Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach

Nadim Kobeissi, Karthikeyan Bhargavan, Bruno Blanchet

Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach

Nadim Kobeissi
INRIA Paris

Karthikeyan Bhargavan
INRIA Paris

Bruno Blanchet
INRIA Paris

***Abstract*—Many popular web applications incorporate end-to-end secure messaging protocols, which seek to ensure that messages sent between users are kept confidential and authenticated, even if the web application's servers are broken into or otherwise compelled into releasing all their data. Protocols that promise such strong security guarantees should be held up to rigorous analysis, since protocol flaws and implementations bugs can easily lead to real-world attacks.**

We propose a novel methodology that allows protocol designers, implementers, and security analysts to collaboratively verify a protocol using automated tools. The protocol is implemented in ProScript, a new domain-specific language that is designed for writing cryptographic protocol code that can both be executed within JavaScript programs and automatically translated to a readable model in the applied pi calculus. This model can then be analyzed symbolically using ProVerif to find attacks in a variety of threat models. The model can also be used as the basis of a computational proof using CryptoVerif, which reduces the security of the protocol to standard cryptographic assumptions. If ProVerif finds an attack, or if the CryptoVerif proof reveals a weakness, the protocol designer modifies the ProScript protocol code and regenerates the model to enable a new analysis.

We demonstrate our methodology by implementing and analyzing a variant of the popular Signal Protocol with only minor differences. We use ProVerif and CryptoVerif to find new and previously-known weaknesses in the protocol and suggest practical countermeasures. Our ProScript protocol code is incorporated within the current release of Cryptocat, a desktop secure messenger application written in JavaScript. Our results indicate that, with disciplined programming and some verification expertise, the systematic analysis of complex cryptographic web applications is now becoming practical.

1. Introduction

Designing new cryptographic protocols is highly error-prone; even well-studied protocols, such as Transport Layer Security (TLS), have been shown to contain serious protocol flaws found years after their deployment (see e.g. [1]). Despite these dangers, modern web applications often embed custom cryptographic protocols that evolve with each release. The design goal of these protocols is typically to protect user data as it is exchanged over the web and synchronised across devices, while optimizing performance for application-specific messaging patterns and deployment constraints. Such custom protocols deserve close scrutiny, but their formal security analysis faces several challenges.

First, web applications often evolve incrementally in an ad hoc manner, so the embedded cryptographic protocol is only ever fully documented in source code. Even when protocol designers or security researchers take the time to create a clear specification or formal model for the protocol, these documents are typically incomplete and quickly go out-of-date. Finally, even if the protocol itself is proved to be secure, bugs in its implementation can often bypass the intended security guarantees. Hence, it is not only important to extract a model of the protocol from the source code and analyze its security, it is essential to do so in a way that the model can evolve as the application is modified.

In this paper, we study the protocol underlying the Signal messaging application developed by Open Whisper Systems. Variants of this protocol have also been deployed within WhatsApp, Facebook Messenger, Viber, and many other popular applications, reaching over a billion devices. The protocol has been known by other names in the past, including Axolotl, TextSecure (versions 1, 2, and 3), and it continues to evolve within the Signal application under the name Signal Protocol. Until recently, the main documentation for the protocol was its source code, but new specifications for key components of the protocol have now been publicly released.[1]

Signal Protocol has ambitious security goals; it enables asynchronous (zero round-trip) authenticated messaging between users with end-to-end confidentiality. Each message is kept secret even if the messaging server is compromised, and even if the user's long term keys are compromised, as long as these keys are not used by the attacker before the target message is sent (forward and future secrecy.) To achieve these goals, Signal uses a novel authenticated key exchange protocol (based on mixing multiple Diffie-Hellman shared secrets) and a key refresh mechanism (called double ratcheting). The design of these core mechanisms in TextSecure version 2 was cryptographically analyzed in [2] but the protocol has evolved since then and the security of Signal as it is currently implemented and deployed remains an open question.

In fact, although they all implement the same core protocol, different implementations of the Signal protocol

1. https://whispersystems.org/docs/specifications/x3dh/

vary in important details, such as how users are identified and authenticated, how messages are synchronised across devices, etc. We seek to develop and analyze one such variant that was recently incorporated into Cryptocat, a desktop messaging application developed by one of the current authors. We call this variant SP in the rest of this paper. We develop a detailed model for SP in the applied pi calculus and verify it using the ProVerif protocol analyzer [3] for these security goals against adversaries in a classic Dolev-Yao model [4]. We also develop a computational proof for SP using the CryptoVerif prover [5]. There remains the challenge of keeping our models up-to-date as the protocol code evolves within Cryptocat. To this end, we design a model extraction tool that can compile the protocol source code to the applied pi calculus.

Signal has been implemented in various programming languages, but most desktop implementations of Signal, including Cryptocat, are written in JavaScript. Although JavaScript is convenient for widespread deployability, it is not an ideal language for writing security-critical applications. Its permissive, loose typing allows for dangerous implementation bugs and provides little isolation between verified cryptographic protocol code and unverified third-party components. Rather than trying to verify general JavaScript programs, we advocate that security-critical components like SP should be written in a well-behaved subset that enables formal analysis.

We introduce ProScript (short for *"Protocol Script"*), a programming and verification framework tailored specifically for the implementation of cryptographic protocols. ProScript extends Defensive JavaScript (DJS) [6], [7], a static type system for JavaScript that was originally developed to protect security-critical code against untrusted code running in the same origin. ProScript is syntactically a subset of JavaScript, but it imposes a strong coding discipline that ensures that the resulting code is amenable to formal analysis. ProScript programs are mostly self-contained; they cannot call arbitrary third-party libraries, but are given access to carefully implemented (well-typed) libraries such as the ProScript cryptographic library (PSCL). Protocols written in ProScript can be type-checked and then automatically translated into an applied pi calculus [8] model using the ProScript compiler. The resulting model can be analyzed directly through ProVerif and can be adapted and extended to a proof in CryptoVerif. As the code evolves, this model can be automatically refreshed to enable new analyses and proofs, if necessary.

Contributions. We present an outline of our contributions in this paper:

A Security Model and New Attacks. We present security goals and a threat model for secure messaging (§ 2). As a motivation for our verification approach, we discuss protocol weaknesses and implementation bugs in the messaging protocol underlying the popular Telegram application.

Automated Model Extraction from JavaScript. We present the ProScript compiler, which allows for the compilation from a subset of JavaScript into a readable protocol model in the applied pi calculus (§4). Model extraction enables formal verification to keep up with rapidly changing source code. Readable models allow the protocol analyst to experiment with different threat models and security goals and to test new features before including them in the implementation.

A Symbolic Security Analysis of SP. We formalize and analyze a variant of Signal Protocol for a series of security goals, including confidentiality, authenticity, forward secrecy and future secrecy, against a classic symbolic adversary (§5). Our analysis uncovers several weaknesses, including previously unreported replay and key compromise impersonation attacks, and we propose and implement fixes which we then also verify.

A Computational Cryptographic Proof for SP. We present proofs of message authenticity, secrecy and forward secrecy for SP obtained using the CryptoVerif computational model prover [5]. (§6)

A Verified Protocol Core for Cryptocat. We integrate our verified protocol code into the latest version of Cryptocat[2] (§7), a popular open source messaging client with thousands of users that is developed and maintained by one of the authors of this paper. We show how the new architecture of Cryptocat serves to protect the verified protocol code from bugs in the rest of the application.

2. A Security Model for Encrypted Messaging

We consider a simple messaging API as depicted below. An initiator A can start a conversation with B by calling `startSession` with long-term secrets for A and any identity credentials it has for B. This function returns the initial conversation state T_0. Thereafter, the initiator can call `send` with a plaintext message M_1 to obtain the encrypted message E_1 that it needs to send on the network. Or it can call `recv` with an encrypted message E_2 it received (supposedly from B) to obtain the plaintext message M_2.

```
T_0^ab = startSession(secrets_A, identity_B)
T_1^ab, E_1 = send(T_0^ab, M_1)
T_2^ab, M_2 = recv(T_1^ab, E_2)
```

The responder B uses a similar API to accept sessions and receive and send messages:

```
T_0^ba = acceptSession(secrets_B, identity_A)
T_1^ba, M_1 = recv(T_0^ba, E_1)
T_2^ba, E_2 = send(T_1^ba, M_2)
```

We deliberately chose a functional state-passing API with no side-effects in order to focus on cryptographic protocol computations, rather than the concrete details of how these messages are sent over the network.

2.1. Threat Model

While threat models vary for different protocols, we consider the following threats in this paper:

2. https://crypto.cat

- **Untrusted Network** We assume that the attacker controls the network and so can intercept, tamper with and inject network messages (e.g. E_1, E_2). Moreover, if two messaging clients communicate via a server, we typically treat that server as untrusted.
- **Malicious Principals** The attacker controls a set of valid protocol participants (e.g. M), for whom it knows the long-term secrets. The attacker may advertise any identity key for its controlled principals; it may even pretend to own someone else's identity keys.
- **Long-term Key Compromise** The attacker may compromise a particular principal (e.g. A) during or after the protocol, to obtain her long-term secrets.
- **Session State Compromise** The attacker may compromise a principal to obtain the full session state at some intermediate stage of the protocol (e.g. T_1^{ab}).

2.2. Cryptographic Models

Traditionally, symbolic cryptographic models have been particularly suitable for automated protocol analysis. They ignore attacks with negligible probability and assume that each cryptographic function is a perfect black-box. For example, in such models, hash functions never collide and encryption is a message constructor that can only be reversed by decryption. In the *computational model*, cryptographic primitives are functions over bitstrings and their security is specified in terms of probabilities. These models are more precise and closer to those used by cryptographers, but usually do not lend themselves to fully automated proofs. Generally, we will use symbolic models when we are trying to find attacks that rely on logical flaws in the protocol and in its use of cryptographic primitives. We will use computational models when we want to build a cryptographic proof of security, starting from standard cryptographic assumptions.

2.3. Security Goals

We state a series of semi-formal security goals in terms of our generic messaging API. We use the phrase "A sends a message M to B" to mean that A calls $\mathsf{Send}(T, M)$ with a session state T that represents a conversation between A and B. Similarly, we say that "B receives a message M from A" to mean that B obtained M as a result of calling $\mathsf{Recv}(T, E)$ with a session T with A.

Unless otherwise specified, the following security properties assume that both A and B are honest, that is, their long-term secrets have not been compromised. We begin with several variants of authenticity goals:

- **Message Authenticity** If B receives a message M from A, then A must have sent M to B.
- **No Replays** Each message received by B from A corresponds to a unique message sent by A. That is, the attacker must not be able to get a single message sent by A to be accepted twice at B.
- **No Key Compromise Impersonation** Even if the long-term secrets of B are compromised, message authenticity must hold at B. That is, the attacker must not be able to forge a message from A to B.

Our definition of message authenticity covers integrity as well as sender and recipient authentication. Obtaining message authenticity also helps prevent *unknown key share* attacks, where B receives a message M from A, but A sent that message to a different intended recipient C. We define four confidentiality goals:

- **Secrecy** If A sends some secret message M to B, then nobody except A and B can obtain M.
- **Indistinguishability** If A randomly chooses between two messages M_0, M_1 (of the same size) and sends one of them to B, the attacker cannot distinguish (within the constraints of the cryptographic model) which message was sent.
- **Forward Secrecy** If A sends a secret message M to B and if A's and B's long-term secrets are subsequently compromised, the message M remains secret.
- **Future Secrecy** Suppose A sends M in a session state T, then receives N, then sends M'. If the session state T is subsequently compromised, the message M' remains secret.

Some protocols satisfy a weaker notion of forward secrecy, sometimes called *weak* forward secrecy, where an attacker is not allowed to actively tamper with the protocol until they have compromised the long-term keys [9]. Some messaging protocols also seek to satisfy more specific authenticity and confidentiality goals, such as non-repudiation and plausible deniability. We will ignore them in this paper.

In the next section, we evaluate two secure messaging applications against these goals, we find that they fail some of these goals due to subtle implementation bugs and protocol flaws. Hence, we advocate the use of automated verification tools to find such attacks and to prevent their occurrence.

3. Analyzing Real-World Messaging Protocols

Modern messaging and transport protocols share several distinctive features [10]: for example, Signal Protocol, SCIMP, QUIC and TLS 1.3 share a strong focus on asynchronous key agreement with a minimum of round trips. Some also guarantee new security goals such as future secrecy. The protocols also assume non-standard (but arguably more user-friendly) authentication infrastructures such as Trust-on-First-Use (TOFU). Modern messaging protocols have several interesting features and properties that set them apart from classic cryptographic protocols and upon which we focus our formal verification efforts:

New Messaging Patterns. In contrast to connection-oriented protocols, modern cryptographic protocols are constrained by new communication flows such as zero-round-trip connections and asynchronous messaging, where the peer may not even be present.

Confidentiality Against Strong Adversaries. Web messaging protocols need to be robust against server compromise

and device theft and so seek to provide strong and novel forward secrecy guarantees.

Weak Authentication Frameworks. Many of these protocols do not rely on public key infrastructures. Instead they may authenticate peers on a TOFU basis or even let peers remain anonymous, authenticating only the shared connection parameters.

Code First, Specify Later. Unlike Internet protocols, which are designed in committee, these protocols are first deployed in code and hand-tuned for performance on a particular platform. The code often remains the definitive protocol specification.

Before outlining our verification approach for such protocols, we take a closer look at two messaging applications: Telegram and Cryptocat.

3.1. Secret Chats in Telegram

Our first example is the "MTProto" [11] secure messaging protocol used in the Telegram messaging application. We focus on the "secret chat" feature that allows two Telegram clients who have already authenticated themselves to the server to start an encrypted conversation with each other. Although all messages pass through the Telegram server, the server is untrusted and should not be able to decrypt these messages. The two clients A and B download Diffie-Hellman parameters from the Telegram server and then generate and send their public values to each other.

The key exchange is not authenticated with long-term credentials. Instead, the two clients are expected to communicate out-of-band and compare a SHA-1 hash (truncated to 128-bits) of the Diffie-Hellman shared secret. If two users perform this authentication step, the protocol promises that messages between them are authentic, confidential, and forward secret, even if the Telegram server is compromised. However this guarantee crucially relies on several cryptographic assumptions, which may be broken either due to implementation bugs or computationally powerful adversaries, as we describe below.

Malicious Primes. MTProto relies on clients checking that the Diffie-Hellman configuration (p, g) that they received from the server is suitable for cryptographic use. The specification requires that p be a large safe prime; hence the client must check that it has exactly 2048 bits and that both p and $(p-1)/2$ are prime, using 15 rounds of the Miller-Rabin primality test. There are several problems with this check. First, the server may be able to carefully craft a non-prime that passes 15 rounds of Miller-Rabin. Second, checking primality is not enough to guarantee that the discrete log problem will be hard. If the prime is chosen such that it has "low weight", the SNFS algorithm applies, making discrete logs significantly more practical [12]. Even if we accept that primality checking may be adequate, it is unnecessary for an application like Telegram, which could simply mandate the use of well-known large primes instead [13].

Public Values in Small Subgroups. A man-in-the-middle can send to both A and B public Diffie-Hellman values g^b and g^a equal to 1 (resp. 0, resp. $p-1$). Both A and B would then compute the shared secret as $g^{ab} = 1$ (resp. 0, resp. 1 or -1). Since their key hashes match, A and B think they have a confidential channel. However, the attacker can read and tamper with all of their messages. More generally, MTProto relies on both peers verifying that the received Diffie-Hellman public values do not fall in small subgroups. This check is adequate to prevent the above attack but could be made unnecessary if the two public values were to be authenticated along with the shared secret in the hash compared by the two peers.

Implementation Bugs in Telegram for Windows. The above two weaknesses, reported for the first time in this paper, can result in attacks if the protocol is not implemented correctly. We inspected the source code for Telegram on different platforms; while most versions perform the required checks, we found that the source code for Telegram for Windows Phone did not check the size of the received prime, nor did it validate the received Diffie-Hellman values against 1, 0 or $p-1$. We reported both bugs to the developers, who acknowledged them and awarded us a bug bounty.

Such bugs and their consequent attacks are due to missed security-relevant checks, and they can be found automatically by symbolic analysis. For example, [14] shows how to model unsafe (malicious) primes and invalid public keys in ProVerif and uses this model to find vulnerabilities in several protocols that fail to validate Diffie-Hellman groups or public keys.

MTProto is also known to other cryptographic weaknesses [15], [16]. How can we be sure that there are no other protocol flaws or implementation bugs hiding in MTProto? Any such guarantee would require a systematic security analysis of both the protocol and the source code against both symbolic and computational adversaries.

3.2. A New Protocol for Cryptocat

Cryptocat is a secure messaging application that is written in JavaScript and deployed as a desktop web application. Earlier versions of Cryptocat implement a variant of the OTR (Off-The-Record) messaging protocol [17] which suffers from several shortcomings. It does not support asynchronous messaging, so both peers have to be online to be able to message each other. It does not support multiple devices or encrypted file transfer. OTR also uses legacy cryptographic constructions like DSA signatures and prime-field Diffie-Hellman, which are slower and less secure than more modern alternatives based on elliptic curves. Furthermore, Cryptocat peers did not have long-term identities and so the authentication guarantees are weak. Early version of Cryptocat suffered from many high-profile implementation bugs, including the reuse of initialization vectors for file encryption [18], bad random number generation, and a classic JavaScript type flaw that resulted in a private key of 255 bits being coerced into a string that held only 55 bits. Some of these implementation flaws would have been found using a static type checker, others required deeper analysis.

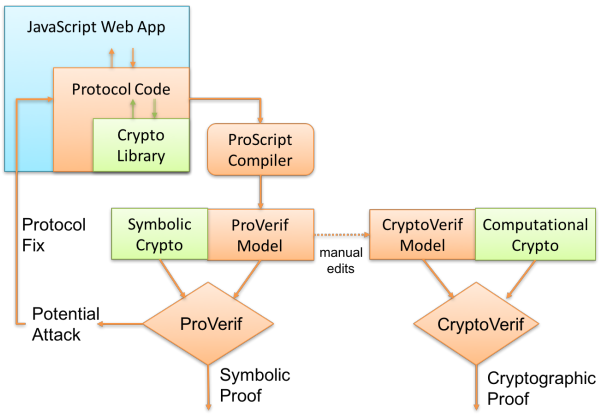

Figure 1: Verification Approach. A ProVerif model is automatically extracted from ProScript protocol code and analyzed for its security goals against a symbolic attacker. The model is then edited by hand and extended with cryptographic assumptions and intermediate lemmas to build a computational proof that is verified by CryptoVerif.

Cryptocat was recently rewritten from scratch to upgrade both its messaging protocol and its implementation. The goal of this redesign was to isolate its protocol core and replace it with a verified messaging protocol written in a statically typed subset of JavaScript.

3.3. Towards Automated Verification

The innovative designs and unusual security guarantees of secure messaging protocols demand formal security analysis. Hand-written models with detailed cryptographic proofs can be useful as a reference, but we observe that the most recent analysis of Signal Protocol [2] is already out of date, as the protocols have moved on to new versions. Furthermore, manual cryptographic proofs often leave out details of the protocol for simplicity and some of these details (e.g. client authentication) may lead to new attacks. In this paper, we advocate the use of automated verification tools to enable the analysis of complex protocols as they evolve and incorporate new features. Moreover, we would also like to find protocol implementation bugs (like the ones in previous versions of Telegram and Cryptocat) automatically.

We advocate the verification approach depicted in Figure 1. The messaging application is written in JavaScript and is broken down into a cryptographic protocol core and untrusted application code that interact through a small well-typed API that hides all protocol secrets within the protocol core and only offers a simple send/receive functionality to the application. Notably, the protocol core is written in a domain-specific language and does not rely on any external libraries except for a well-vetted cryptographic library. The protocol code can be translated to an applied pi calculus model and symbolically analyzed in ProVerif to find protocol flaws and attacks. The model can also be used as the starting point for a cryptographic proof for the protocol developed using CryptoVerif.

In the rest of this paper, we show how we applied this verification methodology to systematically analyze a variant of the Signal protocol, called SP, that is implemented in the new version of Cryptocat.

4. ProScript: A Language for Protocol Implementation

ProScript aims to be an ideal language for reliably implementing cryptographic protocols for web applications. Using ProScript, a protocol designer or implementer can implement a protocol, automatically extract a formal model from the code, verify the model using ProVerif, and then run the protocol code within a JavaScript web application. The ProScript framework does not target general JavaScript code, however existing applications can be adapted to use ProScript for their security-critical protocol components.

Our goal is to allow the developer to go back and forth between their protocol implementation and the ProVerif model, in order to help understand the behavior being illustrated, the properties being verified and how detected attacks, if any, relate to their source code. For these reasons, we pay special attention to generating models that are optimized both for verifiability as well as readability. This increases their utility to a human examiner who may wish to independently expand the model to include more specific process flows or to explore variations of the protocol against a manually defined adversary.

Syntactically, ProScript is a subset of JavaScript that can be naturally translated to the applied pi calculus. This restriction produces casualties, including recursion, `for` loops and extensible objects. A closer look at the ProScript syntax shows JavaScript employed in a particular style to bring out useful features:

Isolation. ProScript is based on Defensive JavaScript (DJS) [6], [7], a typed subset of JavaScript which focuses on protecting security-critical components from malicious JavaScript code running in the same environment. DJS imposes a strict typing discipline in order to eliminate language-based attacks like prototype poisoning. In particular, it forbids the use of unknown external libraries as well as calls to tamperable object methods such as `.toString()`. It also forbids extensible objects and arrays and prevents any access to object prototypes. These restrictions result in protocol implementations that are more robust and less influenced by the runtime environment. The ProScript type-checker builds on and extends DJS and hence, inherits both its language restrictions and isolation guarantees.

Type Declarations and Inference. ProScript requires all variables and functions to be declared before they are used, hence imposing a strict scoping discipline. For example, an expression $v.x$ is well-typed if and only if v has been defined, as a local or global variable, to be an object with a property x. As an extension to the builtin types of DJS, ProScript allows type declarations for commonly used protocol

data structures. For example, an array of 32 hexadecimal integers can be declared as a key type. The ProScript compiler recognizes such type declarations and uses them to translate the code into more concise and informative ProVerif models. Moreover, the typechecker can automatically infer fine-grained sub-types. For example, ProScript differentiates between numbers declared using decimal literals (ex. `128`) and hexadecimal literals (ex. `0x80`). Numbers defined using hexadecimal are sub-typed as bytes. This feature allows us to track how numerical values are employed in the protocol, and prevents type coercion bugs similar to an actual bug that we describe in §3.2, where a significant loss of entropy was caused by a byte being coerced into a decimal value.

State-Passing Functional Style. ProScript's syntax takes advantage of JavaScript's functional programming features in order to encourage and facilitate purely functional protocol descriptions, which the compiler can translate into symbolically verifiable, human-readable models in the applied pi calculus. The functional style encourages the construction of state-passing functions, leaving state modification up to the unverified application outside of the ProScript code. The majority of a ProScript implementation tends to be a series of pure function declarations. A small subset of these functions is exposed to a global namespace for access by the verified application while most remain hidden as utility functions for purposes such as key derivation, decryption and so on. This state-passing style is in contrast to DJS that allows direct modification of heap data structures. The functional style of ProScript allows protocol data structures, including objects and arrays, to be translated to simple terms in ProVerif built using constructors and destructors, hence avoiding the state-space explosion inherent in the heap-based approach that is needed to translate DJS to ProVerif [7].

4.1. ProScript Syntax

A ProScript implementation consists of a series of *modules*, each containing a sequence of type declarations (containing constructors, assertion utilities, and type converters), constant declarations and function declarations.

ProScript

$v ::=$	values
x	variables
n	numbers
s	strings
`true, false`	booleans
`undefined, null`	predefined constants
$e ::=$	expressions
v	values
$\{x_1 : v_1, \ldots, x_n : v_n\}$	object literals
$v.x$	field access
$[v_1, \ldots, v_n]$	array literals
$v[n]$	array access
`Lib.`$l(v_1, \ldots, v_n)$	library call
$f(v_1, \ldots, v_n)$	function call
$\sigma ::=$	statements
`var` $x; \sigma$	variable declaration
$x = e; \sigma$	variable assignment
`const` $x = e; \sigma$	constant declaration
`if` $(v_1 === v_2)$ $\{\sigma_1\}$ `else` $\{\sigma_2\}$	if-then-else
`return` e	return
$\gamma ::=$	globals
`const` $x = e$	constants
`const` $f =$ `function`$(x_1, \ldots, x_n)\{\sigma\}$	functions
`const Type_`$x = \{\ldots\}$	user types
$\mu ::= \gamma_0; \ldots; \gamma_n$	modules

Note that we will use the defined Lib.l notation to access the ProScript Cryptography Library.

Operational Semantics. ProScript's operational semantics is a subset of JavaScript, and both run on JavaScript interpreters. It is tooled based on the formal semantics of Maffeis et al. [19] and is superficially adapted for our language subset.

4.2. ProVerif Syntax

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. Tables store persistent state: The process `insert` $a(M_1, \ldots, M_n); P$ inserts the entry $(M_1, \ldots, M_n)$ in table a, and runs P. The process `get` $a(=M_1, x_2, \ldots, x_n)$ `in` P looks for an entry $(N_1, \ldots, N_n)$ in table a such that $N_1 = M_1$. When such an entry is found, it binds $x_2, \ldots, x_n$ to $N_2, \ldots, N_n$ respectively and runs P. 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.

ProVerif

$M ::=$	terms
v	values
a	names
$f(M_1, \ldots, M_n)$	function application
$E ::=$	enriched terms
M	return value
`new` $a : \tau; E$	new name a of type τ
`let` $x = M$ `in` E	variable definition
`if` $M = N$ `then` E_1 `else` E_2	if-then-else

$P, Q ::=$ processes
- 0 — null process
- $\texttt{in}(M, x : \tau); P$ — input x from channel M
- $\texttt{out}(M, N); P$ — output N on channel M
- $\texttt{let}\ x = M\ \texttt{in}\ P$ — variable definition
- $P \mid Q$ — parallel composition
- $!\,P$ — replication of P
- $\texttt{insert}\ a(M_1, \ldots, M_n); P$ — insert into table a
- $\texttt{get}\ a(=M_1, x_2, \ldots, x_n)\ \texttt{in}\ P$ — get table entry specified by M_1
- $\texttt{event}\ M; P$ — event M
- $\texttt{phase}\ n; P$ — enter phase n

$\Delta ::=$ declaration
- $\texttt{type}\ \tau$ — type τ
- $\texttt{free}\ a : \tau$ — name a
- $\texttt{query}\ q$ — query q
- $\texttt{table}\ a(\tau_1, \ldots, \tau_n)$ — table a
- $\texttt{fun}\ C(\tau_1, \ldots, \tau_n) : \tau$ — constructor
- $\texttt{reduc forall}\ x_1 : \tau_1, \ldots, x_n : \tau_n; f(M_1, \ldots, M_n) = M$ — destructor
- $\texttt{equation forall}\ x_1 : \tau_1, \ldots, x_n : \tau_n; M = M'$ — equation
- $\texttt{letfun}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) = E$ — pure function
- $\texttt{let}\ p(x_1 : \tau_1, \ldots, x_n : \tau_n) = P$ — process

$\Sigma ::= \Delta_1 \ldots \Delta_n.\texttt{process}\ P$ script

4.3. Translation

Within Σ, ProScript functions are translated into ProVerif pure functions. Type declarations are translated into ProVerif type declarations. Individual values, such as strings and numbers, are declared as global constants at the top-level scope of the ProVerif model with identifiers that are then employed throughout the model when appropriate. Objects and Arrays are instantiated in the model using functions, with destructors automatically generated in order to act as getters.

Translation Rules

$M_v ::= v \mid \{x_1 : v_1, \ldots, x_n : v_n\} \mid [v_1, \ldots, v_n]$

$\mathcal{V}[\![M_v]\!] \to M$ Values to Terms

$\mathcal{V}[\![v]\!] = v$

$\mathcal{V}[\![\{x_1 : v_1, \ldots, x_n : v_n\}]\!] = \texttt{Obj\_t}(v_1, \ldots, v_n)$

$\mathcal{V}[\![[v_1, \ldots, v_n]]\!] = \texttt{Arr\_t}(v_1, \ldots, v_n)$

$\mathcal{E}[\![e]\!] \to M$ Expressions to Terms

$\mathcal{E}[\![M_v]\!] = \mathcal{V}[\![M_v]\!]$

$\mathcal{E}[\![v.x]\!] = \texttt{get\_x}(v)$

$\mathcal{E}[\![v[i]]\!] = \texttt{get\_i}(v)$

$\mathcal{E}[\![\texttt{Lib}.l(v_1, \ldots, v_n)]\!] = \texttt{Lib\_l}(\mathcal{V}[\![v_1]\!], \ldots, \mathcal{V}[\![v_n]\!])$

$\mathcal{E}[\![f(v_1, \ldots, v_n)]\!] = f(\mathcal{V}[\![v_1]\!], \ldots, \mathcal{V}[\![v_n]\!])$

$\mathcal{S}[\![\sigma]\!] \to E$ Statements to Enriched Terms

$\mathcal{S}[\![\texttt{var}\ x; \sigma]\!] = \mathcal{S}[\![\sigma]\!]$

$\mathcal{S}[\![x = e; \sigma]\!] = \texttt{let}\ x = \mathcal{E}[\![e]\!]\ \texttt{in}\ \mathcal{S}[\![\sigma]\!]$

$\mathcal{S}[\![\texttt{const}\ x = e; \sigma]\!] = \texttt{let}\ x = \mathcal{E}[\![e]\!]\ \texttt{in}\ \mathcal{S}[\![\sigma]\!]$

$\mathcal{S}[\![\texttt{return}\ v]\!] = \mathcal{V}[\![v]\!]$

$\mathcal{S}[\![\texttt{if}\ (v_1 === v_2)\ \{\sigma_1\}\ \texttt{else}\ \{\sigma_2\}]\!] =$
$\quad \texttt{if}\ \mathcal{V}[\![v_1]\!] = \mathcal{V}[\![v_2]\!]\ \texttt{then}\ \mathcal{S}[\![\sigma_1]\!]\ \texttt{else}\ \mathcal{S}[\![\sigma_2]\!]$

$\mathcal{F}[\![\gamma]\!] \to \Delta$ Types and Functions to Declarations

$\mathcal{F}[\![\texttt{const}\ f = \texttt{function}(x_1, \ldots, x_n)\{\sigma\}]\!] =$
$\quad \texttt{letfun}\ f(x_1, \ldots, x_n) = \mathcal{S}[\![\sigma]\!]$

$\mathcal{F}[\![\texttt{const Type\_}t = \{\ldots\}]\!] = \texttt{type}\ t$

$\mathcal{C}[\![\mu]\!](P) \to P$ Constants to Top-level Process

$\mathcal{C}[\![\epsilon]\!](P) = P$

$\mathcal{C}[\![\texttt{const}\ x = e; \mu]\!](P) = \texttt{let}\ x = \mathcal{E}[\![e]\!]\ \texttt{in}\ \mathcal{C}[\![\mu]\!](P)$

$\mathcal{M}[\![\mu]\!](P) \to \Sigma$ Modules to Scripts

$\mathcal{M}[\![\mu]\!](P) = \mathcal{F}[\![\gamma_1]\!] \ldots . \mathcal{F}[\![\gamma_n]\!] . \mathcal{C}[\![\mu_c]\!](P)$
where μ_c contains all globals $\texttt{const}\ x = e$ in μ
and $\gamma_1, \ldots, \gamma_n$ are the other globals of μ.

Translation Soundness. We currently do not formally prove translation soundness, so proofs of the resulting ProVerif model do not necessarily imply proof of the source code. Instead, we use model translation as a pragmatic tool to automatically generate readable protocol models faithful to the implementation, and to find bugs in the implementation. We have experimented with multiple protocols written in ProScript, including OTR, SP, and TLS 1.3, and by carefully inspecting the source code and target models, we find that the compiler is quite reliable and that it generates models that are not so far from what one would want to write directly in ProVerif. In future work, we plan to prove the soundness of this translation to get stronger positive guarantees from the verification. To this end, we observe that our source language is a simply-typed functional programming language, and hence we should be able to closely follow the methodology of [20].

Generating Top-Level Processes. We are also able to automatically generate top-level ProVerif processes. Aiming to implement this in a way that allows us to easily integrate ProScript code into existing codebases, we decided to describe top-level functions inside `module.exports`, the export namespace used by modules for Node.js [21], a popular client/server run-time for JavaScript applications (based on the V8 engine). This makes intuitive sense: `module.exports` is used specifically in order to define the functions of a Node.js module that should be available to the external namespace once that module is loaded, and executing all this functionality in parallel can give us a reasonable model of a potential attacker process. Therefore, functions declared in this namespace will be translated into top-level processes executed in parallel. We use ProVerif `tables` in order to manage persistent state between these parallel processes: each process fetches the current state from a table, runs a top-level function in that state, and stores the updated state returned by the function in the table.

4.4. Trusted Libraries for ProScript

Protocol implementations in ProScript rely on a few trusted libraries, in particular, for cryptographic primitives and for encoding and decoding protocol messages.

When deployed in Node.js or within a browser, the protocol code may have access to native cryptographic APIs. However, these APIs do not typically provide all modern cryptographic primitives; for example, the W3C Web Cryptography API does not support Curve25519, which is needed in Signal. Consequently, implementations like Signal Messenger end up compiling cryptographic primitives from C to JavaScript. Even if the desired primitives were available in the underlying platform, accessing them in a hostile environment is unsafe, since an attacker may have redefined them. Consequently, we developed our own libraries for cryptography and message encoding.

The ProScript Cryptography Library (PSCL) is a trusted cryptographic library implementing a variety of modern cryptographic primitives such as X25519, AES-CCM and BLAKE2. All of its primitives are fully type-checked without this affecting speed: in the majority of our benchmarks, PSCL is as fast as or faster than popular JavaScript cryptographic libraries like SJCL and MSR JavaScript Crypto, which do not even benefit from defensive type checking

More crucially, PSCL functions used in ProScript code are detected by the ProScript compiler as it produces the applied pi model of the implementation, giving it the ability to convert each call to a cryptographic primitive to a call to the corresponding symbolic function in ProVerif. For example, if the ProScript compiler sees a call to PSCL's X25519 implementation, it will automatically translate it to a standard Diffie-Hellman construction in ProVerif.

5. Implementing and Verifying SP

We describe SP, a variant of Signal Protocol that closely follows TextSecure version 3. We show how we implement and verify this protocol in our framework.

5.1. Protocol Overview

In SP, as illustrated in Figure 2, each client publishes a long-term Diffie-Hellman public key and a set of ephemeral Diffie-Hellman public keys (called *"pre-keys"*). These keys include both signed pre-keys, which can be reused for some period of time, and non-signed, one-time pre-keys, which are fresh at each session. To send a message to Bob, Alice retrieves Bob's long-term keys $(g^{b_{3dh}}, g^{b_{sig}})$, a signed pre-key g^{b_s} and a one-time pre-key g^{b_o}. She then chooses her own ephemeral g^{a_e}. A four-way Diffie-Hellman handshake is accomplished using Alice and Bob's long-term identity keys and their short-term ephemeral keys in order to derive the session secret S. The one-time pre-key is optional: when there remains no available one-time pre-key, the exchange is performed with a triple Diffie-Hellman handshake. An encryption key, k_{enc}, is then derived from S by Hash-Based Key Derivation (HKDF) [22] and the message M_0 is sent encrypted under the authenticated encryption scheme AES-GCM, with public and ephemeral keys as associated data: $\mathsf{ENC}(k, m, ad)$ means that m is encrypted with k and both the message m and the associated data ad are authenticated. Subsequent messages in the conversation obtain authentication by chaining to S via a forward-ratcheting construction that also employs HKDF. Each sent message includes its own newly generated ephemeral public key and the protocol's double ratchet key refresh mechanism manages the key state by advancing key chaining with every message.

SP's forward and future secrecy goals are intended to make it so that the compromise of Alice or Bob's long-term keys allows for their impersonation but not for the decryption of their messages. The use of a signed initial ephemeral pre-key results in weaker forward secrecy guarantees for the first flight of messages from A to B: no forward secrecy is provided if both the long-term keys and pre-keys are leaked, although the guarantees for subsequent flights remain strong. If pre-keys are not signed, then the protocol only offers weak forward secrecy with respect to long-term key leakage. We note that the term "forward secrecy" can be confusing in a protocol like Signal, because a number of keys are at stake: long-term keys $((a_{3dh}, a_{sig}), (b_{3dh}, b_{sig}))$, signed pre-key b_s, one-time pre-key b_o, ephemeral keys $(a_e, a_{e'}, b_{e'})$, root keys $(rk_{ab}, ck_{ab}, rk_{ba}, ck_{ba})$ and message keys (k_{enc}). Any formal analysis of the protocol must precisely state which of these keys can be compromised and when.

Differences from other versions of Signal. An earlier version of the protocol, TextSecure Version 2, was cryptographically analyzed in previous work [2]. There are two key differences between SP and TextSecure Version 2.

Signed, Time-Based Pre-Keys. Version 2 uses a triple Diffie-Hellman handshake with one of a hundred pre-keys that Bob stores on the server (including a "last-resort" pre-key). TextSecure Version 3 and all subsequent versions of Signal, including SP, use a signed time-based pre-key, used in conjunction with an unsigned one-time pre-key in case it is available. Bob periodically replaces his signed pre-key (for example, once every week), which may be re-used until its replacement and refreshes his collection of unsigned one-time pre-keys. In SP, when one-time pre-keys are exhausted, no "last-resort" pre-key is used.

Stronger Identity Protection. Since Version 3, tag_n is expanded to include the long-term identities of the sender and recipient, which is not the case in Version 2. This provides a slightly stronger authentication guarantee in the rare case that the encryption keys for different pairs of users turns out to be the same.

In addition to these differences with Version 2, SP also differs from other variants of Signal in one key aspect. In SP, long-term identities are split into one Diffie-Hellman key pair and one signing key pair. In Signal, the same key pair is used for both operations, by applying an elliptic curve conversion from the Montgomery curve-based Diffie-Hellman key pair to obtain its twisted Edwards curve-based

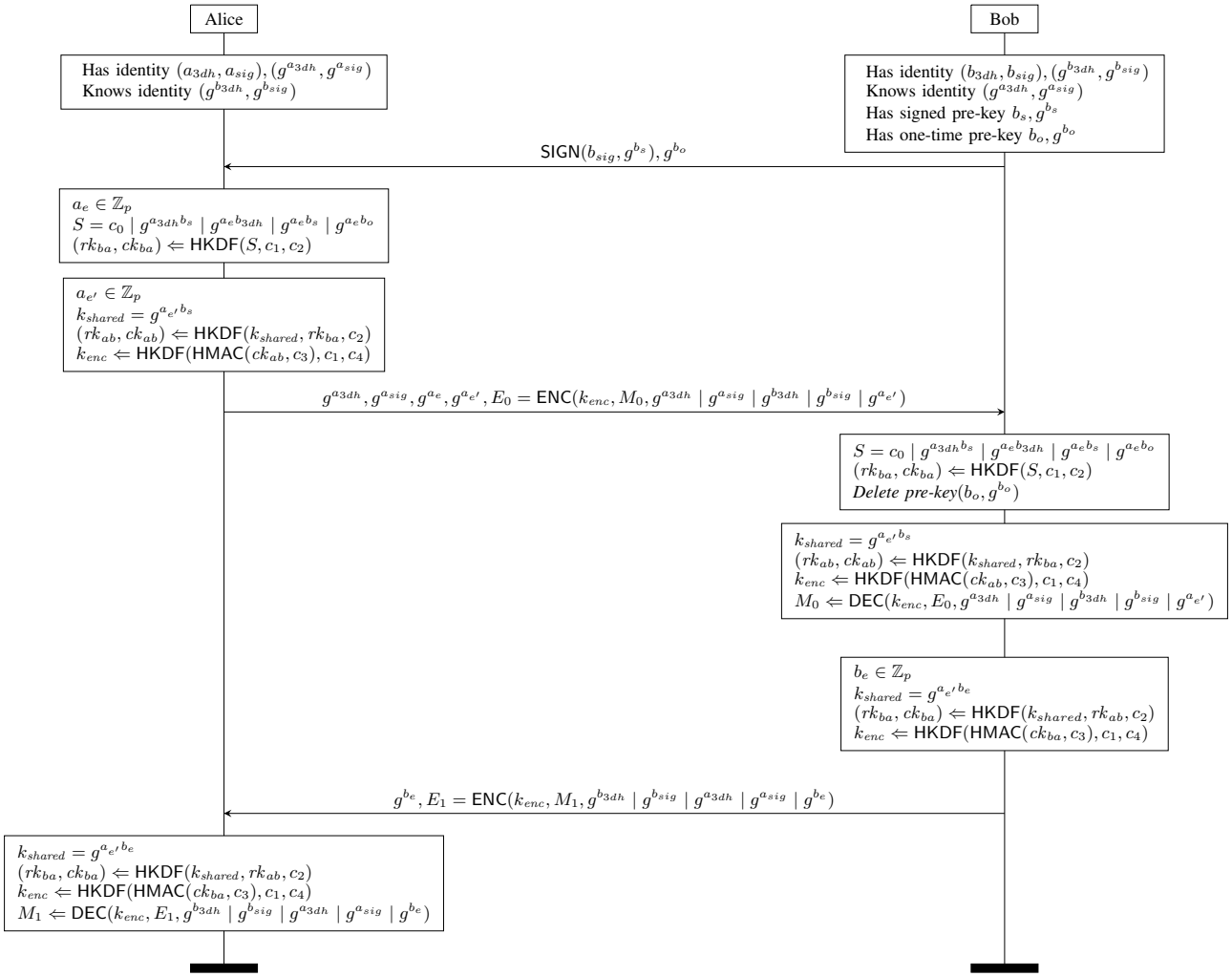

Figure 2: SP, a variant of Signal with minor differences. Alice requests a signed pre-key from Bob (via the server) and sends an initial message M_0. Bob accomplishes his side of the key exchange and obtains M_0. Bob later sends his reply M_1, illustrating the Axolotl ratchet post-AKE. We ignore the hash-based ratchet that occurs when two consecutive messages are sent in the same direction. c_i refers to various constants found throughout the protocol.

signing key pair equivalent. We choose to use separate keys instead, because in our cryptographic proof, we do not want to add a non-standard cryptographic assumption about the use of the same key in two independent cryptographic operations. In exchange for using standard cryptographic assumptions, we consider the cost of adding an extra 32 byte key to the protocol to be acceptable.

5.2. Protocol Implementation

To implement SP in ProScript, we must first deconstruct it into various protocol components: structures for managing keys and user states, messaging functions, APIs and top-level processes. ProScript is well-equipped to handle these protocol components in a way that lends itself to model extraction and verification. We break down our ProScript SP implementation into:

Types for State and Key Management. ProScript's type declaration syntax can be used to declare types for individual elements such as encryption keys but also for collections of elements such as a conversation party's state. These declarations allow for the construction of common data structures used in the protocol and also makes their management and modification easier in the extracted ProVerif models.

Messaging Interface. The ProScript implementation exposes the generic messaging API in a single global object. All interface access provides purely state-passing functionality.

Goals	Messages	Parties	Roles	Time
Secrecy	1	A, B	One	00h.04m.07s.
Secrecy	1	A, B	Two	00h.11m.17s.
Indist.	1	A, B	One	02h.06m.15s.
Authen.	1	A, B, M	One	00h.58m.19s.
Authen.	1	A, B, M	Two	29h.17m.39s.
Fo. Se.	1	A, B	One	00h.04m.14s.
KCI	1	A, B	One	00h.19m.20s.

Figure 3: Verification times for SP ProVerif models.

Long-Term and Session States. Protocol functions take long-term and session states (S^a, T_n^{ab}) as input and return T_{n+1}^{ab}. S^a contains long-term values such identity keys, while T includes more session-dependent values such as `ephemerals`, containing the current ephemeral and chaining keys for the session context and `status`, indicating whether the application layer should perform a state update.

Internal Functions. Utility functionality, such as key derivation, can also be described as a series of pure functions that are not included in the globally accessible interface.

Top-Level Process. A top-level process can serve as a harness for testing the proper functioning of the protocol in the application layer. Afterwards, when this top-level process is described in the extracted ProVerif model, the implementer will be able to use it to define which events and security properties to query for.

Inferred Types in ProScript. ProScript type declarations allow for the easier maintenance of a type-checkable protocol implementation, while also allowing the ProScript compiler to translate declared types into the extracted ProVerif model. Defining a `key` as an array of 32 bytes will allow the ProScript compiler to detect all 32 byte arrays in the implementation as keys and type their usage accordingly.

5.3. Protocol Verification

We use ProVerif to verify the security goals of our extracted model by defining defining queries that accurately test the resilience of security properties against an active adversary. Under an active Dolev-Yao adversary, ProVerif was able to verify confidentiality, authenticity, forward secrecy and future secrecy for Alice and Bob initializing a session and exchanging two secret messages, with a compromised participant, Mallory, also being allowed to initialize sessions and exchange non-secret messages with Alice and Bob. Our analysis revealed two novel attacks: a key compromise impersonation attack and a replay attack, for which we propose a fix. Aside from these attacks, we were also able to model the previously documented Unknown Keyshare Attack [2].

Extracts of our compiled SP implementation are available online [23]. Models begin with type declarations followed by public constant declarations, equational relationships for cryptographic primitives, protocol functions, queries and relevant names and finally the top-level process with its associated queries.

The top-level process queries for security properties such as confidentiality, authenticity and forward secrecy between two roles: an initiator (e.g. Alice) who sends an initial message and thereby initializes an authenticated key exchange, and a responder (e.g. Bob) who receives the message and who may send a response. Some models include a third compromised identity, Mallory, who also communicates with Alice and Bob but while leaking her private keys to the attacker beforehand. In some instances, we also model parallel process executions where each identity (Alice, Bob and optionally Mallory) assumes both the role of the initiator and the responder. We informally call this latter scenario a "two-way role" model.

Secrecy and Indistinguishability. For every message considered in our protocol model, we define a secret constant M_n where M_1 is the initial message in a session. These secret values are then used as the plaintext for the encrypted messages sent by the principals. We show that an active attacker cannot retrieve a message's plaintext M_n using the query:

$$\mathsf{query}(\mathsf{attacker}(M_n)) \tag{1}$$

Similarly, we show indistinguishability using the query $\mathsf{query}(\mathsf{noninterf}(M_n))$.

Forward and Future Secrecy. We examine forward and future secrecy in Signal Protocol in multiple scenarios: the compromise of long-term keys and the compromise of message keys in two different types of message flights. In these scenarios, we need to model that keys are leaked after sending or receiving certain messages. We rely on ProVerif *phases* for that: intuitively, t represents a global clock, and processes occurring after the declaration of a phase t are active only during this phase.

We show that message M_1 remains secret by query (1) even if the long-term keys $(a_{3dh}, a_{sig}, b_{3dh}, b_{sig})$ are leaked after sending M_1. Furthermore, we can modify our ProVerif model to produce a sanity check: if responder Bob skips the signature check on g^{a_s}, ProVerif shows that an active attacker becomes capable of violating this forward secrecy property.

Next, we examine two different messaging patterns in the Double Ratchet algorithm and find that they approach forward and future secrecy differently:

- **Single-Flight Pattern.** In this scenario, Alice sends Bob a number of messages M_n and M_{n+1} where $n > 1$ and does not receive a response. In this scenario, Bob's lack of response does not allow Alice to obtain a fresh ephemeral key share g^{b_e} required to establish a new k_{shared} in T_{n+1}^{ab} to be used for M_{n+1}, so Alice just updates the key ck_{ab} by hashing it. If Alice's session state T_{n+1}^{ab}, (which, recall, contains ${a_e}^{n+1}$ and (rk_{ab}, ck_{ab}) for M_{n+1}), is leaked, then M_n remains secret (forward secrecy). Obviously, to take advantage of this property in case of compromise,

the keys (rk_{ab}, ck_{ab}) for M_n must have been appropriately deleted, which is delicate when messages are received out-of-order: if $M_{n_1}, \ldots, M_{n_k}$ $(n_1 < \ldots < n_k)$ have been received, the receiver should keep the chaining key ck_{ab} for M_{n_k+1} and the encryption keys k_{enc} for the messages M_i not received yet with $i < n_k$. If T_n^{ab} is leaked, then M_{n+1} is not secret, so no future secrecy is obtained.

- **Message-Response Pattern.** In this scenario, Alice sends Bob a single message M_n where $n > 1$ and receives a response M_{n+1} before sending M_{n+2}. Upon receiving M_{n+1}, Alice will be able to derive a fresh $k_{shared} = g^{{a_e}^{n+2} {b_e}^{n+1}}$. As a result, if T_{n+2}^{ab} is leaked, then M_n remains secret (forward secrecy) and if T_n^{ab} is leaked after M_{n+1} is received, then M_{n+2} remains secret (future secrecy).

Message Authenticity. Signal Protocol relies on a Trust-on-First-Use (TOFU) authentication model: Alice assumes that Bob's advertised identity key is authenticated and untampered with and employs it as such until an event causes the trust of the key to be put in question, such as a sudden identity key change or an out of band verification failure. We model TOFU by embedding Alice and Bob's identity keys into each other's initial states. We are then free to model for message authenticity: informally, if B receives a message M from A, we want A to have sent M to B. In ProVerif, we can specify two events: $\mathsf{Send}(A, B, M)$, which means that A sends M to B and $\mathsf{Recv}(A, B, M)$, which means that B receives M from A. We can then formalize the correspondence

$$\mathsf{event}(\mathsf{Recv}(A, B, M)) \Longrightarrow \mathsf{event}(\mathsf{Send}(A, B, M)) \quad (2)$$

which checks if for all $\mathsf{Recv}(A, B, M)$ events, it must be the case that a $\mathsf{Send}(A, B, M)$ event has also been executed.

ProVerif succeeds in proving correspondence (2) using public keys A and B. While this implies the desired property when the relation between the public keys and the identity of the principals is bijective, a limitation of this approach is that the identities of the principals are only expressed in terms of keys and not as a more personally-linked element, such as for example a phone number. Therefore, we cannot formally express stronger identity binding as part of the protocol model. This point leads to the Unknown Key Share Attack first reported for Signal Protocol Version 2 [2]: if an adversary can register the public keys $(g^{b_{3dh}}, g^{b_{sig}})$ of B as public keys of C and A sends a message to C, then C can forward this message to B and B will accept it as coming from A, since B and C have the same public keys.

No Replays. This property is similar to message authenticity, but uses an injective correspondence instead, which means that each execution of $\mathsf{Recv}(A, B, M)$ corresponds to a distinct execution of $\mathsf{Send}(A, B, M)$:

$$\mathsf{inj\text{-}event}(\mathsf{Recv}(A, B, M)) \Longrightarrow \mathsf{inj\text{-}event}(\mathsf{Send}(A, B, M))$$

When a optional one-time pre-key is involved in the initial session handshake, ProVerif shows that the injective correspondence holds for the first message in the conversation. However, when this optional one-time pre-key is not used, a replay attack is detected. Signal Protocol Version 3 will accept a Diffie-Hellman handshake that only employs identity keys and signed pre-keys, both of which are allowed to be reused across sessions. This reuse is what makes a replay attack possible. We propose a fix for this issue by having clients keep a cache of the ephemeral keys used by the sender of received messages, associated with that sender's identity key. We are able to expand our event queries in ProVerif to account for this fix by showing the non-injective correspondence of the Send and Recv events with added ephemeral keys. Coupled with a caching of ephemeral keys, we can ensure that the Recv event is only executed once per ephemeral key. Hence, the injective correspondence is implied by the non-injective correspondence.

Key Compromise Impersonation (KCI). We present a novel key compromise impersonation attack: to detect KCI, we consider a scenario in which Alice or Bob's keys are compromised and test again for authenticity of messages received by the compromised principal. When Alice or Bob's long-term secret key is compromised, ProVerif shows that message authenticity still holds. However, when Bob's signed pre-key is also compromised, ProVerif finds an attack against message authenticity. This is a novel key compromise impersonation attack: when the adversary has Bob's signed pre-key s, he can choose x and x' and compute the session keys using Alice's and Bob' public keys $(g^{a_{3dh}}, g^{a_{sig}})$ and $(g^{b_{3dh}}, g^{b_{sig}})$ and Bob's one time pre-key g^o and send his own message in Alice's name. This message is accepted by Bob as if it came from Alice: the event $\mathsf{Recv}(A, B, M)$ is executed without having executed $\mathsf{Send}(A, B, M)$.

Integrating Symbolic Verification into the Development Cycle. Human-readability of the automatically compiled ProVerif model is key to our verification methodology. In the case of a query failure, users can opt to modify their implementation and recompile into a new model, or they can immediately modify the model itself and re-test for security queries within reasonable model verification times. For example, if an implementer wants to test the robustness of a passing forward secrecy query, they can disable the signature verification of signed pre-keys by changing a single line in the model, causing the client to accept any pre-key signature.

6. Cryptographic Proofs with CryptoVerif

To complement the results obtained in the symbolic model using ProVerif, we use the tool CryptoVerif [5] in order to obtain security proofs in the computational model. This model is much more realistic: messages are bitstrings; cryptographic primitives are functions from bitstrings to bitstrings; the adversary is a probabilistic Turing machine. CryptoVerif generates proofs by sequences of games [24], [25], like those written manually by cryptographers, automatically or with guidance of the user.

The computational model is more realistic, but it also makes it more difficult to mechanize proofs. For this reason,

CryptoVerif is less flexible and more difficult to use than ProVerif, and our results in the computational model are more limited. We model only one message of the protocol (in addition to the pre-keys), so we do not prove properties of the ratcheting algorithm. Considering several data messages exceeds the current capabilities of CryptoVerif—the games become too big.

Rather than directly using models generated from our ProScript code, we manually rewrite the input scripts of CryptoVerif, for two main reasons:

- The syntax of the protocol language of CryptoVerif differs slightly from that of ProVerif. We plan to overcome this difficulty in the future by modifying the syntax of CryptoVerif so that it is compatible with ProVerif.
- The kinds of models that are easy to verify using CryptoVerif differ from those that are easy for ProVerif; therefore, even if the source syntax were the same, we would still need to adapt our compiler to generate specialized models that would be more conducive to CryptoVerif's game-based proofs.

6.1. Assumptions

We make the following assumptions on the cryptographic primitives:

- The elliptic curve Ec25519 satisfies the gap Diffie-Hellman (GDH) assumption [26]. This assumption means that given g, g^a, and g^b for random a, b, the adversary has a negligible probability to compute g^{ab} (computational Diffie-Hellman assumption), even when the adversary has access to a decisional Diffie-Hellman oracle, which tells him given G, X, Y, Z whether there exist x, y such that $X = G^x$, $Y = G^y$, and $Z = G^{xy}$. When we consider sessions between a participant and himself, we need the square gap Diffie-Hellman variant, which additionally says that given g and g^a for random a, the adversary has a negligible probability to compute g^{a^2}. This assumption is equivalent to the GDH assumption when the group has prime order [27], which is true for Ec25519 [28]. We also added that $x^y = x'^y$ implies $x = x'$ and that $x^y = x^{y'}$ implies $y = y'$, which hold when the considered Diffie-Hellman group is of prime order.
- Ed25519 signatures, used for signing pre-keys, are unforgeable under chosen-message attacks (UF-CMA) [29].
- The functions

$$x_1, x_2, x_3, x_4 \mapsto \mathsf{HKDF}(x_1 \| x_2 \| x_3 \| x_4, c_1, c_2)$$
$$x_1, x_2, x_3 \mapsto \mathsf{HKDF}(x_1 \| x_2 \| x_3, c_1, c_2)$$
$$x, y \mapsto \mathsf{HKDF}(x, y, c_2)$$
$$x \mapsto \mathsf{HKDF}(x, c_1, c_4)$$

  are independent random oracles, where x, y, x_1, x_2, x_3, x_4, and c_1 are 256-bit long. We further justify this assumption in the full version of this work [23]: there, we show that these functions are indifferentiable [30] from independent random oracles, assuming that the compression function underlying SHA256 is a random oracle. (The considered HKDF function [22] is defined from HMAC-SHA256, which is itself defined from SHA256.)
- HMAC-SHA256 is a pseudo-random function (PRF) [31]. This assumption is used for $\mathsf{HMAC}(ck_{ab}, \cdot)$ and $\mathsf{HMAC}(ck_{ba}, \cdot)$.
- The encryption scheme ENC, which is AES-GCM, is a secure authenticated encryption with associated data (AEAD). More precisely, it is indistinguishable under chosen plaintext attacks (IND-CPA) and satisfies ciphertext integrity (INT-CTXT) [32], [33].

CryptoVerif provides a library that predefines the most common cryptographic assumptions, so that the user does not have to write them for each protocol. In our work, we had to adapt these predefined assumptions to our specific needs: the GDH assumption is predefined, but the square GDH variant is not; unary random oracles are predefined, but we also needed binary, ternary, and 4-ary ones; predefined PRFs, SUF-CMA MACs, and IND-CPA encryption schemes use a key generation function, while in our schemes the key is a plain random bitstring, without a key generation function. Adapting the definition of primitives did not present any major difficulty. As mentioned in § 5.1, we had to make one modification to the original Signal Protocol, for it to be provable in the computational model: we use different keys for the elliptic curve Diffie-Hellman and elliptic curve signatures. It is well-known that using the same keys for several cryptographic primitives is undesirable, as proving security requires a joint security assumption on the two primitives in this case. Therefore, we assume each protocol participant to have two key pairs, one for Diffie-Hellman and one for signatures. This problem remains undetected in a symbolic analysis of the protocol.

6.2. Protocol Model

We model SP as a process in the input language of CryptoVerif, which is similar to the one of ProVerif. We consider simultaneously the protocol of Figure 2 and the version without the optional one-time pre-key b_o. As mentioned above, we consider only one message in each session. Our threat model includes an untrusted network, malicious principals, and long-term key compromise, as mentioned in § 2. It does not include session state compromise, which is less useful with a single message.

At a high level, we use the same messaging API as in § 2. However, to make verification easier for CryptoVerif, we specify a lower-level interface. We consider two honest principals Alice and Bob, and define separate processes for Alice interacting with Bob, with herself, or with a malicious participant, Bob interacting with Alice, and Bob interacting with himself or a malicious participant, as well as similar processes with the roles of Alice and Bob reversed. The adversary can then implement the high-level interface of § 2 from this lower-level interface: the adversary is supposed to implement the malicious principals (including defining keys for them) and to call the low-level interface processes to run sessions that involve the honest principals Alice and Bob.

We make two separate proofs: In the first one, we prove the security properties for sessions in which Bob generates pre-keys and runs the protocol with Alice. (Other protocol sessions exist in parallel as described above; we do not prove security properties for them. For sessions for which we do not prove security properties, we give to the adversary the ephemeral a'_e and the key rk_{ba} or rk_{ba} and let the adversary encrypt and MAC the message himself, to reduce the size of our processes.) In the second one, we prove the security properties for sessions in which Alice generates pre-keys and runs the protocol with herself. Bob is included in the adversary in this proof. The security for sessions in which Alice generates pre-keys and runs the protocol with Bob follows from the first proof by symmetry. The security for sessions in which Bob generates pre-keys and runs the protocol with himself follows from the second proof. The other sessions do not satisfy security properties since they involve the adversary. (They must still be modeled, as they could break the protocol if it were badly designed.) Therefore, these two proofs provide all desired security properties.

6.3. Security Goals

We consider the following security goals from § 2:

Message Authenticity, No Replays, and Key Compromise Impersonation (KCI). These properties are modeled by correspondences as in ProVerif (§ 5.3). For key compromise impersonation, we consider the compromise of the long-term Diffie-Hellman and signature keys of Bob, and prove again message authenticity. We do not consider the compromise of the signed pre-key since we already know from the symbolic analysis that there is an attack in this case.

Computational Indistinguishability. If A randomly chooses between two messages M_0, M_1 of the same length and sends one of them to B, then the adversary has a negligible probability of guessing which of the two messages was sent. In our model, this is formalized by choosing a random bit $secb \in \{0, 1\}$; then A sends message M_b to B, and we show that the bit $secb$ remains secret, with the query **secret** $secb$.

Forward Secrecy. This is proved exactly like indistinguishability, but with an additional oracle that allows the adversary to obtain the secret keys of the principals, thus compromising them.

We do not consider future secrecy since we have a single message. We do not consider secrecy since we directly deal with the stronger property of indistinguishability.

6.4. Results

CryptoVerif proves message authenticity, absence of key compromise impersonation attacks (when the long-term keys of Bob are compromised), indistinguishability, and forward secrecy, but cannot prove absence of replays. This is due to the replay attack mentioned in § 5.3. Since this attack appears only when the optional one-time pre-key is omitted, we separate our property into two: we use

Goals	Parties	Running Time
Forward Secrecy	A, B, M	3 min. 58 sec.
Forward Secrecy	A, M	7 min. 04 sec.
KCI	A, B, M	3 min. 15 sec.
Others	A, B, M	4 min. 15 sec.
Others	A, M	3 min. 35 sec.

Figure 4: Verification times for SP CryptoVerif models, without anti-replay countermeasure. The runtimes with the anti-replay countermeasure are of the same order of magnitude. Tested using CryptoVerif 1.24.

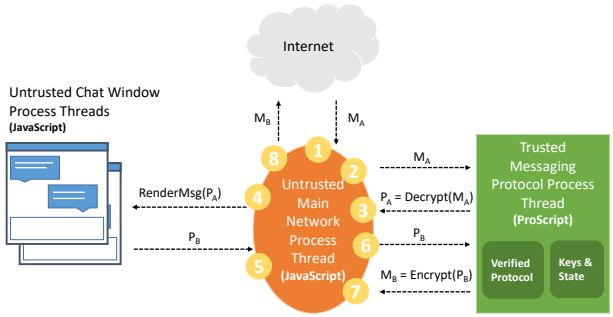

Figure 5: Cryptocat Architecture: isolating verified and untrusted components in Electron apps within separate processes.

events $\mathsf{Send}(A, B, M)$ and $\mathsf{Recv}(A, B, M)$ for the protocol with optional pre-key and events $\mathsf{Send3}(A, B, M)$ and $\mathsf{Recv3}(A, B, M)$ for the protocol without optional pre-key. CryptoVerif then proves

$$\mathsf{inj\text{-}event}(\mathsf{Recv}(A, B, M)) \Longrightarrow \mathsf{inj\text{-}event}(\mathsf{Send}(A, B, M))$$
$$\mathsf{event}(\mathsf{Recv3}(A, B, M)) \Longrightarrow \mathsf{event}(\mathsf{Send3}(A, B, M))$$

which proves message authenticity and no replays when the one-time pre-key is present and only message authenticity when it is absent. This is the strongest we can hope for the protocol without anti-replay countermeasure.

With our anti-replay countermeasure (§5.3), CryptoVerif can prove the absence of replays, thanks to a recent extension that allows CryptoVerif to take into account the replay cache in the proof of injective correspondences, implemented in CryptoVerif version 1.24. Our CryptoVerif proofs have been obtained with some manual guidance: we indicated the main security assumptions to apply, instructed CryptoVerif to simplify the games or to replace some variables with their values, to make terms such as $m^a = m^b$ appear. The proofs were similar for all properties.

7. A Verified Protocol Core for Cryptocat

We now describe how we can rewrite Cryptocat to incorporate our ProScript implementation of SP. We deconstruct the Cryptocat JavaScript code into the following components, as advocated in Figure 1.

1) **Unverified JavaScript Application** This component, which comprises the majority of the code, manages the user's state, settings, notifications, graphical interface and so on. It is connected to the protocol only via the ability to call exposed protocol functions (as documented in § 2.1). We adopt certain assumptions regarding the unverified JavaScript application, for example that it will not modify the protocol state outside of passing it through the protocol implementation interface.
2) **Verified Protocol Implementation** This component is written in ProScript and resides in a separate namespace, functioning in a purely state-passing fashion. Namely, it does not store any internal state or make direct network calls. This implementation is type-checked, and automatically verified every time it is modified.
3) **Trusted Library** This component provides cryptographic functionality. A goal is to include modern cryptographic primitives (X25519, AES-CCM) and provide type-checking assurances without affecting speed or performance.

This layered architecture is essential for our verification methodology, but is quite different from other messaging applications. For example, the Signal Desktop application is a Chrome browser application also written in JavaScript [34]. Parts of the protocol library are compiled from C using Emscripten, presumably for performance, parts are taken from third-party libraries, and other protocol-specific code is written in JavaScript. The resulting code (1.5MB, 39Kloc) is quite hard to separate into components, let alone verify for security. We hope that our layered approach can lead to verified security guarantees without sacrificing performance or maintainability.

7.1. Isolating Verified Code

We build Cryptocat using Electron [35], a framework for JavaScript desktop applications. Electron is built on top of the Node.js JavaScript runtime and the Chromium web renderer. By default, Electron allows applications to load any Node.js low-level module, which can in turn perform dangerous operations like accessing the file system and exfiltrate data over the network. Since all Node.js modules in a single process run within the same JavaScript environment, malicious or buggy modules can tamper with other modules via prototype poisoning or other known JavaScript attack vectors. Consequently, all Electron apps, including other desktop Signal implementations like WhatsApp and Signal messenger effectively include all of Electron and Node.js into their trusted computing base (TCB).

We propose a two-pronged approach to reduce this TCB.

Language-Based Isolation. Since ProScript is a subset of Defensive JavaScript, ProScript protocol code is isolated at the language level from other JavaScript code running within the same process, even if this code uses dangerous JavaScript features such as prototype access and modification. To ensure this isolation, ProScript code must not call any external (untyped) libraries.

Process Thread Isolation. We exploit features of Electron in order to isolate components of our application in different CPU threads as seen in Figure 5. When a message arrives on the network (1), the main network thread can only communicate with our Protocol TCB using a restrictive inter-process communication API. The TCB then uses its internal verified protocol functionality and state management to return a decryption of the message (3), which is then forwarded again via IPC to the chat window (4), a third separate CPU thread which handles message rendering. Furthermore, the TCB process is disallowed from loading any Node.js modules.

In particular, the network process is isolated from the chat media rendering process; neither ever obtain access to the key state or protocol functionality, which are all isolated in the ProScript protocol process. When Bob responds, a similar IPC chain of calls occurs in order to send his reply back to Alice (5, 6, 7, 8). Even if an error in the rendering code or in the XML parser escalated into a remote takeover of the entire web renderer, the calls to the protocol TCB would be restricted to those exposed by the IPC API. However, these isolation techniques only protect the ProScript code within our application when executed within a correct runtime framework. None of these techniques can guard against bugs in V8, Node.js, or Electron, or against malicious or buggy Node.js or Electron modules loaded by the application.

7.2. Performance and Limitations

Although we have verified the core protocol code in Cryptocat and tried to isolate this code from unverified code, the following limitations still apply: we have not formally verified the soundness of the cryptographic primitives themselves, although writing them in Defensive JavaScript does provide type safety. We have also not formally verified the Electron framework's isolation code. Similarly, we do not claim any formal verification results on the V8 JavaScript runtime or on the Node.js runtime. Therefore, we rely on a number of basic assumptions regarding the soundness of these underlying components. Cryptocat's successful deployment provides a general guideline for building, formally verifying, and isolating cryptographic protocol logic from the rest of the desktop runtime. Designing better methods for implementing and isolating security-critical components within Electron apps with a minimal TCB remains an open problem.

8. Related Work

Extracting Protocol Models from Running Code. There have been previous attempts [36] to extract ProVerif models from typed JavaScript, such as *DJS2PV* [7]. However, DJS2PV was only tested on small code examples: attempting to translate a complete implementation such as Signal Protocol resulted in a 3,800 line model that attempts to precisely account for the heap, but could not verify due to an exploding state space. Previous efforts such as FS2PV [20]

avoided this problem by choosing a purely functional source language that translated to simpler pi calculus scripts. We adopt their approach in ProScript to generate briefer, more readable models.

Type Systems for JavaScript. TypeScript [37], Flow [38], Defensive JavaScript and TS* [39] all define type systems that can improve the security of JavaScript programs. The type system in ProScript primarily serves to isolate protocol code from untrusted application and to identify a subset of JavaScript that can be translated to verifiable models.

Formal Analysis of Web Security Protocols. Tools like WebSpi [40] and AuthScan [41] have been used to verify the security of web security protocols such as OAuth. An expressive web security model has also been used to build manual proofs for cryptographic web protocols such as BrowserID [42]. These works are orthogonal to ProScript and their ideas can potentially be used to improve our target ProVerif models.

Analysis of Secure Messaging Protocols. Unger et al. survey previous work on secure messaging [10]. We discuss three recent closely-related works here.

Future secrecy was formalized by Cohn-Gordon et al. as "post-compromise security" [43]. Our symbolic formulation is slightly different since it relies on the definition of protocol phases in ProVerif.

Cryptographic security theorems and potential unknown key-share attacks on TextSecure Version 2 were presented by Frosch et al. [2]. In comparison to that work, our analysis covers a variant of TextSecure Version 3, our analysis is fully mechanized, and we address implementation details. Our CryptoVerif model only covers a single message, but we consider the whole protocol at once, while they prove pieces of the protocol separately. Like we do, they consider that HKDF is a random oracle. We further justify this assumption by an indifferentiability proof.

More recently and in parallel with this work, Cohn-Gordon et al. [44] prove, by hand, that the message encryption keys of Signal are secret in the computational model, in a rich compromise scenario, under assumptions similar to ours. Thereby, they provide a detailed proof of the properties of the double ratcheting mechanism. However, they do not model the signatures of the signed pre-keys, and they do not consider key compromise impersonation attacks or replay attacks or other implementation-level details. In contrast to their work, our computational proof is mechanized, but limited to only one message.

9. Conclusion and Future Work

Drawing from existing design trends in modern cryptographic web application, we have presented a framework that supports the incremental development of custom cryptographic protocols hand-in-hand with formal security analysis. By leveraging state-of-the-art protocol verification tools and building new tools, we showed how many routine tasks can be automated, allowing the protocol designer to focus on the important task of analyzing her protocol for sophisticated security goals against powerful adversaries.

We plan to continue to develop and refine ProScript by evaluating how it is used by protocol designers, in the spirit of an open source project. All the code and models presented in this paper, and a full version of this paper are available online [23]. Proving the soundness of translation from ProScript to ProVerif, by relating the source JavaScript semantics to the applied pi calculus, remains future work. The process of transforming the compiled model to a verified CryptoVerif script remains a manual task, but we hope to automate this step further, based on new and upcoming developments in CryptoVerif.

Finally, a word of caution: a protocol written in ProScript and verified with ProVerif or CryptoVerif does not immediately benefit from assurance against all possible attacks. Programming in ProScript imposes a strict discipline by requiring defensive self-contained code that is statically typed and can be translated to a verifiable model and subsequent verification can be used to eliminate certain well-defined classes of attacks. We believe these checks can add confidence to the correctness of a web application, but they do not imply the absence of security bugs, since we still have a large trusted computing base. Consequently, improving the robustness and security guarantees of runtime frameworks such as Electron, Node.js, and Chromium, remains an important area of future research.

Acknowledgments. This work was funded by the following grants: ERC CIRCUS, EU NEXTLEAP, and ANR AJACS.

References

[1] K. Bhargavan, A. Lavaud, C. Fournet, A. Pironti, and P. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *IEEE Symposium on Security & Privacy (Oakland)*, 2014, pp. 98–113.

[2] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz, "How secure is TextSecure?" in *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[3] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016.

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

[5] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 193–207, 2008.

[6] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Language-based defenses against untrusted browser origins," in *USENIX Security Symposium*, 2013, pp. 653–670.

[7] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Defensive JavaScript - building and verifying secure web components," in *Foundations of Security Analysis and Design (FOSAD VII)*, 2013, pp. 88–123.

[8] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. London, United Kingdom: ACM Press, Jan. 2001, pp. 104–115.

[9] H. Krawczyk, "HMQV: A High-performance Secure Diffie-Hellman Protocol," in *International Conference on Advances in Cryptology (CRYPTO)*, ser. Lecture Notes in Computer Science, V. Shoup, Ed., vol. 3621. Springer, 2005, pp. 546–566.

[10] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith, "SoK: Secure Messaging," in *IEEE Symposium on Security & Privacy (Oakland)*, 2015.

[11] N. Durov, "Telegram MTProto protocol," 2015, https://core.telegram.org/mtproto.

[12] O. Schirokauer, "The number field sieve for integers of low weight," *Mathematics of Computation*, vol. 79, no. 269, pp. 583–602, 2010.

[13] D. Gillmor, "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)," 2016, IETF RFC 7919.

[14] K. Bhargavan, A. Delignat-Lavaud, and A. Pironti, "Verified contributive channel bindings for compound authentication," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS '15)*, Feb 2015. [Online]. Available: http://antoine.delignat-lavaud.fr/doc/ndss15.pdf

[15] A. Rad and J. Rizzo, "A 2^64 attack on Telegram, and why a super villain doesn't need it to read your telegram chats." 2015.

[16] J. Jakobsen and C. Orlandi, "On the cca (in)security of mtproto," Cryptology ePrint Archive, Report 2015/1177, 2015, http://eprint.iacr.org/2015/1177.

[17] N. Borisov, I. Goldberg, and E. A. Brewer, "Off-the-record communication, or, why not to use PGP," in *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, V. Atluri, P. F. Syverson, and S. D. C. di Vimercati, Eds. ACM, 2004, pp. 77–84. [Online]. Available: http://doi.acm.org/10.1145/1029179.1029200

[18] N. Wilcox, Z. Wilcox-O'Hearn, D. Hopwood, and D. Bacon, "Report of Security Audit of Cryptocat," 2014, https://leastauthority.com/blog/least_authority_performs_security_audit_for_cryptocat.html.

[19] P. A. Gardner, S. Maffeis, and G. D. Smith, "Towards a program logic for JavaScript," *SIGPLAN Not.*, vol. 47, no. 1, pp. 31–44, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2103621.2103663

[20] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 1, 2008.

[21] Joyent Inc. and the Linux Foundation, "Node.js," 2016, https://nodejs.org/en/.

[22] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *Advances in Cryptology (CRYPTO)*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6223, pp. 631–648.

[23] N. Kobeissi, "SP code repository," https://github.com/inria-prosecco/proscript-messaging, February 2017.

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

[25] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Advances in Cryptology (Eurocrypt)*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed., vol. 4004. Springer, May 2006, pp. 409–426.

[26] T. Okamoto and D. Pointcheval, "The gap-problems: a new class of problems for the security of cryptographic schemes," in *Practice and Theory in Public Key Cryptography (PKC)*, ser. Lecture Notes in Computer Science, K. Kim, Ed., vol. 1992. Springer, 2001, pp. 104–118.

[27] A. Fujioka and K. Suzuki, "Designing efficient authenticated key exchange resilient to leakage of ephemeral secret keys," in *Topics in Cryptology (CT-RSA)*, ser. Lecture Notes in Computer Science, A. Kiayias, Ed., vol. 6558. Springer, 2011, pp. 121–141.

[28] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *Public Key Cryptography (PKC)*, 2006, pp. 207–228.

[29] S. Goldwasser, S. Micali, and R. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal of Computing*, vol. 17, no. 2, pp. 281–308, April 1988.

[30] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-Damgård revisited: How to construct a hash function," in *Advances in Cryptology (CRYPTO)*, ser. Lecture Notes in Computer Science, vol. 3621. Springer, 2005, pp. 430–448.

[31] M. Bellare, "New proofs for NMAC and HMAC: Security without collision-resistance," in *Advances in Cryptology (CRYPTO)*, ser. Lecture Notes in Computer Science, C. Dwork, Ed., vol. 4117. Springer, 2006, pp. 602–619.

[32] D. A. McGrew and J. Viega, "The security and performance of the Galois/Counter Mode (GCM) of operation," in *Progress in Cryptology - INDOCRYPT 2004*, ser. Lecture Notes in Computer Science, A. Canteaut and K. Viswanathan, Eds., vol. 3348. Chennai, India: Springer, Dec. 2004, pp. 343–355.

[33] P. Rogaway, "Authenticated-encryption with associated-data," in *Ninth ACM Conference on Computer and Communications Security (CCS-9)*. Washington, DC: ACM Press, Nov. 2002, pp. 98–107.

[34] Open Whisper Systems, "Signal for the browser," 2015, https://github.com/WhisperSystems/Signal-Browser.

[35] GitHub, "Electron framework," 2016, http://electron.atom.io/.

[36] M. Avalle, A. Pironti, R. Sisto, and D. Pozza, "The Java SPI framework for security protocol implementation," in *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, Aug 2011, pp. 746–751.

[37] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *ECOOP 2014 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, R. Jones, Ed., vol. 8586. Springer, 2014, pp. 257–281.

[38] Facebook Inc., "Flow, a static type checker for JavaScript," http://flowtype.org/docs/about-flow.html.

[39] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to JavaScript," *SIGPLAN Not.*, vol. 48, no. 1, pp. 371–384, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480359.2429114

[40] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis," *Journal of Computer Security*, vol. 22, no. 4, pp. 601–657, 2014.

[41] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "AUTHSCAN: automatic extraction of web authentication protocols from implementations," in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[42] D. Fett, R. Küsters, and G. Schmitz, "An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System," in *35th IEEE Symposium on Security and Privacy (S&P 2014)*. IEEE Computer Society, 2014, pp. 673–688.

[43] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *IEEE Computer Security Foundations Symposium (CSF)*, 2016, pp. 164–178.

[44] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," in *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2017.