also mentioned in [3], would be to ensure that the members of the network are not allowed to use weak keys for encryption or singing. This can be easily accomplished by implementing key validation practices.

Remark 1. *If the Diffie-Hellman key derivation or the signature scheme is modeled classically, even with network identifier being public, ProVerif is not able to find the attacks above.*

4.4 Invite Redemption

The model of the invite redemption protocol is the same as for the handshake. The only difference we make are that:

1. Alice uses the invite key pair instead of her long-term key pair.
2. Alice does not send any message to the Pub after the invitation code is used.

The security queries are updated accordingly.

ProVerif can prove all the modified security queries except one: since the client (Alice) never sends the message to the server after confirming its identity, the server (i.e. Bob) can terminate without waiting for the client to confirm her identity. For this reason, the authentication of the server to the Client does not hold. We do not consider this attack to be practically dangerous, since the only person to decrypt Alice's invite code is the holder of the pub's secret key. So, even if Alice were to communicate the encrypted message to the attacker, no information can be derived from it.

4.5 Modeling Private Messaging

We model a restricted private messaging scenario with three participants: Alice, Bob, and Charlie. Alice sends an encrypted message $m1$, which is a private constant, to Bob and Charlie. Bob sends an encrypted message $m2$, to Alice and Charlie, and Charlie sends an encrypted message $m3$, to Alice but not to Bob. These three scenarios are executed in parallel for an unbounded number of sessions.

To model encryption and decryption of the messages, we employ four sub-processes:

1. EncryptM42($skMe$: sk, $pkReceiver1$: pk, $pkReceiver2$: pk, m_out : bitstring): the holder of $skMe$ encrypts a message m_out for participants with public keys $pkReceiver1$ and $pkReceiver2$.
2. DecryptM42($skMe$: sk, $pkSender$: pk): the holder of $skMe$ attemts to decrypt a message sent by a member with public key $pkSender$ for two people.
3. EncryptM42($skMe$: sk, $pkReceiver1$: pk, m_out : bitstring): the holder of $skMe$ encrypts a message m_out for a participant with public key $pkReceiver1$.
4. DecryptM41($skMe$: sk, $pkSender$: pk): the holder of $skMe$ attemts to decrypt a message sent by a member with public key $pkSender$ for one person.

We need to model differently the encryption for different numbers of participants, since a different number of term is transmitted over the channel. This approach can be extended to an unbounded number of participants in a straghtforward manner.

We also model for forward secrecy using phases. In phase 0, three parties exchange messages as described above. In phase 1, the secret long-term keys of all participats are leaked and the communication between them stops.

4.6 Modeling Private Messaging: Queries and Results

We were able to verify the following queries for Scuttlebutt's private messaging protocol:

$$\begin{aligned}&\mathsf{query\ attacker}(m1)\ \mathsf{phase}\ 0.\\&\mathsf{query\ attacker}(m2)\ \mathsf{phase}\ 0.\\&\mathsf{query\ attacker}(m3)\ \mathsf{phase}\ 0.\end{aligned}$$

These three queries test for forward secrecy of the messages $m1$, $m2$, $m3$ transmitted over the network.

$$\begin{aligned}&\mathsf{query}\ pkSender : \mathsf{pk},\ pkReceiver : \mathsf{pk},\ mes : \mathsf{bitstring};\\&\mathsf{event}\ (\mathsf{Receivedmessage}(pkSender,\ pkReceiver,\ mes)) ==>\\&\mathsf{event}\ (\mathsf{Sentmessage}(pkSender,\ pkReceiver,\ mes)).\end{aligned}$$

The query above checks authentication of all the messages that are sent.

$$\mathsf{query\ event}\ (\mathsf{Receivedmessage}(\mathsf{get\_pk}(skC),\ \mathsf{get\_pk}(skB),\ m3)).$$

This is an additional query to test if Bob can decrypt a message which Alice sends a message sends only to Charlie.

5 Computational Proofs with CryptoVerif

The Computational Security Model is often used by cryptographers. In this model the efficiency of the algorithms is calculated with respect to a security parameter. We model an adversary as a probabilistic Turing machine. It is restricted to run in polynomial time in security parameter. The security property holds if it can only occur with negligible probability in security parameter. In this case we say that the property holds asymptotically. We summarize the difference between two tools following the paper by Bruno Blanchet [10] and on the analysis of the WireGuard protocol [25] in CryptoVerif by Benjamin Lipp, Bruno Blanchet and Karthikeyan Bhargavan [26].

The symbolic model checks the logic of protocol design and assumes cryptographic primitives to be perfect, while the computational model is more realistic and relies on the probability of breaking the primitive. However, this results in more a more complex and demanding proof methodology. Furthermore, even computational models are still just models, and not take into account all possible real-world attacks, for example, they do not consider timing or side-channel attacks.

5.1 Game-based proofs

Our chosen proof technique in CryptoVerif depends on the notion of indistinguishability of two PPT algorithms.

Definition 2. *Computational Indistinguishability: Let* $\{D_\lambda\}_{\lambda\in N}$ *and* $\{E_\lambda\}_{\lambda\in N}$ *be two distribution ensembles indexed by a security parameter* λ *(which usually refers to the length of the input); we say they are computationally indistinguishable if for any non-uniform probabilistic polynomial time algorithm A, the following quantity is a negligible function in* λ*:*

$$\mu(\lambda) = \left| \Pr_{x\leftarrow D_\lambda}[A(x) = 1] - \Pr_{x\leftarrow E_\lambda}[A(x) = 1] \right|. \tag{1}$$

Remark 2. *Two PPT algorithms are called indistinguishable if their outputs are indistinguishable in the sense of the definition above.*

Many security properties in cryptography can be defined as a game between a challenger and an adversary. In the game the challenger executes the game, which is a PPT algorithm, on adversarial inputs and the gives the output to the adversary. The adversary then tries to derive information from the output. The probability distribution of the output by our security assumptions is called the target probability. In most of the cases it is equal to zero or for example 1/2 for guessing a bit, the difference between the target probability and the output the adversary obtains when playing the game is called the advantage of the adversary, and the security property holds if and only if for any PPT adversary the advantage in the corresponding game is negligible.

To prove a security property with a game-based proof, we construct a sequence of indistinguishable games in the last of which the advantage is easy to upper-bound by a negligible function. Since the difference between games output is negligible the overall advantage is also negligible.

5.2 Introduction to CryptoVerif

CryptoVerif is a proof assistant that performs indistinguishable transitions between games with aim to obtain a game where the property hold trivially. This sequence of games is a proof that the property holds asymptotically. The tool also records the probability of distinguishing games after every transition and outputs an exact bound based on the number of sessions of the protocol and the probabilities of breaking the primitives. This number is an explicit bound on the probability of breaking the scheme, it is called exact security. CryptoVerif has a builtin proving strategy it follows to choose which transformations to apply, it can also be done manually in the interactive mode. Possible transformations fall in the following categories [27]:

- Transitions based on indistinguishability - In this case, if the change made is detected by the adversary it implies an efficient method of distinguishing between two distributions that are indistinguishable by one of the security assumptions(e.g. IND-CPA, IND-CCA etc.)
- Transitions based on failure events - In such a transition, one argues that Games i and $i + 1$ are identical unless a certain "failure event" occurs the probability of the failure event should be negligible.
- Bridging steps - The third type of transition introduces a bridging step, which is typically a way of restating how certain quantities can be computed in a completely equivalent way. This step ought to make the proof easier to follow.

Protocol description in CryptoVerif consists of describing the primitives and the underlying assumptions, normally using a builtin library $default.cvl$, configuring

the type parameters, describing the security queries and then the protocol using the language of applied pi-calculus. Applied pi-calculus is also used to describe the protocols in ProVerif so translating the description from one tool to another is almost straightforward. The syntax is described in detail in the User Manual [28]

There are two differences in the syntax first being that CryptoVerif differentiates between input and output processes and in order for execution to not block after output in a value on the channel the process should turn to an input process, with first line waiting to receive a value, and put in the queue of input process, otherwise the rest of the calculations will never be executed. When a value is sent over the network a random waiting input process is chosen to process it. For this reason it is advised for all channels to have different names. Then the adversary has full control over the network: it can decide precisely from which copy of which input it receives a message and to which copy of which output it sends a message, by using the appropriate channel name.

The second difference is that we need to specify the types of the variables in more details, since now they are real bitstrings instead of abstract terms. For CryptoVerif to estimate the probability of collision every type is declared to be *bounded*, *fixed*, *large*, *password*, *sizen* where n is an integer. If a new variable of such type is created it is taken uniformly at random from the set of all bitstring in this type. If the user needs it to be nonuniform the type is defined with option *nonuniform*.

Other parts of the protocol can be translated without changes, which eliminates human error and allows us to reason about the protocol based on the result we already have from ProVerif model.

5.3 Cryptographic Primitives in the Computational Model

CryptoVerif comes with a library of predefined primitives for many classic assumptions. Following the description of the primitives in §2.2 we choose the following primitives in our model. [9]

5.3.1 SHA-2

SHA-256 is modeled as collision resistant hash function,
CollisionResistant_hash($key, hashinput, hashoutput, hash, hashoracle, Phash$) where:

- $key, hashinput, hashoutput$ are the types of corresponding structures.
- $hash$ is a name of a function being defined its type is $hash(key, hashinput) : hashoutput$.
- $hashoracle$ is a process that leaks the key of the hash to the adversary, so even if it is defined in the beginning of the game as a random element is it immediately leaked.
- $Phash$ stands for the probability of breaking collision resistance. Another option is to represent it as a perfect random oracle with no probability of collision being 0 and reason about the security property in terms of number of queries to the random oracle. This is a stronger assumption on a hash function so we would like to avoid it.

5.3.2 HMAC-SHA-512-256

HMAC-SHA-512-256 is modeled as a pseudo-random function following the definition is §2.2. The arguments of PRF($key, input, output, \mathsf{f}, Pprf$) are:

- $key, input, output$ the types of corresponding structures.
- $\mathsf{f}(key, input) : output$ is the primitive being defined
- $\mathsf{Pprf}(t, N, l)$ - probability of breaking pseudo-random function in time t, for one key, with N queries of length at most l.

5.3.3 Ed25519

Ed25519 is modeled as a probabilistic signature scheme that is unforgeable under chosen message attacks. While malleable, it does not take into account weak keys of a low order. CryptoVerif's default primitives do not capture the structure of the key pair from Ed25519, which relies on hardness of Elliptic Curve Discrete Logarithm Problem (ECDLP). The public and secret keys are both generated from a random seed in our model. The randomness used to make the signature probabilistic is generated internally. $\mathsf{UF\_CMA\_proba\_signature}(keyseed, pkey, skey, signinput, signoutput, skgen, pkgen, sign, check, Psign, Psigncoll)$ takes the following parameters:

- $keyseed, pkey, skey, signinput, sigoutput$ are types of the structures.
- $\mathsf{skgen}(keyseed) : skey$ is the secret key generation function.
- $\mathsf{pkgen}(keyseed) : pkey$ is the public key generation function.
- $\mathsf{sign}(signinput, skey) : signoutput$ is the signature function.
- $\mathsf{check}(signinput, pkey, signature) : bool$ is the verification function.
- $\mathsf{Psign}(t, N, l)$ is the probability of breaking the UF-CMA property in time t, for one key, N signature queries with messages of length at most l.
- Psigncoll is the probability of collision between independently generated keys.

5.3.4 XSalsa20-Poly1305

XSalsa20 and Poly1305 are modeled as an authenticated encryption with associated data (AEAD) construction. We assume it to have IND-CPA and INT-CTXT properties, which we already assume for the secret_box function, as described in in §2.2. $\mathsf{AEAD\_nonce}(key, cleartext, ciphertext, add\_data, nonce, enc, dec, injbot, Z, Penc, Pencctxt)$ has the following arguments:

- $key, cleartext, ciphertext, add\_data, nonce$ are types.
- $\mathsf{enc}(cleartext, add\_data, key, nonce)$: $ciphertext$ is the authenticated symmetric encryption function.
- $\mathsf{dec}(ciphertext, add\_data, key, nonce) : bitstringbot$ is the corresponding authenticated symmetric decryption function. The type *bitstringbot* is a type bitstring joint with a special constant *bottom* that decryption function outputs in case of failure. In the definition of the primitive it comes with a function *injbot (cleartext) : bitstringbot* which is a natural injection to *bitstringbot* type and can be used as a regular expressions to obtain the inverse.
- $\mathsf{Z}(cleartext) : cleartext$ is a function used in cryptographic transformations: on input message m, it returns a message of the same length consisting only of zeroes.
- $\mathsf{Penc}(t, N, l)$ - the probability of breaking the IND-CPA property in time t for one key, N encryption queries with cleartext of length at most l.

- $\mathsf{Pencctxt}(t, N, N\prime, l, l\prime, ld, ld\prime)$ - is the probability of breaking the INT-CTXT property in time t for one key, N encryption queries, with cleartext of length at most l and additional data for encryption of length at most ld, $N\prime$ decryption queries with ciphertext of length at most $l\prime$ and additional data of length at most $ld\prime$.

5.3.5 Curve25519

Curve25519 has its own primitive that captures low order elements $\mathsf{DH\_X25519}(\dots)$.

$\mathsf{DH\_X25519}(G, Z, g, exp, mult, subG, g_k, exp\_div\_k, exp\_div\_k\prime, pow\_k, subGtoG, zero, sub\_zero)$ models an elliptic curve defined by the equation $Y^2 = X^3 + AX^2 + X$ in the field $\mathbb{F}_p$ of non-zero integers, modulo the large prime p, where $A2 - 4$ is not a square modulo p. This curve must form a commutative group of order kq where k is a small integer and q is a large prime. Its parameters are:

- G: type of public keys
- $subG$: type of the subgroup of order q
- Z: type of exponents Z is the set of integers multiple of k, that is, exponents without weak keys.
- $g : G$ represents the base point.
- $exp(G, Z) : G$ exponentiation function.
- $g_k : subG = exp(g, k)$ is a generator of large order subgroup.
- $mult(Z, Z) : Z$ the multiplication function for exponents.
- $exp\_div\_k(subG, Z) : subG$ is defined by $exp\_div\_k(x, y) = X^{\frac{y}{k}}$.
- exp_div_k' symbol that replaces exp_div_k after game transformation, with the same definition as exp_div_k.
- $pow\_k(G) : subG$, defined by $pow\_k(x) = x^k$.
- $subGtoG(subG) : G$ is an embedding function.
- $zero : G$ is the public key 0.
- $sub\_zero : subG$ is 0, as an element of subG.

We also model the Gap Diffie-Hellman assumption $GDH(\dots)$ to be able to apply the corresponding transformation. $\mathsf{GDH}(G, Z, g, exp, exp\prime, mult, p)$ depends on:

- Parameters $G, Z, g, exp, mult$ should coincide if the ones defined in $DH\_X25519(\dots)$.
- $exp\prime(G, Z) : G$ is a symbol used to replace exp after game transformations.
- $p(t, N)$ is the probability of breaking the GDH assumption for one pair of exponents in time t with at most N calls to the decisional Diffie-Hellman oracle.

5.4 Security Queries

The set of possible queries in CryptoVerif is much smaller comparing to ProVerif since the proving technique is much more complex; for example, in ProVerif, one can check for indistinguishability of the variable from a random value of the same type, or check the impossibility of offline guessing attacks, or separate the algorithm on phases to check for forward secrecy. In CryptoVerif, we find support for only two

types of queries: secrecy queries, which correspond to a classic secrecy definition, and correspondence queries, which are meant to be used for protocol properties that are generally related to authentication.

Query declaration is of the form $\mathsf{query} x_1 : \mathsf{T}_1, \ldots, x_n : \mathsf{T}_n; \mathsf{Q}_1; \ldots; \mathsf{Q}_n$ where first we declare variables with x_i with their type T_i that occur in the correspondence queries declared after. Q_i can be one of the following:

- $\mathsf{secret}\ x$ [$\mathsf{public\_vars} l$] proves x is indistinguishable from a random value even when the variables in l are public. The adversary here performs several test to distinguish the value from random.
- $\mathsf{secret} x$[$\mathsf{public\_vars} l$][$\mathsf{cv\_onesession}$] same as above, but the adversary performs a single test to distinguish the value from random.
- $\mathsf{A} ==> \mathsf{B}$: A must be a conjunction of terms $\mathsf{event(e)}$, $\mathsf{inj-event(e)}$, $\mathsf{event(e}(\mathsf{M}_1, \ldots, \mathsf{M}_n))$, or $\mathsf{inj-event(e}(\mathsf{M}_1, \ldots, \mathsf{M}_n))$, where e is an event declared by event and the M_i are simple terms not containing events. B must be formed by conjunctions and disjunctions of terms $\mathsf{event(e)}$, $\mathsf{inj-event(e)}$, $\mathsf{event(e}(\mathsf{M}_1, \ldots, \mathsf{M}_n))$, or $\mathsf{inj-event(e}(\mathsf{M}_1, \ldots, \mathsf{M}_n))$, or simple terms not containing events.
- A where A is defined as above. This query is an abbreviation for $\mathsf{M} ==> \mathsf{false}$. Proving the $\mathsf{A} ==> \mathsf{B}$ where A and B are non-injective events means that if A was executed in by the process there was an occurrence of B that happened before. Proving $\mathsf{inj-event}$ means that for any n executions of the event declared before the arrows the event after the arrows was executed at least n times before.

In this work, we were able to prove an authentication property on the Scuttlebutt handshake and invite redemption subprotocols. For the handshake, we were able to prove the following two queries:

$$\mathsf{query}\ x : G, y : \mathsf{hmac\_key}; \mathsf{event}(\mathsf{ServerEndSession}(x, y)) ==> \mathsf{event}(\mathsf{ClientConfirmNetwork}(y)).$$

$$\mathsf{query}\ x : G, y : \mathsf{hmac\_key}; \mathsf{event}(\mathsf{ClientEndSession}(x, y)) ==> \mathsf{event}(\mathsf{ServerAccept}(x))\&\&\mathsf{event}(\mathsf{ServerConfirmNetwork}(y)).$$

Theorem 1. *(Initiator Authentication and Network Confirmation) If the initiator of the communication is able to complete the session it is authenticated to the responder. When parties complete the key exchange they agree on the network identifier.*

Proof. The CryptoVerif model is built with the primitives described above and the process description is translated directly from the ProVerif model. The proof is done automatically by CryptoVerif. □

The same queries were proved for the invite redemption protocol, but the formulation of the theorem is somewhat different due to the context:

Theorem 2. *(Client Authentication and Network Confirmation) If the client is able to complete the session, then his invitation key is accepted by the Pub. If parties were able to complete the protocol, then they agree on the network identifier.*

Remark 3. *The CryptoVerif model is built with the primitives described above and the process description is translated directly from the corresponding ProVerif model. Due to similarities between the proofs and their size we only present the proof for the previous theorem.*

6 Conclusion and Future Work

We have presented the first comprehensive analysis of the entire Scuttlebutt suite of protocols in the symbolic model. We have also presented the first game-based computational proof on certain authentication elements of Scuttlebutt's handshake and invite protocols.

CryptoVerif is able to produce proofs only if guided manually, and the art of assisting the deducation of the proof is delicate. Some transformations, which may be obvious to a seasoned cryptographer, are sometimes not performed by the tool. In other cases additional functions need to be defined to help perform the transformation. In the future, we would like to enhance out model with the transformations that it lacks in order to complete our proofs with additional queries.

Despite the presence of clear ground for a more complete coverage of Scuttlebutt, we hope that this work, in its present state, can constitute the most solid available groundwork yet for future, deeper analysis of Scuttlebutt, and that it can also serve as an example of the strength and diversity of the analysis that is achievable by combining formal verification approaches in both symbolic and computational models.

Acknowledgments. This work was accomplished during a summer research internship at Symbolic Software, under the direction of Prof. Nadim Kobeissi. We also sincerely thank Bruno Blanchet and Benjamin Lipp for answering questions pertaining to CryptoVerif.

References

[1] Secure Scuttlebutt Consortium. Scuttlebutt protocol guide. `https://ssbc.github.io/scuttlebutt-protocol-guide/`, 2016. [Online; accessed 20-August-2019].

[2] Dominic Tarr. Designing a secret handshake: Authenticated key exchange as a capability system. 2015.

[3] Cas Cremers and Dennis Jackson. Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman. *IEEE Computer Security Foundations Symposium (CSF)*, 19, 2019.

[4] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In Stephen Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 78–94. IEEE, 2012.

[5] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against diffie-hellman. In *NDSS*, 2017.

[6] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[7] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *International Conference on Principles of Security and Trust*, pages 226–246. Springer, 2013.

[8] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In *Foundations of Security Analysis and Design VII*, pages 54–87. Springer, 2013.

[9] Bruno Blanchet. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar on Applied Formal Protocol Verification*, page 117, 2007.

[10] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In *Principles of Security and Trust (POST)*, pages 3–29, 2012.

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

[12] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 435–450. IEEE, 2017.

[13] Daniel J. Bernstein. Cryptography in NaCl. 2009.

[14] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[15] Florian Mendel; Tomislav Nad; Martin Schläffer. Improving local collisions: New attacks on reduced SHA-256. *Advances in Cryptology – EUROCRYPT*, 2013.

[16] Information Technology Laboratory. Secure hash standard, FIPS 180-4. 2015.

[17] Huo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. 2013.

[18] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. *Advances in Cryptology - CRYPTO*, 2006.

[19] Daniel J. Bernstein. Extending the Salsa20 nonce. 2012.

[20] Bart Preneel Nicky Mouha. Towards finding optimal differential characteristics for ARX: Application to Salsa20. 2013.

[21] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018.

[22] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[23] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46(3-4):225–259, 2011.

[24] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. ProVerif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial. *Version from*, pages 05–16, 2018.

[25] Jason A Donenfeld. WireGuard: Next generation kernel network tunnel. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[26] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. A mechanised cryptographic proof of the WireGuard virtual private network protocol. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[27] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. IACR Cryptology ePrint Archive, 2004. `http://eprint.iacr.org/2004/332`.

[28] CryptoVerif: Computationally sound, automatic cryptographic protocol verifier user manual. *Bruno Blanchet and David Cadé*, 2012.