

From Specs to Apps: Verifying and Monitoring Models of Signal and WhatsApp

Full Version*

Moustafa Said

moustafa.said@cispa.de

CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Kevin Morio

kevin.morio@cispa.de

CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Aurora Naska

aurora.naska@cispa.de

CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Robert Künnemann

robert.kuennemann@cispa.de

CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

ABSTRACT

The Signal protocol is a prominent messaging protocol that secures communication for billions of users. It powers WhatsApp, the most widely used messaging application worldwide, and the Signal app, popular among privacy-conscious users. Extensive research in the computational and Dolev-Yao settings provides strong formal security guarantees for the protocol itself. However, a gap remains between the guarantees of the protocol specification and the implementation’s actual behavior at runtime.

In this work, we bridge this gap by applying SpecMon, a recently proposed runtime monitor, to check whether observed executions conform to formal protocol models. To this end, we instrument two applications (WhatsApp Web and Signal Desktop) to capture their interactions with the network and the cryptographic components. Using this instrumentation, we develop two multiset-rewrite models that are compatible with Tamarin, thus enabling verification. We derive the first model of WhatsApp Web’s implementation of the Signal protocol and the most detailed model to date of Signal’s original protocol. Monitoring establishes that observed executions conform to these models, relative to the trusted event extraction and the symbolic abstraction. For the core components of the Signal protocol, we verify authentication and secrecy properties. Finally, monitoring reveals previously undocumented differences between the original libsignal library and WhatsApp’s fork.

We evaluate our methodology and demonstrate its reproducibility. Developing the WhatsApp Web model, instrumenting the app, adding fuzzing, and running the experiments took three person-weeks. We also demonstrate efficient monitoring of real-world applications and detection of deliberately injected security faults, with low overhead in our measured setting.

1 INTRODUCTION

The Signal protocol is a prominent messaging protocol that secures the daily communications of billions of users worldwide. This open-source protocol [60] serves as the backbone of a multitude of modern messaging apps, including Signal [63], WhatsApp [71], Facebook Messenger [25], and Google Messages [30].

Due to its prevalence, there has been extensive research [7, 18, 21, 27, 37, 43, 68, 3, 15, 10, 34, 26, 39] to formally analyze the protocol.

In particular, the use of state-of-the-art automated verification tools like Tamarin [45] and ProVerif [11] has enabled analysis of some of the more complex security guarantees of the protocol [9, 21, 39]. However, keeping verification tractable often requires abstracting protocol details or focusing on a particular component instead of the entire system.

This leaves a verification gap: guarantees proved on a formal model do not transfer to the actual implementation of the protocol. The implementation may deviate from the model in various ways, e.g., by omitting security-relevant steps, by using different cryptographic primitives, or by introducing bugs. To further complicate matters, although messaging apps use an open-source protocol, they are typically closed-source, making it even more difficult to build trust that the theoretical guarantees of the underlying protocol transfer to the application. Facebook (now Meta), the current owner of WhatsApp, was named in the NSA’s PRISM disclosures [32], which contributed to long-standing concerns among security-conscious users about WhatsApp’s trustworthiness. Nevertheless, both Signal [28, 48] and WhatsApp [33] are still used for highly classified communication despite government restrictions.

It is hence fundamental to ensure that the implementation conforms to the verified model. Techniques for doing so include code verification [65], verified compilers (cv2ocaml [14], cv2fstar [40], and the compiler of [2]), type checking (F* [66]), model extraction [1, 37, 51], (automated) theorem proving [36], and refinement from specifications [8, 56, 6]. In contrast to these static techniques, which either require substantial proof effort or apply only to limited implementations, *dynamic verification* (runtime monitoring) provides a flexible, black-box view of the program. SpecMon [50], a recently proposed runtime monitoring engine, addresses this problem by checking compliance with a formal model during the execution of a protocol implementation. Within the bounds of the Dolev-Yao (DY) model and the monitored scope, guarantees of the formal model transfer to accepted implementation traces.

In this work, we establish formal models that allow SpecMon to monitor secure messaging applications. This entails two key tasks. First, the implementation needs to be *instrumented*, i.e., cryptographic components and network interfaces are identified and their inputs and outputs are exposed to the monitor. Second, the

* This is an extended version of [59].

model needs specification-level information that verification models sometimes omit. In theory, this only requires a description of the concrete on-the-wire format (SpecMon provides a language to express this that is backward compatible with Tamarin). In practice, formal models often omit potentially critical details for the sake of verification. To capture the desired behaviors of the implementation, more detail can be necessary.

Our work targets two prominent Signal protocol implementations: WhatsApp Web, version 2.3000.1029798056 (November 2025), and Signal Desktop, version 7.56.0-alpha.1 (April 2025). Our Signal Desktop experiments use libsignal v0.70.0.

In our first case study, we monitor Signal Desktop [62], an open-source client, to detect deviations during application runtime. Starting from an initial model of Signal’s session-handling layer, Extended Triple Diffie-Hellman (X3DH), and Double Ratchet (DR) [21, 57], we enhance the model with the post-quantum key exchange (PQXDH) [38], now supported by Signal, Sealed Sender [41], and other details omitted in previous work, such as cipher-key derivations. This leads to the most detailed model for Signal to date, which is validated against the actual implementation.

In our second case study, we show that this methodology also applies to closed-source implementations. We monitor a WhatsApp client [71] using Chrome DevTools [29], instrumenting WhatsApp’s modified version of libsignal. Starting from traces generated by the application and no previous model, we extract the first formal model of WhatsApp’s implementation of the Signal protocol.

For both models, we use Tamarin to prove authentication and secrecy of the initial root key under the threat models described in Section 7. For Signal, the analysis also considers an attacker that can break the Diffie-Hellman (DH) assumption. The authentication guarantee does not cover a DH break before the responder completes the handshake. We also instantiate an impossibility result [22] against post-compromise security (PCS, the guarantee that a conversation’s security can be restored after its secrets are compromised) in our setting. Additionally, we point out undocumented differences between Signal’s and WhatsApp’s DR implementations.

Our work demonstrates the practical value of monitoring real-world applications. First, our methodology enables fast and efficient model extraction of complex protocols: developing the WhatsApp Web model, instrumenting the app, adding fuzzing, and running the experiments took three person-weeks. Second, and more significantly, maintaining closely related monitoring and verification variants of one MSR model creates a shared validation point. Accepted observed traces conform to the monitorable variant within the stated trust and abstraction boundaries, while documented transformations connect that variant to the model used for proof. This makes the remaining abstraction gap explicit and permits both variants to be updated as the implementation evolves. Third, our artifacts support model extraction for other Signal-based apps.

More broadly, monitorable models offer benefits beyond verification by enabling integration into the development lifecycle for continuous validation. Developers can use these models as in-depth testing tools to ensure ongoing compliance with protocol specifications and as a form of executable documentation. Our work thus provides both the methodology and instrumentation for tech-savvy end users to establish greater trust in their messaging applications,

while offering vendors a systematic approach to document and maintain protocol correctness over time.

All materials and models needed to reproduce and extend our results are available as described in Appendix A.

Contributions. We make the following contributions:

- We are the first to demonstrate the feasibility of monitoring production-scale messaging applications with low measured overhead using SpecMon. We provide a methodology and reusable artifacts that enable quick and efficient monitoring of other Signal-based applications.
- We develop the most detailed model of Signal, unifying and extending prior Tamarin models. We monitor it against the implementation and verify authentication and initial-root-key secrecy. Sealed Sender is monitored for fidelity but not separately verified.
- We develop the first formal model of WhatsApp Web’s Signal-protocol variant, monitor it against the implementation, and verify authentication and initial-root-key secrecy. Both yield the known conversation-PCS counterexample.

Outline. We introduce formal verification and runtime monitoring in Section 2. Our monitoring methodology is presented in Section 3, followed by case studies on Signal Desktop and WhatsApp Web in Sections 4 and 5. We evaluate our approach through model validation, fault injection, and performance measurements in Section 6. Formal verification results are discussed in Section 7, and related work in Section 8. Finally, we present limitations, lessons learned, future work, and conclusions in Sections 9 to 11.

2 BACKGROUND

We first provide a high-level overview of the Signal protocol [64] used in both Signal Desktop and WhatsApp Web. Then, we introduce SpecMon [50], the runtime monitoring framework, and Tamarin [45], the verification tool used to formally verify the models. Finally, we explain how symbolic rules become monitorable.

2.1 The Signal Protocol

The Signal protocol consists of two main subprotocols: an authenticated key agreement protocol, instantiated with the Extended Triple Diffie-Hellman (X3DH) [44] or its post-quantum variant PQXDH [38], and a continuous message encryption algorithm, instantiated with the Double Ratchet (DR) [55]. The DR algorithm updates the key material used for sending and receiving messages after the initial key agreement. Typically, each pair of devices maintains one or more Signal connections (X3DH+DR). Each such connection is referred to as a session. Sesame [43] is the session management layer responsible for creating, deleting, and selecting these sessions.

At a high level, X3DH enables two parties to authenticate each other’s identities while deriving a shared initial secret, called the initial root key. Its authentication and secrecy guarantees depend on which keys are compromised and when. PQXDH adds post-quantum forward secrecy, but its authentication still relies on the hardness of the discrete logarithm problem [38]. To enable asynchronous communication, each user uploads prekey bundles to the server, which include a hierarchy of keys: an identity key for the

user, prekeys signed by the identity key, one-time prekeys, and, if supported, post-quantum prekeys signed by the identity key.

Using the initial secret as a seed, the DR algorithm derives message encryption keys while providing strong security guarantees: forward secrecy (FS) [13] ensures that past messages remain secure against future compromise, while post-compromise security (PCS) [19] allows future messages to become secure again after a current-state compromise and a subsequent healing period. The DR algorithm realizes these guarantees by combining (a) an asymmetric ratchet (PCS, FS) and (b) a symmetric ratchet (FS). In the asymmetric ratchet, parties exchange ephemeral Diffie-Hellman shares, and each new shared secret is merged into the root key. The symmetric ratchet uses a key derivation function (KDF) to derive message keys from the session secret. This way, every message is encrypted with a unique key, and compromise of later keys does not reveal information needed to compute earlier keys.

At the end-user level, Sesame manages the creation and use of sessions according to a specified policy, e.g., on decryption errors or desynchronization between the parties, which prevents some stronger PCS properties from holding [21, 22]. Finally, the protocol can be extended to also provide sender anonymity by enveloping the Signal messages using the Sealed Sender algorithm [41].

2.2 SpecMon and Tamarin

Tamarin and SpecMon use multiset-rewrite rules (MSRs) as their common specification language. This common language lets one protocol model serve two purposes: Tamarin verifies security properties specified as trace properties, while SpecMon checks whether concrete implementation traces can be explained by the same rules. Using the same specification language reduces divergence between the model used for verification and the model used for monitoring.

A rule is denoted as $l \xrightarrow{[a]} r$ and rewrites a multiset of facts. A fact consists of a fact symbol F and a list of terms. The premise l , actions a , and conclusion r are multisets of facts. If the system state is described by a multiset of facts S , then the rule is *applicable* if $l \subseteq S$ and, in that case, its application removes l from S and replaces it with r . Such a transition is labeled with the actions a . In Tamarin, terms are symbolic DY terms such as $\text{senc}(m, k)$, whose meaning is determined by user-defined equations. Tamarin reasons about the symbolic traces generated by the MSRs in the presence of the DY attacker to verify the specified security properties.

SpecMon, by contrast, observes concrete bitstrings produced by an implementation. It separates event extraction from checking: the event aggregator (EA) records relevant implementation events and forwards them to the monitor, which checks the event stream against the model. This separation allows different event aggregation mechanisms depending on the deployment context. We write such an event as $\langle f(x_1, \dots, x_n), y \rangle$, where f is the function name, the x_i are concrete arguments, and y is the concrete return value.

Since monitor events carry bitstrings rather than constructed terms, SpecMon rules cannot pattern-match on symbolic term structure in premises the way Tamarin rules can. Instead, message structure is recovered through format strings, which pattern-match on bitstring layouts. For cryptographic structure, the implementation must execute the corresponding function call. For example, instead of matching an encryption term in a premise, the monitor observes

whether a decryption call succeeds. The monitor maintains a set of configurations, each consisting of current facts. If several rules can explain an event, SpecMon keeps the corresponding configurations until later events potentially resolve the nondeterminism.

MSRs enriched with monitoring annotations are called *extended MSRs*. SpecMon’s soundness theorem [50, Theorem 7.1] states that, for a *likely* event stream (roughly, one in which freshly generated values are unique) accepted by a set of extended MSRs, there exists an abstraction from bitstrings to symbolic terms and a corresponding symbolic trace of the same model. For properties stated over the modeled symbolic trace, this theorem provides the link from Tamarin proofs to accepted monitored executions within the modeled scope and assumptions. We expand on the threat model, trusted event extraction, and pre-trace initialization in Section 3.

2.3 Monitorable Rules

A Tamarin model is not immediately monitorable: implementations expose concrete events, randomness, return values, and bitstrings. SpecMon therefore interprets function symbols in the symbolic model as function calls that the implementation may perform in the corresponding protocol state. An observed event that no rule permits is reported as a violation. SpecMon’s rule decomposition [50, Def. 6.1] splits a symbolic rule with nested function applications into a sequence of rules, each carrying at most one trigger for an individual function application. Format strings add the information needed to parse and construct concrete messages, and trace rewriting optionally normalizes implementation-specific calling conventions and multi-call operations.

Triggers. A trigger is a rule annotation. Operationally, it is the observed program event that lets the monitor apply the rule, provided the rule’s premise matches the current configuration. Users can write triggers directly, while SpecMon provides them automatically for ordinary symbolic computations. For example, the rule

```
rule Hash:
  [Start(x)] → [Next(h(x))]
```

is decomposed, schematically, into a rule with a trigger that matches an observed hash computation:

```
rule Hash [x-trigger=<h(x), y>]:
  [Start(x)] → [Next(h(x))]
```

When the event $\langle h(x), y \rangle$ is observed, SpecMon replaces the symbolic function application $h(x)$ with the concrete return value y when applying the rule. Similarly, facts In and Out , which in Tamarin denote communication with the network attacker, are monitored through network I/O events, and Fr , which asserts freshness, is monitored through randomness generation.

Format strings. Network messages in a Tamarin model are symbolic terms, while implementations send and receive bitstrings. Format strings specify how cryptographic messages are parsed and constructed. For monitoring, we define a concrete bitstring layout with ciphertext length prefix l :

```
message(RK, CT) = cat(string('magic_byte'),
                      byte(RK, 32), byte(l, 1),
                      byte(CT, l)),
```

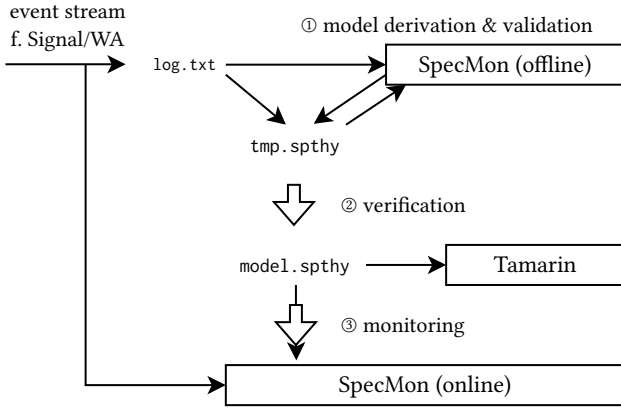


Figure 1. Methodology: data flow (solid) and outcomes (outlined).

For verification, the same symbolic message can be abstracted as a tuple giving the attacker access to each of its arguments:

$$\text{message}(RK, CT) = \langle CT, RK \rangle.$$

This lets SpecMon check bitstring-level messages while Tamarin reasons about symbolic terms.

Unified model. SpecMon also supports Tamarin’s preprocessor directives, including `#ifdef`, `#else`, and `#endif`. We use them to keep the verification-specific and monitoring-specific definitions in a single model file, for example to distinguish Tamarin’s attacker and corruption rules from the concrete network and implementation events used during monitoring.

Trace rewriting. Raw implementation events often differ from model events: function names may differ, functions may include extra implementation parameters, or one symbolic operation may be implemented through several library calls. SpecMon supports trace rewriting rules that normalize these events before they are checked against the protocol model (see Section 3.4).

3 MONITORING METHODOLOGY

We describe how we extend SpecMon, instrument Signal Desktop and WhatsApp Web to extract relevant events, and adapt or derive models of the Signal protocol to fit the actual implementations. Figure 1 provides an overview of our methodology.

Threat model for monitoring. The verification threat models in Section 7 describe the protocol attacker. Monitoring adds a deployment trust boundary. We target *honest-but-buggy* implementations and interpret SpecMon’s guarantees relative to trusted event extraction and pre-trace initialization, both detailed below. Concretely, we trust (1) the instrumentation, which consists of annotated libraries for Signal Desktop and DevTools-based runtime wrappers for WhatsApp Web (Section 3.2), together with the event aggregator, (2) the browser’s WebSocket and TLS stack, and (3) the pre-trace initialization of the monitor’s state (Section 3.3). Defenses against malicious applications are out of scope and discussed in Section 9.

Scope of the assurance claim. Monitoring ensures that accepted executions conform to the monitorable MSR model. This guarantee

is *trace-relative* and depends on the trusted components above, the symbolic (Dolev-Yao) abstraction of cryptography, and the behaviors exercised during validation (Section 6.1). Within this scope, monitoring catches observable model–implementation divergences, including incorrectly sequenced cryptographic operations, message-format mismatches, logical errors, invalid state-machine transitions, cryptographic misuse visible in events (e.g., a wrong IV), and implementation steps hidden by model abstractions. However, it cannot detect behaviors outside the event stream, side channels, memory leaks, storage behavior, or errors on unexercised paths.

Two further aspects follow from the modeling language. First, outputs are not enforced: the monitor checks that a sent message is *permitted*, not that it is eventually sent. Delaying or not sending a permitted message does not itself disclose additional data. Second, Tamarin’s input and freshness deduction applies in every state, so an implementation may sample additional randomness or receive additional messages without violating the model. Unused randomness and inputs do not affect the modeled protocol behavior. If these values are subsequently used in monitored operations, those operations must conform to the model.

Deployment scenarios. We currently deploy monitoring for development, testing, and auditing: it checks an implementation’s conformance to its formal model and can run as part of continuous integration, complementing protocol-level fuzzing. Long-running production monitoring, for example in regulated domains or to produce audit evidence, is a natural extension. It requires automating the currently manual instrumentation, as discussed in Section 9.

3.1 Extending SpecMon

Applying SpecMon to Signal Desktop and WhatsApp Web required extending the monitor in two directions. First, we added support for protocol behavior that appears in these implementations but was not covered by the previous monitor: cryptographic computations whose return values are not reused later in the protocol model, and recursive evaluation of nested format strings. These extensions let the monitor describe the concrete encodings and helper computations used by the applications without forcing artificial protocol state into the model.

Second, we improved the monitor’s execution engine to handle the size of these case studies. In particular, we reduced repeated work in rule matching and state updates, improved conflict-set computation to discard impossible states earlier, and shared repeated representations of terms and facts in memory. These changes do not alter SpecMon’s monitoring semantics. They make the same modeling approach practical for the larger rule sets, concurrent sessions, and event streams encountered in our experiments. The extended SpecMon version is included in the artifact.

3.2 Event Extraction

The event aggregator (EA) is responsible for extracting relevant events from the application and feeding them to the monitor. This is done by instrumenting the application to log function calls to networking libraries and cryptographic components. We use two instrumentation strategies: annotated libraries when the relevant code is available, and dynamic browser instrumentation when it is

not. The case-study sections describe the concrete instrumentation points for Signal Desktop and WhatsApp Web.

Network communication uses WebSockets protected by TLS. This means we trust the WebSocket implementation in the browser (for WhatsApp Web) or the Chromium networking stack bundled with Electron (for Signal Desktop). Ideally, rather than trusting the WebSocket layer, we would push the trust boundary down to the operating system’s networking stack and monitor it there directly with SpecMon. This is possible in principle, but would require a holistic model covering both TLS and the protocol under consideration, which is currently infeasible due to the size and verification time of existing TLS models [20].

Instrumenting the cryptographic library requires identifying its components and logging their function names, parameters, and return values. SpecMon’s guarantees depend on the EA emitting complete event streams, since it cannot distinguish a faulty implementation observed by a correct EA from a correct implementation observed by a faulty EA. An omitted event may cause rejection if a later monitored rule depends on it, but this is not guaranteed. For example, an uninstrumented send function may transmit data without the monitor observing it. Omissions can therefore hide invalid behavior. To allow a class of events to be omitted, one would have to show that adding those events back to an accepted stream preserves acceptance.

3.3 State Initialization

One part of the protocol cannot be monitored: the *setup*, typically the out-of-band configuration of common secrets. SpecMon works around this issue by allowing monitoring to start with a *pre-trace*, a prefix to the event stream. Typically, the user provides a script that reads the application’s configuration and emits some specified event (e.g., `<setup('key'), <>>`) that the monitoring rules pick up to initialize facts. Messaging apps retain sessions across restarts. Their keys evolve locally (e.g., by ratcheting), and the updated session state is persisted for the next run.

We use the pre-trace to recover this evolving state when the application starts using two scripts: (1) one that extracts the relevant key material from the application’s session table and (2) one that emits events that initialize the monitor’s state with these keys.

3.4 Trace Rewriting

The event stream includes function calls as they appear in the library, often with implementation details outside the model. Function names may differ from model symbols; for example, the library uses `aes_encrypt` where the model uses `senc`. Events may also include implementation-specific parameters (e.g., IVs and padding schemes) or split one model step into several calls, such as hash-state initialization, update, and finalization.

We use trace rewriting to adapt the event stream to the model. This is done by specifying a set of rewriting rules that transform the event stream to match the model. For example, we rewrite the function name `aes_encrypt` to `senc` and combine multistep operations into a single step. As shown in Listing 1, trace rewriting is built into SpecMon and uses the same MSR formalism as the protocol model: the special action `PPEvent` passes program events to the next layer, in our case, the Signal model.

Listing 1. Example: trace-rewriting rule that rewrites AES encryption calls to the model’s symmetric encryption function. Here, `ct` denotes the ciphertext returned by `aes_encrypt`.

```
rule ex [x-trigger=[<aes_encrypt(m,k,iv), ct>]]:
  [ ] -[PPEvent(<senc(m, k), ct>)] -> [ ]
```

3.5 Model Construction

When adapting a model for monitoring, we need to address abstractions of concrete message formats [49, 69], protocol features, and cryptographic operations. This involves adding missing function calls and parameters observed in the traces, as well as adapting the model to account for differences in cryptographic operations. For example, we extend Cremers et al. [21]’s Sesame model to include the derivation of a cipher key from the chain key, rather than encrypting directly with the chain key. We follow a systematic approach to adapt the model based on the rewritten traces from the EA: we analyze the cause of a rejected event, verify its validity according to the specification, and adapt the model accordingly.

Monitoring also revealed details that our initial adaptation of the Sesame model had omitted. The original model abstracts session setup by initializing a chain key directly from a fresh root key. In Signal Desktop, the initiator first performs a Diffie-Hellman ratchet step using its fresh ratchet key and the recipient’s signed prekey. This step updates the root key and derives the sending chain key before the first message is encrypted. SpecMon rejected the corresponding trace because our initial adaptation lacked this step. We therefore extended the model to include it.

Missing format strings. Input and output events that cannot be matched may indicate missing or incorrect format strings in the model. This is to be expected, as verification models often abstract away message formats as a simple nested pair encoding. We consult the Signal specification [64] and implementation [62] for details, noting that Signal uses Protocol Buffers [61] for its wire format.

Incorrect transition. SpecMon outputs the current facts when it rejects an event. If that state is not the one expected at this point in the trace, we identify the rule that led to the observed state fact and correct it. If there are multiple current states or if the prior state was also incorrect, we truncate the trace to find the first rejected event and inspect the state at that point to analyze what led us to the current state.

No valid transitions. In this case, we check:

- If there is a *missing transition* to an existing state, we add a new rule for this transition.
- If the model expects a *different order of operations* than the implementation, we adapt the model accordingly.
- If the model is *incomplete*, we either extend the model by adding missing functionality or introduce abstractions.

4 SIGNAL DESKTOP

We target the desktop version of Signal, written in TypeScript. We monitor the PQXDH key-exchange protocol, the DR protocol, and the Sealed Sender mechanism in Signal Desktop [62]. Since the implementation is open-source, we adopt the annotated-library approach from SpecMon [50]. For both network and cryptographic

components, we annotate functions in libsignal [60] used by Signal Desktop [62], and use our modified versions to run the app.

Furthermore, we monitor the session initialization in Signal Desktop by merging the Sesame model [21] with the X3DH model [57]. The model is then extended to fit the actual implementation.

4.1 Event Aggregation

Network events come from the WebSocket handlers `onmessage` (incoming) and `send_request` (outgoing). For cryptography, Signal clients and servers use the platform-agnostic libsignal [60] library. It is implemented in Rust and exposed to JavaScript. We annotate the functions that appear in the base models for DR [21] and X3DH [57], as listed in Table 3.

4.2 Pre-traces and Database Decryption

Signal Desktop stores session keys in an encrypted database on the user’s device. On Linux, it is typically located in the user’s application configuration directory (e.g., `~/.config/Signal` for production or `~/.config/Signal-development` for staging).

To obtain the database decryption key, we read the Signal profile configuration and recover the SQLCipher key from the stored key or `encryptedKey` entry. On Linux, this requires decrypting the stored key using the secret retrieved from the system secret service. We then use the resulting SQLCipher key to decrypt the database, extract the current session keys, and provide them to the monitor as pre-traces. For each session, we extract the root key; the sending ratchet key pair and chain key; the receiving public ratchet keys and chain keys; the base key (the public key of the ephemeral secret used in the handshake); and the session identifier (a tuple of the sender’s and recipient’s identifiers). These pre-traces act as triggers for rules that have no premises and whose conclusions introduce the facts required by subsequent rules. This setup allows us to monitor pre-existing sessions of Signal Desktop without having to monitor session initialization every time (see Section 3.3).

4.3 Trace Model Construction

We start with the DR model by Cremers et al. [21] for monitoring the DR protocol in Signal Desktop, and with the X3DH model [57] for the handshake. We then merge both models and extend them to fit the implementation traces and include missing details. These extensions cover post-quantum ML-KEM keys used during session initialization, track ratchet keys, and model the derivation of message and cipher keys from chain keys for encryption and decryption. We explicitly model the initial DH ratchet step before the first message and the successive root-key updates that derive a party’s receiving and sending chains.

Additionally, Signal Desktop uses a sender-identity-hiding mechanism, called Sealed Sender. This mechanism is used on top of the DR and PQXDH protocols to hide the sender’s identity from the server. Since neither initial model covers Sealed Sender and no prior model is available, we extend the model with rules for this mechanism. This makes our final model the most detailed model of Signal to date. Our model includes the following rules.

- Initiator: starts a session with a recipient.
- Responder: receives the initial message and responds.

- Prekey: the recipient publishes a prekey bundle to the server.
- Sender symmetric ratchet: advances the sending chain key and encrypts a message.
- Receiver symmetric ratchet: advances the receiving chain key and decrypts a message.
- Asymmetric (Diffie-Hellman) ratchet: updates the root key and derives new sending and receiving chain keys when receiving a message with a new ratchet key.
- Sealed send: the sender encrypts their public identity key using the recipient’s public key and an ephemeral key, and encrypts the message using the recipient’s public identity key together with the sender’s private identity key.
- Sealed receive: the recipient decrypts the sender’s identity using their private identity key and the ephemeral key, and the message using the sender’s public identity key and their own private identity key.

4.4 Format Strings

We extend the model with format strings to cover the actual structure of messages as observed in the traces. We derive the message structure from the Protocol Buffers definitions used in Signal Desktop [61], following the approach described in Section 2.3.

5 WHATSAPP WEB

WhatsApp Web [71] is a closed-source application, which makes monitoring more challenging than monitoring Signal Desktop [62]. However, although its JavaScript source code is minified, it remains partially readable, and many function names align closely with those from the white paper [70]. This correspondence significantly simplifies the task of identifying functions and operations relevant to the Signal protocol.

5.1 Event Aggregation

WhatsApp Web communicates over WebSockets. We use Chrome DevTools to identify and override WebSocket-related functions. To distinguish calls relevant to the Signal protocol from unrelated traffic, we inspect the stack trace at the time of the WebSocket invocation. The stack trace reveals the calling context, allowing us to selectively forward relevant events to SpecMon.

Additionally, since WhatsApp Web runs within the browser sandbox, it can only communicate over browser-exposed APIs. This security model simplifies monitoring by ensuring that the application cannot bypass our instrumentation through lower-level channels (e.g., using system calls, as would be possible in a native application). We assume that the browser and TLS are trusted and extract network I/O from the WebSocket components (Section 3.2).

WhatsApp Web uses a modified version of libsignal [60] for its cryptographic operations [42]. We identify cryptographic components by matching function names observed at runtime with those described in the white paper [70]. Using Chrome DevTools, we intercept these functions and step into them via the debugger to analyze their logic in detail. We then override the relevant cryptographic functions, as shown in Table 4, to log event traces to standard output for SpecMon to consume.

5.2 Pre-traces and Database Access

WhatsApp Web stores session keys in the IndexedDB database `signal-storage`. For each session, the `session-store` object store holds the base key; the root key; the five most recent receiver chains (including the receiver’s public ratchet keys and chain keys); and the sender’s chain (including the sender’s most recent ratchet key pair and chain key).

Other tables, like `prekey-store` and `identity-store`, contain the user’s one-time keys and public identity key. However, the private identity key is not saved in the database—it is stored as a non-extractable `CryptoKey` object in the browser’s memory and is not directly accessible.

Using Chrome DevTools, we query the database to extract the available keys. For the private identity key, we instead access the code module that loads it during runtime through the internal registration API `WWebCryptoLibrary.DbCallbacksApi.getRegistrationInfo`. We then pass all these keys to the monitor as pre-traces (Section 3.3).

5.3 Model Derivation

For WhatsApp Web, no prior formal model is available. We derive a model of its implementation of the Signal protocol directly from the recorded traces. The model covers both the Double Ratchet protocol and the X3DH key exchange protocol. Our model includes the following rules.

- X3DH initiator: starts a session with a recipient.
- X3DH responder: receives the initial message and responds.
- X3DH prekey: publishes the recipient’s bundle to the server.
- Sender symmetric ratchet: advances the sending chain key and encrypts a message.
- Receiver symmetric ratchet: advances the receiving chain key and decrypts a message.
- Asymmetric (Diffie-Hellman) ratchet: updates the root key and derives new sending and receiving chain keys when receiving a message with a new ratchet key.

5.4 Format Strings and BLOB Decoding

When the user sends a message (by clicking Send in the UI), WhatsApp Web encrypts it for both the user’s primary device (phone) and all recipient devices. Each encrypted payload is serialized as a separate `SignalMessage`, which contains the ciphertext, public ratchet key, and counters using Protocol Buffers [31]. These individual `SignalMessage` objects are then bundled together into a single `WebSocketMessage` for transmission.

Because a single `WebSocket BLOB` can contain an arbitrary number n of `SignalMessage` objects, the monitor would require a format string for every value of n . To manage this complexity, we delegate the decoding of the BLOBs and `SignalMessage` objects to the event aggregator (EA). Using Chrome DevTools, the EA extracts and decodes each `SignalMessage` from the `WebSocket` bundle and emits it individually to the monitor. Consequently, the monitor receives each `SignalMessage` as a separate event, allowing it to process the messages one by one.

5.5 Findings: Differences from Signal Desktop

Monitoring surfaced two differences between WhatsApp Web and Signal Desktop. The first is an engineering difference whose theoretical bearing on security we discuss below, and the second is a known feature-adoption difference.

Observation 1: read receipts outside the Double Ratchet. We observed that WhatsApp Web transmits read-receipt messages outside the Double Ratchet encryption layer, unlike text messages and reaction messages. In contrast, Signal Desktop integrates such messages into the DR protocol. This distinction is evident during runtime monitoring. Using SpecMon, we observe the DR rules being triggered and applied in Signal Desktop even for read-receipt messages, whereas in WhatsApp Web they are triggered and applied only for actual message exchanges. Furthermore, inspecting the `signal-storage` database in WhatsApp Web during read-receipt transmission shows no updates to any of the keys, confirming that these events are not encrypted as part of the DR protocol. Instead, WhatsApp Web handles read receipts on the server side. The client sends a server-visible *presence message* whenever the user opens a chat. Depending on the user’s settings, the server decides whether to send a read receipt to the other party.

As a result, we observe that WhatsApp Web performs DH ratchets less frequently than Signal Desktop. In Signal Desktop, read receipts participate in the encrypted message exchange and can trigger DH ratchets. In WhatsApp Web, receiving a read receipt does not trigger a DH ratchet.

For a single session, post-compromise recovery depends on the asymmetric ratchet mixing fresh DH material into the root key. Signal may therefore heal sooner than WhatsApp for comparable user interactions. Forward secrecy for already-sent messages is provided by the symmetric ratchet in the modeled protocol and is not weakened by this difference in DH-ratchet frequency. In practice, we do not conclude a weakening of WhatsApp’s security from this observation alone. Whether a compromised session actually heals also depends on how long old key material persists in memory and storage, which our methodology does not observe. We show no concrete attack based on this difference. An analysis of finer-grained healing properties remains future work.

Observation 2: feature-adoption differences. The WhatsApp Web version studied here does not incorporate a post-quantum mechanism (such as Signal’s PQXDH) during session initialization and has no mechanism to hide the sender’s identity (such as Signal’s Sealed Sender), unlike Signal Desktop. These are known feature-adoption differences between the applications. Our contribution is to surface them at the model level and document their behavioral consequences through monitoring.

6 EVALUATION

We performed an extensive evaluation of our monitoring approach on both Signal Desktop [62] and WhatsApp Web [71].

6.1 Model Validation

To validate the functionality of our monitoring models for Signal Desktop and WhatsApp Web, we monitor a party *Monique*, running the respective software, in communication with an unmonitored

partner *Parker*. Our goal is to cover a variety of bilateral communication patterns, including network and state anomalies. We do not attempt to cover group messages (not covered by our models) or linked devices for Signal (from the protocol perspective, linked devices are essentially independent communication partners).

To this end, we trigger the following UI or external actions, which capture the main behaviors of the Signal protocol: (1) starting a new conversation, triggering session *initialization*; (2) sending m consecutive messages, triggering *symmetric* ratcheting; (3) receiving m consecutive messages, triggering *symmetric* ratcheting; (4) swapping two messages in the network stack, so they appear out of order; (5) skipping a message in the network stack; and (6) sending or receiving a message to or from a device that has lost its session, triggering a *retry request*. This can occur when the communication partner switches phones or otherwise loses state.

For WhatsApp Web, (4) and (5) intercept messages after Signal-protocol encryption and before Noise transport encryption. Re-ordering the encrypted WebSocket frames instead would violate Noise’s sequence-number checks. Both our models and instrumentation support out-of-order messages, and the artifact includes the corresponding fuzzer actions. See Appendix C for details.

The artifact provides randomized fuzzers that exercise the enabled actions. The Signal Desktop fuzzer selects uniformly among actions whose prerequisites hold. The WhatsApp Web fuzzer first selects uniformly among the available action groups—sending, receiving, session initialization, retry, and out-of-order delivery when enabled—and then uniformly within the selected group. Actions that replay held messages are eligible only when such messages are available. Consecutive-send/receive batch sizes use a configurable rounded lognormal distribution with a minimum of one. The action set, distribution parameters, and run length are configurable. The idea is that the actions cover all the features in the scope of the monitor so that, from their random arrangement, corner cases may emerge that we may not anticipate with handwritten tests. For instance, while we knew a priori that switching from sending to receiving (or vice versa) would trigger the *Diffie-Hellman* ratcheting step, we found that the combination of (6) followed by (2) triggers fail-safe behavior. When Parker receives Monique’s message for the deleted session, they cannot decrypt it and return a decryption error. Monique then starts a new session and sends a new message.

We validate the models on three mixed workloads. The Signal Desktop trace contains 995 send/receive hooks and 35,040 raw instrumentation observations (25.02 MB). The ordinary WhatsApp Web trace contains 418 hooks and 11,918 observations (6.51 MB); its out-of-order workload contains 499 hooks and 15,960 observations (8.47 MB). Hook counts refer to send/receive instrumentation events, not necessarily distinct chat messages. Rewriting and monitoring these traces yields 17,393, 7,232, and 9,019 monitor events, respectively, including initialization events. All replays succeed.

We also replay consecutive-send workloads containing 7,007 send hooks for Signal Desktop and 3,009 for WhatsApp Web, with 168,271 and 73,246 raw observations, respectively. Both replays succeed. Additional ratcheting traces exercise repeated alternation between sending and receiving, and these replays also succeed.

6.2 Fault Injection Experiments

To evaluate the effectiveness of monitoring in detecting security-critical errors, we inject faults through implementation modifications and a trace-level omission. SpecMon detects the tested protocol-level faults; only payload-encoded leakage is outside the model’s reach.

Critical secret leakage. We leak a critical secret in three ways. First, we send it in an extra message. Second, we place it in a chat message’s content. Third, we add it as a new field in a protocol message.

In the first case, SpecMon detects the leak because of an unexpected event in the trace, and in the third case, because of an unmatched format string. In the second case, SpecMon cannot distinguish the secret from a legitimate message, since the secret is encoded as application payload. See Section 9 for details.

Incorrect use of a cryptographic library. First, during the Diffie-Hellman ratchet, we reuse a DH private key in Signal Desktop. For WhatsApp Web, we omit the key-generation event from the trace. SpecMon rejects the subsequent Diffie-Hellman computation because the required fresh-key generation was not observed. Second, we skip a signature check in the function `process_prekey_bundle`, so all prekey bundles from the server are accepted (similar to goto-fail, CVE-2014-1266 [52]). SpecMon detects the missing signature verification in both applications. These experiments do not establish the random-number generator’s security, as predictability of fresh-looking random values is outside the symbolic model.

6.3 Performance Overhead

We measure monitoring performance by replaying three workloads: Signal Desktop, WhatsApp Web, and WhatsApp Web with out-of-order delivery. We use a native Linux ARM64 virtual machine with 16 virtual CPUs and 16 GiB of RAM, Go 1.26.8, and three sequential repetitions per prefix after one warmup at the smallest prefix size. Prefixes contain 90, 180, ..., 900 send/receive hooks for Signal and 40, 80, ..., 400 hooks for each WhatsApp workload. Hooks are instrumentation observations, not necessarily distinct messages.

Figures 2a and 2b show the mean main-monitor processing time per event and mean peak process RSS. The processing-time statistic excludes the preceding rewrite stage; peak RSS covers the entire SpecMon process, including rewriting. The ranges are Signal: 0.277–0.336 ms per event and 31.29–53.54 MiB peak RSS; WhatsApp: 0.082–0.257 ms per event and 22.00–37.44 MiB peak RSS; WhatsApp OOO: 0.099–0.271 ms per event and 26.22–43.30 MiB peak RSS. Across individual repetitions, peak RSS reaches 83.56 MiB for Signal, 43.99 MiB for WhatsApp, and 46.09 MiB for WhatsApp with out-of-order delivery.

The 900-hook Signal prefix contains 31,692 raw observations and 15,724 monitor events, including its initialization pretrace. The 400-hook WhatsApp prefix contains 11,446 raw observations and 6,938 monitor events; the corresponding WhatsApp out-of-order prefix contains 12,915 raw observations and 7,195 monitor events. Thus the ordinary prefixes contain 35.21 and 28.62 raw observations per hook, respectively.

In a separate experiment, we evaluate the latency of Signal Desktop. The overall latency consists of SpecMon’s processing time and

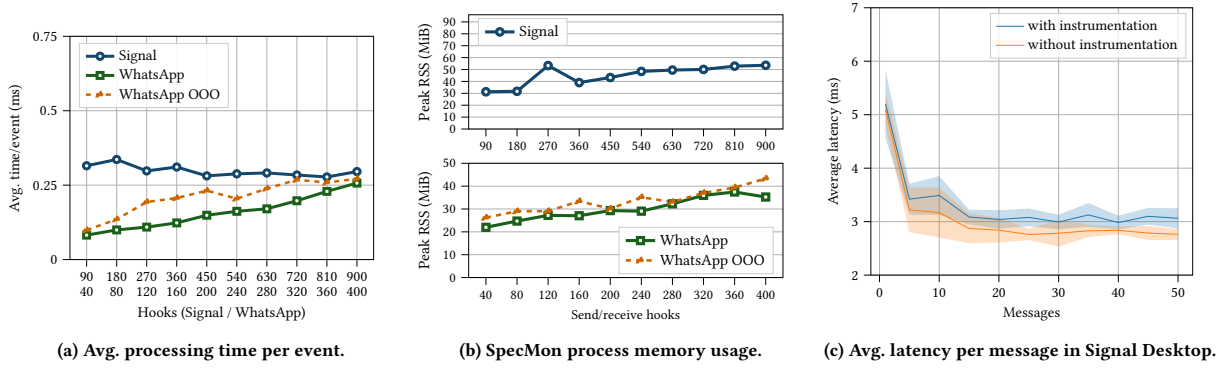


Figure 2. SpecMon performance for Signal Desktop and WhatsApp Web. Bands in (c) show mean $\pm$ one sample standard deviation across run means.

the computation overhead of the instrumentation (e.g., dereferencing pointers, copying buffers, and building strings in the event streams). We only measure end-to-end latency for Signal Desktop, as we instrument WhatsApp Web dynamically at runtime (‘monkey patching’), which is not a realistic deployment scenario. The expectation that source-level WhatsApp instrumentation would have comparable latency is therefore an extrapolation from the comparable SpecMon processing times (Figure 2a). Monkey patching identifies functions by name in the minified bundle, which makes it a cheap alternative to reverse-engineering a closed-source client from scratch; in exchange, each upstream rename or bundle reshuffle requires relocating the affected functions. For Signal Desktop, we define latency as the time between the event of inputting a message (collected from the function `sendMessageToServiceId` in [62]) and the event of the encrypted payload being handed off to the WebSocket layer for transmission (collected from `sendMessages` in [62]), i.e., the encryption overhead of the send path. For each m , we measure the latency on the unaltered code (the baseline) and on the instrumented code interacting with SpecMon. In Figure 2c, we report both latency measurements over $m \in \{1, 5, 10, 15, \dots, 50\}$. The curves flatten by $m = 50$, so we omit larger values. We find that the latency differences fall within the fluctuation of the average latency. This variation decreases as the number of messages increases. For $m = 50$, the average instrumentation overhead is 0.30 ms over the uninstrumented average of 2.76 ms. This performance can still be improved upon, as the prototype is not particularly optimized. For example, the instrumentation sends JSON objects to SpecMon through standard output. Calling the SpecMon library directly would avoid several copy operations.

7 VERIFICATION

In this section, we summarize our verification results for the core key-agreement and session components of the WhatsApp and Signal models. The monitorable Signal model additionally covers Sealed Sender for implementation fidelity, but we do not verify a sender-privacy property for it. This would require an observational-equivalence analysis outside our current Tamarin workflow.

7.1 Models

Our models are structured around three components: the setup phase, the key agreement, and the message exchange.

Setup. We model an unbounded number of users that can have an unbounded number of sessions and send or receive an unbounded number of messages during their communication. Each user is equipped with an unbounded number of prekey bundles, and depending on whether they execute X3DH or PQXDH, we model different initialization steps. In WhatsApp, we model a long-term identity key pair (ltk_A, LTK_A), signed prekeys (sk_A, SK_A), and one-time prekeys (opk_A, OPK_A). The numbers of signed and one-time prekeys are unbounded. For Signal, we also model an unbounded number of one-time post-quantum keys ($mlk_{emsk}_A, mlk_{emPK}_A$).

Key Agreement. As discussed, Signal employs the PQXDH protocol and WhatsApp the classical X3DH protocol. Typically, modelers would condense the key agreement into two rules for the initiator and responder, since this helps during verification. However, to allow for monitoring, our model represents the step-by-step atomic construction of the state in the implementation, where, for example, the initial key can be computed only after verification of the prekey signatures. We capture the key agreement for WhatsApp in six initiator rules and two responder rules, and for Signal in five initiator rules and two responder rules.

Message Exchange. The Double Ratchet protocol is defined by its two components: the symmetric and asymmetric ratchets. In the symmetric ratchet, captured in three rules, each party can send and receive an unbounded number of messages by advancing the sending and receiving chain keys. This is expressive enough to capture out-of-order delivery of messages. The asymmetric ratchet models the updated state upon receiving a message with a new Diffie-Hellman share. In contrast to previous models, ours captures the implementation’s fast-forwarded state, where an incoming message immediately also advances the sending chain, i.e., if B receives a message, they consecutively perform two asymmetric-ratchet steps: one to update the receiving chain and decrypt the message, and the other to prepare a new sending chain for the future. This is captured in two rules for WhatsApp and one rule for Signal. The single rule in Signal was created by merging two rules into one, which was necessary to speed up verification. This transformation is sound, as we exploit the rule composition result of [17, Appendix E.2.1], validating that these two rules adhere to their compatibility criterion *canmerge*. The monitored model still observes

the implementation-level steps; the merged rule is used to make verification tractable. Previous models also treat this step as atomic.

7.2 Threat Models

Our threat model includes Tamarin’s predefined network attacker, who can drop, inject, and replay any message in the conversation. Additionally, we strengthen the attacker’s capabilities by leaking the setup keys of any user to the network. This is modeled in rules like the following, where the attacker learns the long-term key:

```
rule compromiseLTK:
  [!PrivateKey(A, ltk_A)]  $\longrightarrow$  [Out(ltk_A)]
```

We define three threat models. For the authentication and secrecy properties of Signal and WhatsApp, we define $\mathcal{A}_{\text{Signal}}$ and $\mathcal{A}_{\text{WhatsApp}}$, respectively. For PCS, we use the threat model $\mathcal{A}_{\text{PCS}}$.

$\mathcal{A}_{\text{Signal}}$. The attacker can compromise any user’s setup keys, including the long-term identity key, signed prekeys, one-time prekeys, and one-time post-quantum prekeys. In addition, the attacker can break the Diffie-Hellman assumption. We model this in one additional rule, which allows the attacker to learn the private share corresponding to any honest public Diffie-Hellman key.

$\mathcal{A}_{\text{WhatsApp}}$. The attacker can compromise all setup keys of any user, including the long-term identity key, signed prekeys, and one-time prekeys.

$\mathcal{A}_{\text{PCS}}$. Following the threat model of the impossibility result for PCS [22], the attacker can compromise the static state of any user. In the WhatsApp and Signal implementations, this translates to the compromise of the long-term identity key.

7.3 Properties

We focus on three main properties: authentication, secrecy of the initial root key, and conversation PCS for the end user. The authentication and secrecy properties are comparable to those in the ProVerif analysis of PQXDH [9], while the PCS property is defined as in [22]. Figure 3 shows the properties for Signal, since it involves a more complicated threat model, and we provide the corresponding properties for WhatsApp in the artifact.

Authentication. We prove that a responder completing PQXDH has a matching initiator session with the same parameters, unless the attacker has compromised specific keys or already broken the DH assumption (Figure 3a). The exception for an earlier DH break means that this lemma does not establish authentication against an active quantum attacker.

Secrecy. We show the responder secrecy property in Figure 3b. We have also specified and proved the equivalent property for the initiator. These are initial-root-key secrecy properties, not new proofs of DR message-key forward secrecy. Informally, the initial secret should remain secret except under specific conditions: compromise of the identity key, compromise of prekeys, a break of the DH assumption before the protocol run, or the attacker learning all relevant keys through compromise or a broken primitive. This is also called forward secrecy for the session key [9]. This key-agreement property is distinct from the DR message-key forward secrecy listed separately in Table 2.

Table 1: Tamarin formal analysis summary. Proofs were obtained either automatically using Tamarin’s heuristic proof search or by replaying manually constructed proofs. The runtime shows the time needed for Tamarin to find a proof automatically or to verify a stored proof.

Property	Threat model	Result	Runtime
WhatsApp			77 s
Authentication	$\mathcal{A}_{\text{WhatsApp}}$	✓	26 s
Initiator secrecy	$\mathcal{A}_{\text{WhatsApp}}$	✓	2 s
Responder secrecy	$\mathcal{A}_{\text{WhatsApp}}$	✓	30 s
PCS	$\mathcal{A}_{\text{PCS}}$	att / ✓	19 s
Signal			53 s
Authentication	$\mathcal{A}_{\text{Signal}}$	✓	13 s
Initiator secrecy	$\mathcal{A}_{\text{Signal}}$	✓	7 s
Responder secrecy	$\mathcal{A}_{\text{Signal}}$	✓	12 s
PCS	$\mathcal{A}_{\text{PCS}}$	att / ✓	21 s

✓ = verified property att / ✓ = expected known attack reproduced (counterexample)

Post-Compromise Security. We define conversation PCS for the end user, where, informally, any message sent after a healing period should be secure, as long as the attacker did not compromise the parties again after that period. However, this property does not hold in these applications for two reasons. First, their session layer enables multiple sessions. Second, once the long-term identity key is compromised, the attacker can start their own sessions in parallel. The property is falsified in our model as expected (Table 1), reproducing the known impossibility result in our session-layer setting. Modeling a single-session client would sidestep the first issue, but is out of scope here since we target the deployed, session-layer-enabled clients. We give the falsified property in Figure 3c.

7.4 Summary of Results

The WhatsApp model is constructed from 15 rules modeling the protocol components, and the Signal model from 14. The threat model contributes between 1 and 5 attacker rules, depending on the property under analysis: 3 for $\mathcal{A}_{\text{WhatsApp}}$, 5 for $\mathcal{A}_{\text{Signal}}$ (which adds a one-time post-quantum key compromise rule and a Diffie-Hellman break rule), and 1 for $\mathcal{A}_{\text{PCS}}$. For each model, we verified the authentication, initiator-secrecy, and responder-secrecy lemmas listed in Table 1; reproduced the expected conversation-PCS counterexample from the literature; and proved 12 sanity traces that safeguard the executability of each step of the model. The secrecy lemmas concern the initial root key, while DR message-key forward secrecy is outside our proved properties. The sanity traces were proved in 4 minutes for WhatsApp and 6 minutes for Signal. The verification process was run on a Lenovo ThinkPad X1 Carbon Gen 9 with 16 GB of RAM using Tamarin 1.11.0 on the develop branch. We summarize our results in Table 1.

8 RELATED WORK

We first survey previous formal analyses of the Signal protocol, then summarize known limitations in the DY model, and finally discuss approaches for aligning formal models with implementations by using fuzzing techniques and static analyses.

$\forall LTK_A LTK_B SPK_B OPK_B mlkemPK_B key i j.$ <i>// B completes the key agreement at i</i> $RespDone(LTK_B, LTK_A, SPK_B, OPK_B,$ $mlkemPK_B, key) @ i$ <i>// A completed a matching agreement</i> $\Rightarrow (\exists j. j < i \wedge$ $InitDone(LTK_A, LTK_B, SPK_B, OPK_B,$ $mlkemPK_B, key) @ j)$ <i>// Or A's LTK was compromised before i</i> $\parallel (\exists k. k < i \wedge CompromiseLTK(LTK_A) @ k)$ <i>// Or B's SPK was compromised before i</i> $\parallel (\exists k. k < i$ $\wedge CompromiseSPK(LTK_B, SPK_B) @ k)$ <i>// Or DH was broken before i</i> $\parallel (\exists k. k < i \wedge BrokenDH() @ k)$	$\forall LTK_A LTK_B SPK_B OPK_B mlkemPK_B key i j.$ <i>// B completes the key agreement with key</i> $RespDone(LTK_B, LTK_A, SPK_B, OPK_B,$ $mlkemPK_B, key) @ i$ $\wedge K(key) @ j$ <i>// and the attacker knows key</i> $\Rightarrow (\exists k. k < i \wedge CompromiseLTK(LTK_A) @ k)$ <i>// Or B's LTK or SPK was compromised before i</i> $\parallel (\exists k. k < i \wedge CompromiseLTK(LTK_B) @ k)$ $\parallel (\exists k. k < i$ $\wedge CompromiseSPK(LTK_B, SPK_B) @ k)$ <i>// Or both one-time prekeys were compromised</i> $\parallel (\exists k_1 k_2. CompromiseOPK(OPK_B) @ k_1$ $\wedge CompromisePQK(LTK_B, mlkemPK_B) @ k_2)$ <i>// Or DH was broken before i</i> $\parallel (\exists k. k < i \wedge BrokenDH() @ k)$ <i>// Or later DH break and B's PQ compromise</i> $\parallel (\exists k_1 k_2. i < k_1 \wedge BrokenDH() @ k_1$ $\wedge CompromisePQK(LTK_B, mlkemPK_B) @ k_2)$	$\forall LTK_A LTK_B key_0 key_1 key_2 i_1 i_2 j t.$ $i_1 < i_2 \wedge i_2 < j$ <i>// A and B exchange DH shares to heal</i> $\wedge Heal(LTK_A, LTK_B, key_0) @ i_1$ $\wedge Heal(LTK_B, LTK_A, key_1) @ i_2$ <i>// No compromise occurs between i₁ and i₂</i> $\wedge \neg(\exists \text{any } k. i_1 < k \wedge k < i_2$ $\wedge Compromise(\text{any}) @ k)$ <i>// An honest message step follows</i> $\wedge StepParty(LTK_A, LTK_B, key_2) @ j$ $\wedge K(key_2) @ t$ <i>// The attacker knows key₂</i> $\Rightarrow$ <i>// A's or B's LTK is compromised after i₂</i> $(\exists k. i_2 < k \wedge CompromiseLTK(LTK_A) @ k)$ $\parallel (\exists k. i_2 < k \wedge CompromiseLTK(LTK_B) @ k)$
(a) Authentication.	(b) Responder secrecy.	(c) Post-compromise security.

Figure 3. Tamarin formulations of the main Signal properties considered in our verification.

Table 2: Comparison of Signal models by covered features. Security properties are compared approximately. Prior models target a narrower protocol fragment. Our model covers the full handshake (X3DH+PQXDH), the ratchet, and Sealed Sender. We trade re-proving session/device-level DR forward secrecy for broader coverage and monitorability.

Feature	[21]	[9]	[39]	Ours
X3DH handshake	✗	✓	✓	✓
PQXDH ($\approx$ X3DH + KEM)	✗	✓	✗	✓
Ratcheting	✓	✗	✓	✓
Accurate asymmetric ratchet	✗	✗	✓	✓
Accurate symmetric ratchet	✓	✗	✓	✓
Message-key derivation	✗	✗	✓	✓
Message formats	✗	✗	✗	✓
Authentication	✗	✓	✗	✓
Secrecy	✗	✓	✓	✓
(DR) Forward secrecy	✓ ^{s,d}	✗	✗	✗
Post-compromise security	✓ ^s att ^{*d}	✗	✗	att ^d

(*) = attack found (s/d) = session / device compromise

8.1 Formal Analyses of Signal

Analysis in the computational model. The Signal protocol has been analyzed several times in the computational model [3, 18, 15, 10], including its later-developed post-quantum handshake PQXDH [34, 26, 9]. Cohn-Gordon et al. [18] analyze the Double Ratchet protocol in a multistage authenticated key exchange model of two devices without session handling, establishing forward secrecy, post-compromise security, and session-key indistinguishability. As is customary in the computational model, these works idealize the specification, making direct comparison with the deployed implementation difficult.

Kobeissi et al. [37] present a TypeScript reimplementaion of Signal and derive models from a restricted language subset. Their

implementation omits the symmetric ratchet, supports only a single session at a time, and lacks multi-device support, session replacement, and prekey replenishment. Our work instead monitors deployed clients and incorporates Signal’s Sesame session-handling layer [64], a detailed DR model, and the PQXDH handshake.

Analysis in the DY model. We compare our work with other DY analyses of the Signal protocol, as summarized in Table 2.

Cremers et al. [21] introduce the first Tamarin model that includes Sesame. They show that the DR achieves post-compromise security at the session level, but not at the conversation level due to Signal’s multi-session handling (Section 7). Their model intentionally abstracts from the handshake to retain tractability, using a simplified DR subprotocol and omitting X3DH entirely. The authors deem a complete model intractable for automated verification in Tamarin, noting that state-of-the-art formal approaches struggle to handle X3DH key agreement and DR accurately even without additional mechanisms. We extend their model to align with the runtime behavior of Signal Desktop and WhatsApp Web, enabling monitoring (Section 4 and Section 5). This includes the PQXDH handshake and other details necessary for monitoring, such as detailed message formats.

Bhargavan et al. [9] perform the first formal analysis of the PQXDH protocol, i.e., the handshake protocol, but not the DR protocol. They use the verification tools ProVerif and CryptoVerif, the latter providing guarantees in the computational model. They analyze authentication and forward secrecy of the session secret under classical and post-quantum threat models. Their guarantees account for the timing of key compromises and cryptanalytic attacks. In particular, their authentication result excludes a DH break before the key exchange [9, Theorem 8]. We prove similar, but coarser, properties for PQXDH in our work, e.g., we do not prove quantum forward secrecy for the initiator or model cryptanalytic attacks that decapsulate KEM ciphertexts.

Linker et al. [39] model X3DH and DR in Tamarin, capturing both the handshake and ratcheting protocol, but not out-of-order

delivery. They prove message secrecy under leakage of setup keys and Diffie-Hellman shares, using a new methodology to reason about the protocol’s looping behavior. In comparison, we prove the secrecy of only the initial secret, instead of the message key.

Limitations in the DY model. Tamarin’s constraint solver uses unification to determine the possible origins of terms such as encrypted messages or keys. Its built-in Diffie-Hellman theory supports multiplication in the exponent, but omits addition [45]. Jackson [35, Chapters 4–6] explains that directly extending unification-based reasoning to the full algebraic structure of DH exponents faces undecidability barriers. He also shows that abstracting away small-subgroup and invalid-curve behavior can hide attacks. Thus, the choice of cryptographic abstraction limits both the protocols that can be represented and the attacks that an analysis can detect. Monitoring alone does not remove these abstraction limits.

The Sesame model [21] abstracts away the X3DH key agreement and focuses on session handling, assuming that key establishment was performed correctly. Our models include the key agreement and ratchet computations needed for monitoring. For verification, we simplify implementation details and merge rules to keep proof search tractable (Section 7.1).

8.2 Implementation and Fuzzing Studies

Automated model inference has been used to reconstruct finite automata that approximate Signal’s implementation from observed I/O and compare them with the specification automaton [68]. That work finds differences that could prevent future secrecy from holding against attackers with device access. The approach relies on fuzzing-based model extraction to find attacks, but reconstructs only a finite-state machine and omits the ratchet state.

Outside of Signal, there are few works that compare protocol models with existing implementations. TLSPuffin [4] combines fuzzing with a DY attacker to find implementation-level attacks that can only be triggered deep within the protocol run. Their work is intended for attack finding, not verification, but could be used with a monitored implementation to find where it diverges from the model, thereby helping to improve the implementation or identify missing behaviors in the model. Tookan [12] determines the configuration of a PKCS#11 model from a set of tests run on a real hardware token and translates attacks found in the model into interactions with it. This methodology is protocol dependent.

Static analyses [65, 66, 1, 37, 51, 36, 8, 56, 6] can establish conformance without runtime overhead on small examples, but require substantial expertise and become increasingly costly as the implementation grows. The most closely related work, Igloo [65], soundly links compositional refinement and separation logic for distributed-system verification. Arquint et al. [6] connect this style of implementation reasoning to Tamarin models for security protocols. Both approaches require considerable proof and specification effort. Verified compilers (cv2ocaml [14], cv2fstar [40], and the compiler of [2]) do not apply to implementations that are already in use. In contrast, SpecMon [50] performs runtime monitoring by observing concrete execution traces of the implementation, rather than exploring all possible execution paths symbolically, as static verifiers do. This allows SpecMon to fully support the composition of the X3DH/PQXDH key agreement and the Double Ratchet in

one model and to be extended further. Using this capability, we extend the Sesame model to allow for monitoring.

9 LIMITATIONS AND FUTURE WORK

Although we have demonstrated the applicability of our methodology, there are still several open problems around functionality, security, and trustworthiness. Fortunately, many of them motivate interesting directions for future research.

Security: key exfiltration. Currently, SpecMon provides no protection against *malicious* applications, as opposed to merely incorrect ones. As we have seen in Section 6.2, all protocols that have payloads allow the exfiltration of secrets. For Signal/WhatsApp, the application simply encodes the long-term secrets in a chat message to a malicious outsider. There is no way to detect encodings of secrets, as they are not known a priori. This does not contradict the formal soundness of SpecMon or Tamarin, as the payload (e.g., the content of chat messages) is considered an atom in the MSR modeling language: the monitor cannot distinguish ordinary text from bytes that encode a long-term key. Thus, the modeling language and soundness result encode the requirement that payload be independent of protocol secrets, which, in our setting, is an assumption about an application we do not fully trust.

As exfiltration cannot be prevented, the secrets must be hidden from malicious applications. If set up correctly, cryptographic APIs like PKCS#11 can (provably [12, 24]) guarantee key secrecy. A key challenge here is to find a way to transparently apply this change, even for applications that do not use PKCS#11.

Functionality: behavior coverage. We tested our monitoring models through manual GUI interactions and randomized sequences of UI and external actions (Section 6.1). There could be other desired implementation behaviors that we missed, in which case the monitor would reject the trace. For a closed-source application distributed with the monitor, this would lead to unexpected termination on the end user’s system. Both UI actions and input from the network can cause such rejections.

Our fuzzer explores combinations of the selected actions but provides no coverage guarantee. Extending the action set and guiding exploration using coverage feedback are directions for future work. Existing fuzzing approaches for UI actions [46] and network traffic [4] could inform these extensions.

Deployment: standardized instrumentation. Our instrumentation is tailored to each application: annotated libraries for Signal Desktop and DevTools-based runtime wrappers for WhatsApp Web (Section 3.2). A standardized instrumentation interface, provided by vendors, would let monitors like SpecMon attach directly to any conforming application, supporting a “don’t trust, but verify” deployment model. In the browser setting, the monitor could also be deployed in the spirit of integrity checkers such as Code Verify [47]. It could run as a browser extension that observes the application or as a WebAssembly module alongside the application in the page.

Trustworthiness: automated model refinement. Our current toolchain exposes the gap between verification and monitoring models, but does not automate their reconciliation. The simplifications that constitute that gap are only informally justified. This is not unique

to our methodology; it only becomes *visible* here: Tamarin models frequently omit messages or parts of messages, ignore functionality believed orthogonal to the argument (e.g., DDoS protections), etc. Nevertheless, we find that our verification model is much closer to the implementation than prior models, while remaining tractable for Tamarin.

Moreover, a gap between two MSR models is preferable to a gap between a model and its implementation. Most importantly, we can reason about this gap. There is work on formally justifying modeling abstractions specifically on message terms [53]. This does not fully close the gap: their method is not implemented, and the gap is not purely about terms. Still, even term-level abstractions already yield large speedups; mechanized congruence proofs or automatically simplified models would have wider applications.

Scalability: trace rewriting and composable verification. Both Signal Desktop and WhatsApp Web communicate with their servers using WebSockets over TLS. Inside this channel, Signal wraps messages using its Sealed Sender protocol to hide the sender’s identity, while WhatsApp Web establishes the Noise protocol for client–server communication. We found that SpecMon’s trace rewriting mechanism (see Section 3.4) makes it straightforward to layer monitoring models to capture nested communication. Tamarin, by contrast, lacks compositional reasoning, and composition results in comparable tools (e.g., for the applied- π calculus [5]) do not match trace rewriting, though channel-specific results [16] could be adapted.

The existing TLS model [20] is very large. Verifying this model alone already requires about a week of continuous computation. Analyzing the composed system using the current holistic analysis methods is prohibitively expensive, even though monitoring (which we tested for the intermediate layers, e.g., Sealed Sender, not for TLS) seems to handle nesting well.

10 DISCUSSION AND LESSONS LEARNED

Our case studies combined verification-oriented and monitoring-oriented modeling in a single methodology. The lessons below may also apply to other protocols and implementations.

One model for verification and monitoring. Verification and monitoring place different demands on the model. Verification needs enough abstraction to remain tractable, whereas monitoring needs implementation details such as concrete message formats, helper computations, and persistent state. These details can substantially increase verification time. Maintaining a unified model with monitorable and verification variants is therefore harder than performing either task alone, but it provides a shared validation point. Accepted implementation traces are checked against the monitorable variant, and explicit transformations produce the tractable verification variant (Sections 3.5 and 7.1). These transformations are not yet mechanized, so they expose rather than eliminate the remaining abstraction gap. In our experience, this discipline also produces better-structured models. Most of the following lessons improved both monitoring performance and verification time.

Scope rules by their inputs. Rule boundaries are marked by input and freshness facts. We found that a rule should be scoped so that its first observable computation directly uses an input or a freshly sampled value. This criterion is sufficient for our models

and benefited both monitoring and verification. Future SpecMon versions could automate this check.

Avoid accidental nondeterminism. SpecMon natively supports non-deterministic specifications by exploring all applicable rules, which is flexible but costly. At the scale of our case studies, it is easy to introduce nondeterminism *accidentally*. Using SpecMon’s debug output, we identified states in which multiple rules were applicable. We then added differentiating parameters to the relevant facts, which limited exploration to the intended cases.

Scope facts for pruning. The monitor’s core work is matching the current configuration against potential rule applications, so its performance depends on how quickly it can prune facts that cannot match. Pruning on the first argument of a fact is cheaper than pruning on the last. Scoping facts to a role’s identity and local protocol state sped up both monitoring and verification. Across the tested trace lengths, mean peak process memory increases but remains below 54 MiB for all three workloads (Section 6.3).

Distinguish protocol messages from modeling artifacts. Tamarin’s Out facts model communication with the adversary, such as publishing public keys. They are syntactically indistinguishable from protocol messages that the implementation actually sends. The monitor therefore stored these facts initially but never consumed them because the implementation does not emit corresponding events. Marking them with the same macro mechanism used for protocol messages allows the monitor to skip storing them. A model should therefore distinguish messages in the implemented protocol from artifacts of the modeling language.

Instrument chokepoints. A small number of low-level functions can expose a large share of protocol behavior. In WhatsApp Web, the WebCrypto encrypt, decrypt, and sign functions produced events across X3DH, the Double Ratchet, and the Noise transport layer. The sign function also covered hashing. Such chokepoints provide broad coverage with little instrumentation and help locate the remaining protocol-relevant functions in a minified codebase.

Reusable methodology. The instrumentation is reusable. For other Signal-based applications, whether open or closed source, reusing libsignal instrumentation reduces the manual effort. Developing the initial WhatsApp Web model and instrumentation took two person-weeks. Revising the instrumentation, adding fuzzing and out-of-order message support, and rerunning the experiments took one additional person-week.

11 CONCLUSION

We present the first runtime monitoring of the Signal protocol in production messaging applications. By applying SpecMon to both Signal Desktop [62] and WhatsApp Web [71], we demonstrate that runtime monitoring can bridge the verification gap between formal protocol models and real-world implementations.

Our contributions include: (1) the most comprehensive monitorable model of Signal to date, combining the X3DH/PQXDH handshake and Double Ratchet protocols with accurate implementation details, including message formats and the Sealed Sender mechanism; (2) the first formal model of WhatsApp’s Signal-protocol variant, revealing previously undocumented implementation

differences; (3) a methodology for monitoring both open-source (Signal Desktop via annotated libraries) and closed-source (WhatsApp Web via browser DevTools) applications; and (4) empirical evidence that monitoring has low overhead in our measured setting while detecting security-relevant protocol deviations.

Our evaluation shows successful monitoring of session initialization and symmetric and asymmetric ratcheting. The Signal out-of-order example and the WhatsApp workload containing reordered messages also demonstrate monitoring of out-of-order delivery. Fault-injection experiments show that SpecMon detects unexpected network outputs, malformed messages, and incorrect uses of cryptographic libraries. Secrets encoded as ordinary message payloads remain outside this detection capability (Section 9).

For vendors, our work demonstrates a practical path to establish trust in proprietary implementations: provide formal models that serve as executable documentation, integrate runtime monitors to check protocol compliance within the monitored scope, and submit models to rigorous verification—all while maintaining proprietary control over source code. For researchers, the case studies show that the methodology is viable, flexible, and portable: after the Signal Desktop case study, adapting the workflow to WhatsApp Web was mainly a matter of identifying and instrumenting the relevant implementation functions. The main remaining opportunity is automation. Model refinement still requires manual analysis; reducing that effort would make monitorable, verified models easier to maintain as messaging implementations evolve.

ACKNOWLEDGMENTS

Moustafa Said completed this work as a master’s student at Saarland University, Germany. Kevin Morio and Aurora Naska completed this work as members of the Graduate School of Computer Science, Saarland University, Germany.

This paper was edited for grammar with ChatGPT and Claude.

REFERENCES

- [1] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. 2012. Computational verification of C protocol implementations by symbolic execution. In *ACM CCS 2012*. ACM, 712–723.
- [2] José Bacerlar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2013. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *ACM CCS 2013*. ACM, 1217–1230.
- [3] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT 2019*. Springer, 129–158.
- [4] Max Ammann, Lucca Hirschi, and Steve Kremer. 2024. DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing. In *IEEE S&P 2024*. IEEE, 1481–1499.
- [5] Myrto Arapinis, Vincent Cheval, and Stéphanie Delaune. 2012. Verifying Privacy-Type Properties in a Modular Way. In *IEEE CSF 2012*. IEEE Computer Society, 95–109.
- [6] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David Basin, and Peter Müller. 2023. Sound Verification of Security Protocols: From Design to Interoperable Implementations. In *IEEE S&P 2023*. IEEE Computer Society, 1077–1093.
- [7] Hugo Beguinet, Céline Chevalier, Thomas Ricosset, and Hugo Senet. 2023. Formal Verification of a Post-quantum Signal Protocol with Tamarin. In *VECoS 2023*. Springer, 105–121.
- [8] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. 2010. Modular verification of security protocol code by typing. In *ACM POPL 2010*. ACM, 445–456.
- [9] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. 2024. Formal Verification of the PQXDH Post-Quantum Key Agreement Protocol for End-to-End Secure Messaging. In *USENIX Security 2024*. USENIX Association, 469–486.
- [10] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. 2022. A More Complete Analysis of the Signal Double Ratchet Algorithm. In *CRYPTO 2022*. Springer, 784–813.
- [11] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Found. Trends Priv. Secur.*, 1, 1-2, 1–135.
- [12] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. 2010. Attacking and fixing PKCS#11 security tokens. In *ACM CCS 2010*. ACM, 260–269.
- [13] Colin Boyd, Anish Mathuria, and Douglas Stebila. 2020. *Protocols for Authentication and Key Establishment, Second Edition*. Springer.
- [14] David Cadé and Bruno Blanchet. 2015. Proved generation of implementations from computationally secure protocol specifications. *J. Comput. Secur.*, 23, 3, 331–402.
- [15] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. 2022. Universally Composable End-to-End Secure Messaging. In *CRYPTO 2022*. Springer, 3–33.
- [16] Vincent Cheval, Véronique Cortier, and Eric le Morvan. 2015. Secure Refinements of Communication Channels. In *FSTTCS 2015*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 575–589.
- [17] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. 2022. SAPIC+: protocol verifiers of the world, unite! Cryptology ePrint Archive, Paper 2022/741. <https://eprint.iacr.org/2022/741>.
- [18] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2020. A Formal Security Analysis of the Signal Messaging Protocol. *J. Cryptol.*, 33, 4, 1914–1983.
- [19] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. 2016. On Post-compromise Security. In *IEEE CSF 2016*. IEEE Computer Society, 164–178.
- [20] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3. In *ACM CCS 2017*. ACM, 1773–1788.
- [21] Cas Cremers, Charlie Jacomme, and Aurora Naska. 2023. Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations. In *USENIX Security Symposium*. USENIX Association, 1235–1252.
- [22] Cas Cremers, Niklas Medinger, and Aurora Naska. 2025. Impossibility Results for Post-Compromise Security in Real-World Communication Systems. In *IEEE S&P 2025*. IEEE, 4391–4405.
- [23] Cryspen. 2025. libcrux-ml-kem. Retrieved Apr. 27, 2026 from <https://crates.io/crates/libcrux-ml-kem/0.0.7>.
- [24] Alexander Dax, Robert Künnemann, Sven Tangermann, and Michael Backes. 2019. How to Wrap it up - A Formally Verified Proposal for the use of Authenticated Wrapping in PKCS#11. In *IEEE CSF 2019*. IEEE, 62–77.
- [25] Facebook, Inc. 2017. Messenger Secret Conversations: Technical Whitepaper. Technical Whitepaper. Version 2.0. Facebook, Inc., (May 18, 2017).
- [26] Rune Fiedler and Felix Günther. 2025. Security Analysis of Signal’s PQXDH Handshake. In *PKC 2025*. Springer Nature Switzerland, 137–169.
- [27] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. 2016. How Secure is TextSecure? In *IEEE EuroS&P 2016*. IEEE, 457–472.
- [28] Jeffrey Goldberg. 2025. The Trump Administration Accidentally Texted Me Its War Plans. (Mar. 24, 2025). Retrieved Feb. 19, 2026 from <https://www.theatlantic.com/politics/archive/2025/03/trump-administration-accidentally-texted-me-its-war-plans/682151/>.
- [29] Google LLC. 2026. Chrome DevTools Protocol. (Feb. 11, 2026). Retrieved Feb. 11, 2026 from <https://chromedevtools.github.io/devtools-protocol/>.
- [30] Google LLC. 2022. Messages End-to-End Encryption: Overview. Technical Paper. Version 1.2. Google, (Feb. 2022).
- [31] Google LLC. 2025. Protocol Buffers Documentation. Retrieved Feb. 11, 2026 from <https://protobuf.dev/programming-guides/encoding/>.
- [32] Glenn Greenwald and Ewen MacAskill. 2013. NSA Prism Program Taps in to User Data of Apple, Google and Others. <https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data>.
- [33] Guardian. 2025. WhatsApp Messaging App Banned on All US House of Representatives Devices. *The Guardian. Technology*, (June 23, 2025).
- [34] Keitaro Hashimoto, Shuichi Katsumata, and Thom Wiggers. 2025. Bundled Authenticated Key Exchange: A Concrete Treatment of Signal’s Handshake Protocol and Post-Quantum Security. In *USENIX Security 2025*. USENIX Association.
- [35] Dennis Jackson. 2020. *Improving Automated Protocol Verification: Real World Cryptography*. Ph.D. Dissertation. University of Oxford.
- [36] Jan Jürjens. 2008. Using Interface Specifications for Verifying Crypto-Protocol Implementations. In *FIT 2008*.
- [37] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. 2017. Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach. In *IEEE EuroS&P 2017*. IEEE, 435–450.
- [38] Ehren Kret and Rolfe Schmidt. 2024. The PQXDH Key Agreement Protocol. Specification. Version Revision 3. Signal, (Jan. 23, 2024).

[39] Felix Linker, Christoph Sprenger, Cas Cremers, and David A. Basin. 2025. Looping for Good: Cyclic Proofs for Security Protocols. In *ACM CCS 2025*. ACM, 2759–2773.

[40] Benjamin Lipp. 2022. *Mechanized Cryptographic Proofs of Protocols and Their Link with Verified Implementations*. Ph.D. Dissertation. Université Paris sciences et lettres.

[41] Joshua Lund. 2018. Technology Preview: Sealed Sender for Signal. (Oct. 29, 2018). Retrieved Feb. 11, 2026 from <https://signal.org/blog/sealed-sender/>.

[42] Moxie Marlinspike. 2016. WhatsApp’s Signal Protocol Integration Is Now Complete. (Apr. 5, 2016). Retrieved Feb. 11, 2026 from <https://signal.org/blog/whatsapp-complete/>.

[43] Moxie Marlinspike and Trevor Perrin. 2017. The Sesame Algorithm: Session Management for Asynchronous Message Encryption. Specification. Version Revision 2. Signal, (Apr. 14, 2017).

[44] Moxie Marlinspike and Trevor Perrin. 2016. The X3DH Key Agreement Protocol. Specification. Version Revision 1. Signal, (Nov. 4, 2016).

[45] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *CAV 2013*. Springer, 696–701.

[46] Atif M. Memon. 2008. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18, 2, 4:1–4:36.

[47] Meta Platforms, Inc. 2022. Introducing Code Verify, an Open Source Browser Extension for Verifying Code Authenticity on the Web. (Mar. 10, 2022). Retrieved July 21, 2026 from <https://engineering.fb.com/2022/03/10/security/code-verify/>.

[48] Chris Michael. 2025. Photos Reveal Trump Cabinet Member Using Less-Secure Signal App Knockoff. (May 2, 2025). Retrieved Feb. 18, 2026 from <https://www.theguardian.com/us-news/2025/may/02/trump-cabinet-signal-chat-app>.

[49] Sebastian Mödersheim and Georgios Katsoris. 2014. A Sound Abstraction of the Parsing Problem. In *IEEE CSF 2014*. IEEE Computer Society, 259–273.

[50] Kevin Morio and Robert Künnemann. 2024. SpecMon: Modular Black-Box Runtime Monitoring of Security Protocols. In *ACM CCS 2024*. ACM, 2741–2755.

[51] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemat. 2023. CryptoBap: A Binary Analysis Platform for Cryptographic Protocols. In *ACM CCS 2023*. ACM, 1362–1376.

[52] National Institute of Standards and Technology. 2014. CVE-2014-1266. Retrieved Feb. 11, 2026 from <https://nvd.nist.gov/vuln/detail/cve-2014-1266>.

[53] Thanh Binh Nguyen, Christoph Sprenger, and Cas Cremers. 2018. Abstractions for security protocol verification. *J. Comput. Secur.*, 26, 4, 459–508.

[54] [SW] Artiom Pavlov, Tony Arcieri, and Contributors. Rust Crypto 2025. Retrieved Feb. 11, 2026 from <https://github.com/RustCrypto>.

[55] Trevor Perrin, Moxie Marlinspike, and Rolfe Schmidt. 2025. The Double Ratchet Algorithm. Specification. Version Revision 4. Signal, (Nov. 4, 2025).

[56] Nadia Polikarpova and Michal Moskal. 2012. Verifying Implementations of Security Protocols by Refinement. In *VSTTE 2012*. Springer, 50–65.

[57] Albert Qi, Ronak Malik, and Justin Ye. 2024. An Approximate Tamarin Analysis of the X3DH Key Agreement Protocol. Course Project Report. Harvard University.

[58] RustCrypto Developers. 2025. RustCrypto - hkdf. Retrieved Apr. 23, 2026 from <https://github.com/RustCrypto/KDFs/tree/master/hkdf>.

[59] Moustafa Said, Aurora Naska, Kevin Morio, and Robert Künnemann. 2026. From Specs to Apps: Verifying and Monitoring Models of Signal and WhatsApp. In *ACM CCS 2026*. ACM. doi: 10.1145/3830454.3846691.

[60] [SW] Signal Messenger, LLC. Libsignal version 0.87.1. Feb. 6, 2026. Retrieved Feb. 11, 2026 from <https://github.com/signalapp/libsignal>.

[61] [SW] Signal Messenger, LLC. Libsignal’s Storage Protocol Buffer 2025. Retrieved Feb. 11, 2026 from <https://github.com/signalapp/libsignal/blob/bcfa9a7d8ed6cc250bcc8128e06b725c8fe5b4f6/rust/protocol/src/proto/storage.proto>.

[62] [SW] Signal Messenger, LLC. Signal Desktop 2026. Retrieved Feb. 11, 2026 from <https://github.com/signalapp/Signal-Desktop>.

[63] Signal Technology Foundation. 2025. Signal. Retrieved Feb. 11, 2026 from <https://signal.org/>.

[64] Signal Technology Foundation. 2026. Signal | Technical Documentation. Retrieved Feb. 11, 2026 from <https://signal.org/docs/>.

[65] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. 2020. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proc. ACM Program. Lang.*, 4, OOPSLA, 152:1–152:31.

[66] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *ACM ICFP 2011*. ACM, 266–278.

[67] [SW] The Dalek Cryptography Developers. curve25519-dalek 2026. Retrieved Feb. 11, 2026 from <https://github.com/dalek-cryptography/curve25519-dalek>.

[68] Dion van Dam. 2019. *Analysing the Signal Protocol. A Manual and Automated Analysis of the Signal Protocol*. Master’s thesis. Radboud University, Nijmegen, The Netherlands.

[69] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. 2023. Compare: Provably Secure Formats for Cryptographic Protocols. In *ACM CCS 2023*. ACM, 564–578.

[70] WhatsApp LLC. 2025. About End-to-End Encryption | WhatsApp Help Center. Retrieved Feb. 11, 2026 from <https://faq.whatsapp.com/820124435853543>.

[71] WhatsApp LLC. 2026. WhatsApp Web. Retrieved Feb. 11, 2026 from <https://web.whatsapp.com/>.

[72] World Wide Web Consortium. 2025. Web Cryptography Level 2. Retrieved Feb. 11, 2026 from <https://www.w3.org/TR/weccrypto/>.

A OPEN SCIENCE

The companion artifact contains our formal models and trace-rewriting rules, application instrumentation, recorded traces, and the monitor measurements underlying Figures 2a and 2b. It is available at:

<https://doi.org/10.5281/zenodo.19892766>

Experiment scripts and a reproduction environment with pinned source revisions and patches support rerunning verification and recorded-trace monitoring, including new performance measurements. Plotting scripts regenerate the processing-time and memory figures from the archived measurements or new runs. The documentation describes experiment configuration and live monitoring setups. The latter require application accounts and access to the respective services.

B ETHICAL CONSIDERATIONS

Our WhatsApp experiments were conducted exclusively in the client-side environment of WhatsApp Web using personal test accounts. No unauthorized access to user data or Meta infrastructure occurred. These experiments were conducted for academic purposes and comply with Meta’s Bug Bounty Policy guidelines.

We did not identify a concrete bug, security vulnerability, or demonstrable misbehavior in WhatsApp. Our findings on protocol design and feature adoption in WhatsApp and Signal reflect deliberate engineering choices, despite both using libsignal.

Separately, we found and reported to the developers that Signal Desktop crashes after sending 4,348 consecutive messages.

C INSTRUMENTATION DETAILS

Tables 3 and 4 list the instrumented functions for Signal Desktop and WhatsApp Web, respectively.

We implement (1) and (6) by removing the common session either from both session tables or, for (6), only from Parker’s. For (2) and (3), we instrument the sending functionality; for the latter, we simply run it on Parker. For (4) and (5), we manipulate the network delivery functions by inserting a proxy into the code that delays the first message for Signal Desktop. For WhatsApp Web, reordering at the WebSocket proxy layer fails because the client-server channel is protected by Noise and sequence-number checks reject reordered frames. We therefore intercept outgoing messages after Signal-protocol encryption and before Noise transport encryption, at `NoiseSocket.sendFrame`. The instrumentation can hold back a message and release it after later messages. Each released message passes through the original Noise encryption function, so the transport sequence numbers remain valid while the enclosed Signal messages arrive out of order. Our models support out-of-order delivery. The artifact includes the instrumentation and fuzzer actions needed to exercise it.

Table 3: Instrumented functions for Signal Desktop.

Function	Symbolic abstraction	Comment
hmac_sha256 hkdf	$h(x)$ $hkdf(salt, seed)$	Imported from the HKDF module [58].
diffie_hellman	x^y	Curve25519 scalar multiplication [67].
aes_256*_encrypt aes_256*_decrypt	$senc(m, k)$ $sdec(m, k)$	AES-256 CBC/CTR operations, partially from RustCrypto [54].
handle_inner_response send_request	$recv(m)$ $send(m)$	WebSocket module (ws2) wrappers in libsignal.
encapsulate decapsulate	$aenc(ss, pk)$ $adec(ct, sk)$	ML-KEM operations provided by libsignal via libcrux-ml-kem [60, 23].
KeyPair.generate	$rand()$	Randomness generation in libsignal.
verify_signature	$verify(s, m, pk)$	Signature verification in libsignal.

Table 4: Instrumented functions for WhatsApp Web.

Function	Symbolic abstraction	Comment
sign(HMAC, 0^{64} , y) sign(HMAC, x, y)	$h(y)$ $hkdf(x, y)$	Imported from the crypto.subtle module [72].
ecdh(x, y)	x^y	Base-point multiplication on an elliptic curve, imported from the WASignalKeys module [71].
encrypt(m, k) decrypt(c, k)	$senc(m, k)$ $sdec(c, k)$	CBC encryption/decryption from crypto.subtle [72].
verifyMsgSignalVariant(pk, m, s)	$verify(s, m, pk)$	Signature verification via the WASignalSignatures module [71]. Arguments are the public identity key, signed prekey, and signature.
onmessage(m) send(m)	$recv(m)$ $send(m)$	Imported from the WebSocket module [71].
getRandomValues() makeSerializedKeyPair() makeKeyPair()	$rand()$	Randomness generation. Imported from [72] and partially from the WASignalKeys module [71].