Formal Security Analysis of the Olvid Messenger

Noemi Terzo
Max Planck Institute
for Security and Privacy (MPI-SP)
Bochum, Germany
noemi.terzo@mpi-sp.org

Cas Cremers
CISPA Helmholtz Center for
Information Security
Saarbrücken, Germany
cremers@cispa.de

Ruben Gonzalez
Neodyme AG
Berlin, Germany
mail@ruben-gonzalez.de

Peter Schwabe*
Max Planck Institute
for Security and Privacy (MPI-SP)
Bochum, Germany
peter@cryptojedi.org

Yuval Yarom
Ruhr University Bochum
Bochum, Germany
yuval.yarom@rub.de

Zhiyuan Zhang
Max Planck Institute
for Security and Privacy (MPI-SP)
Bochum, Germany
zhiyuan.zhang@mpi-sp.org

Abstract

We perform the first formal security analysis of the cryptographic core of Olvid, an end-to-end encrypted messaging app notably used by French government officials, including ministers. Despite its deployment in sensitive contexts and its role in critical communications infrastructure, Olvid's cryptographic security has received little independent analysis.

To address this gap, we develop detailed models of Olvid's authenticated key exchange and continuous key agreement protocols. We formally verify that our protocol models achieve security properties such as mutual authentication, session-key secrecy, forward secrecy, and replay protection, under an active Dolev-Yao network adversary model that can compromise parties.

While we constructively prove that the protocol design meets core security guarantees, our analysis also reveals that, contrary to its claims, the protocol does not meet strong modern security properties that are met by other state-of-the-art secure-messaging protocols, such as Signal. For example, we show in our formal analysis that Olvid is not secure in modern security models such as eCK. Along the way, we uncover a potential timing leakage, and discuss Olvid's anonymity claims.

Keywords

Cryptographic Protocols; Protocol Verification; Symbolic Verification; Authenticated Key Exchange; Timing Attacks.

1 Introduction

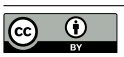
The Olvid secure messenger was released in 2019 and, like other messengers, promises state-of-the-art end-to-end encryption. In 2023, French Prime Minister Élisabeth Borne issued a ministerial circular requiring ministers and their top staff to switch to Olvid [30, 52, 66, 67] by December 8, 2023. In the same memo, she banned her cabinet from using other messaging apps, such as WhatsApp, Signal or Telegram. In December 2023, the then French Minister for Digital Transition and Telecommunications, Jean-Noël Barrot, subsequently called Olvid the *"most secure instant messenger in the world"* [10]. As a result, Olvid is now used by employees of the French Presidency–including French President Emmanuel Macron [58]–and its associated administration (Élysée) and multiple ministries in France. Additionally, Olvid developers have previously mentioned French special forces police and the Ministries of Justice and Defense as clients [29]. Thus, Olvid constitutes critical communication infrastructure for a major NATO member state with nuclear capabilities.

Olvid makes strong security claims, notably stating that it is "the most secure messenger in the world" and that its "security model is utterly game-changing" [55]. Despite the political endorsement of Olvid and its subsequent deployment for sensitive communication, very little effort has so far gone into public security analysis of Olvid.

In 2020 and 2021, Olvid received a level-one security certification ("certification de sécurité de premier niveau", CSPN) of the French government agency ANSSI. The reports pertaining to the ANSSI certification are publicly available on the Olvid website and cover both the Android and iOS clients. The only independent public analyses of Olvid's security are the reports by Synacktiv in the context of the ANSSI certification [77–79], an analysis by Abdalla of Olvid's trust-establishment sub-protocol [1], and an experimental investigation of the post-compromise-security properties of Olvid and other messengers by Cremers, Fairoze, Kiesl, and Naska [25].

This lack of a more complete public analysis of Olvid's core key exchange and ratcheting protocols is somewhat surprising, because unlike many other messenger services including WhatsApp, Skype, Facebook Messenger, and of course Signal, Olvid does not use the Signal protocol, which is thoroughly studied and widely regarded as the "gold standard" for key exchange in secure messaging [3, 17, 21, 32, 34]. Instead, Olvid uses its own suite of protocols for its cryptographic core: the *Channel Creation with Contact Device Protocol* plays roughly the role that X3DH key exchange protocol plays in the Signal protocol, i.e., it is used to agree on symmetric keys between two parties that do not yet have a secure communication channel. These symmetric keys are subsequently used in an *Oblivious Channel* to exchange payloads, which also updates keys through a built-in symmetric ratchet. Periodically or after a number of sends, keys are updated through the asymmetric *Full Ratchet Protocol*. The combination of these two ratchets is akin to Signal's double ratchet, but at the same time technically so different that any results for Signal's protocols have no implications for Olvid's security.

*Also with Radboud University, Nijmegen, The Netherlands.



Contributions. We present the first formal security analysis of the cryptographic core of Olvid. Along the way, we make observations on the Java client source code and its implementation security.

- We formally model the cryptographic core of Olvid in the Tamarin symbolic protocol prover [51]. This model is partly derived from the technical specification of Olvid [8], and for many aspects not covered by this specification we consulted the Olvid source code [56].
- We present computer-verified symbolic proofs of multiple properties of our Olvid models that one typically expects from authenticated key exchange and continuous key agreement protocols, such as session-key secrecy, mutual authentication, forward secrecy, and replay protection.
- We investigate if Olvid's cryptographic core also meets stronger security properties similar to Signal. We formally prove that the Full Ratchet Protocol in principle offers a form of Post-Compromise Security (PCS) for a single ratchet chain, similar to Signal[1]. Moreover, we investigate whether Olvid's cryptographic core also achieves key secrecy after certain combinations of ephemeral and static keys have been compromised. This would be mandated by, e.g., the well established extended Canetti-Krawczyk (eCK) model for authenticated key exchange [46]. Our analysis shows that this is not the case for some key-reveal patterns and is the case for other patterns only under the non-standard assumption that ephemeral public keys do not leak to the adversary.
- To conduct our formal analysis, we inspected parts of Olvid's source code, leading to several additional security observations regarding the implementation. First, we find that Olvid is at several spots assuming certain values to be in a canonical representation without actually enforcing it. As a result of this, signatures do not achieve the strong unforgeability notion. We were, however, not able to escalate this to a full attack. Second, we identify a rather slow, but persistent denial-of-service (DoS) attack vector which stems from the way Olvid implements replay protection. Third, we uncovered a timing leakage in the implementation of elliptic-curve scalar multiplication and present a proof-of-concept exploit on an x86_64 CPU showing that this leakage could potentially be exploitable by an on-device attacker.

We conclude the paper by discussing remaining observations, e.g., on the anonymity properties claimed by Olvid.

Analysis scope. Throughout this work, we use the term *cryptographic core* of Olvid to refer exclusively to its authenticated key exchange and continuous key agreement protocols. Other cryptographic components, such as the group messaging protocols, device handling mechanisms and backup protocols fall outside the scope of our analysis.

Related Work on Olvid. There is currently only one public technical report presenting formal-analysis results for Olvid [1]. The report is very limited in scope and restricted to the short-authentication-strings (SAS) mechanism that is part of Olvid. The core conclusion is that the Olvid-specific changes to the SAS sub-protocol do not invalidate its security arguments, building on Pasini [60]. The certification reports by Synacktiv and ANSSI [77–79] only give a high-level summary of the protocols used in Olvid, but do not provide any formal analysis of these protocols.

Ethical Considerations. We discuss ethical considerations and responsible disclosure in Appendix A.

Artifact Availability. We provide the link to our artifact in Appendix F, together with instructions on how to reproduce our results.

2 Preliminaries

In this section, we provide background on the security models used for the key exchange and ratcheting protocols, to help the reader understand our modeling choices. We also give a brief introduction to the Tamarin prover, in order to facilitate understanding of the rules, lemmas, and restrictions presented in Section 4. Finally, we provide an overview of cache side-channel attacks, with a focus on the `Flush+Reload` technique used in our proof-of-concept attack. For more background on cryptographic primitives, see Appendix C.

2.1 Security models for key exchange

There is a long history of security models for key exchange of ever-increasing strength. Classical notions of key exchange security, such as [13] consider an active adversary that controls the network, and which can reveal selected session keys. The security notion requires the session key of a specific session (often referred to as the Test session) to be indistinguishable from random, even if the adversary can learn the session keys of other sessions. Many subtleties in this domain then arise from the exact definition of "other session": one might expect this to be any session that computes a different session key. However, in most security models for key exchange, the adversary can also reveal the session key of sessions whose variables/messages differ from the Test session. Effectively, this models a form of implicit authentication.

Later security models for key exchange give the adversary the ability to reveal long-term keys of some participants. This allows for modeling perfect forward secrecy and identity misbinding attacks, by considering that not all participants might be honest, or could be compromised in the future.

Newer key exchange models such as CK [19] and eCK [46] also consider compromise of the randomness of specific sessions.

2.2 Security models for CKA

While standard models for key exchange consider stateless protocols that start from asymmetric key pairs and produce a shared session key, the notion of CKA (*continuous* key agreement) [3] starts from a shared symmetric key and produces a new shared key. This primitive was introduced in [3] as part of an attempt to decompose the Signal protocol design into building blocks. The corresponding security model is relatively simple, as the adversary is assumed to be passive.

[1]Note however that already [25] showed that protocol-level PCS security does not extend to conversation level, either in Signal, or in Olvid.

2.3 The Tamarin prover

There is a large body of literature analyzing and formalizing real-world security protocols; see, e.g., [3, 4, 21, 62, 70, 76, 80] for some examples. During the last decade, formal symbolic analysis has emerged as a technology that can aid analyzing such complex systems, both for attack-finding and establishing security guarantees, notably using the Tamarin prover [11, 12, 20, 26, 27, 48, 51].

For our formal analysis, we use the Tamarin prover [71], which is a state-of-the-art tool for the automated analysis of security protocols. Tamarin performs symbolic analysis based on the Dolev-Yao model, where cryptographic primitives are treated as "perfect" black boxes. Cryptographic primitives and their properties are modeled using a term algebra with an equational theory. Tamarin takes the protocol to analyze, the adversarial capabilities that define the threat model, and the security properties, and verifies the properties or provides counterexample traces that constitute attacks on the properties. Both the protocol and the adversary are defined using *multiset rewriting rules*, which induce a labeled transition system. A *rule* has a premise, actions and a conclusion, which in turn are multisets of so-called facts:

```
[ PremiseFact1(.), PremiseFact2(.), ... ]
--[ ActionFact1(.), ActionFact2(.), ... ]->
[ ConsequenceFact1(.), ConsequenceFact2(.), ... ]
```

Rules model transition steps of protocol role instances. For example, a simple challenge-response protocol would typically be modeled by three rules: one to model an initiator generating and sending a challenge, one to model a responder receiving a message and computing and sending a response, and one to model an initiator receiving the response to an earlier challenge.

Formally, rules modify the global system state by adding or removing facts. A *fact* is a user-defined symbol *FactName*(*t1*, *t2*, ..., *tn*) that contains *terms ti* as arguments. Facts can be linear, meaning they are consumed upon rule execution, or persistent (denoted by a ! prefixed to the fact name), remaining in the system indefinitely. Each concrete execution of the transition system is a sequence of states with transitions labeled with rule instances, and models an unbounded number of protocol sessions. For each execution, the corresponding sequence of action facts is called a *trace*.

Security properties are expressed as first-order logic formulas over traces, referring to the actions of the protocol rules and quantified on timepoints and terms. Timepoint variables are prefixed by the # symbol. To refine the analysis by filtering out traces that do not satisfy specific conditions, it is possible to utilize *restrictions*, which are modeled in the same way as the security properties and allow Tamarin to focus on relevant traces. Within this transition system, the protocol and adversary interact by exchanging messages over the adversary-controlled network.

2.4 Cache side-channel attacks

Modern processors rely on caches, small fast memories storing frequently accessed data, exploiting the spatial and temporal locality of the software. Multiple works observed that sharing caches between workloads leaks program behavior across security boundaries, from cryptographic code [59, 63, 83], user activity [41, 75], and other software [18, 73, 82].

We focus on the `Flush+Reload` [83], a popular attack technique for creating and observing timing differences [41, 42, 64, 69]. The adversary evicts a target cache line, waits for the victim to execute, then measures the access time to that line: fast access indicates that the line is cached, presumably due to victim access. Repeating this reveals the victim's access pattern to the monitored line. While traditionally demonstrated on Intel, recent work by Lin et al. [47] extends such memory-sharing attacks to Arm processors.

The `Flush+Reload` technique has been used for attacks on elliptic curves [14, 31]. While the Montgomery ladder implementation [53] used in Olvid is considered more resilient to timing attacks than others [44, 54], past work has demonstrated that it is not always secure [38, 39]

3 The Olvid encrypted messenger

Like other prominent secure messengers such as Signal, Olvid offers end-to-end encrypted one-to-one and group communication, and supports multiple devices per user. A notable difference is registration: rather than a phone number, Olvid users provide only basic information such as their name, with all other details optional. Upon account creation, the app generates the client-side cryptographic keys, including the long-term KEM key pair used to encrypt messages sent via the Asymmetric Channel (a core component described in Section 3.2), and the signing-verification key pair.

To establish a contact relationship, users go through an authentication process. Unlike Signal, the free version of Olvid requires users to add contacts through out-of-band channels and to run the SAS-based authentication protocol [61], authenticating peers without relying on a centralized directory. This protocol shares cryptographic tokens via QR codes or URLs that serve as unique identifiers for peer authentication, consisting of a URL to the Olvid backend server, the verification key, and the long-term KEM public key; they include no user-related metadata such as usernames or phone numbers. In contrast, the paid enterprise version uses a centralized contact-discovery system based on the open-source Keycloak identity and access management solution [57].

Once authenticated, users establish a secure communication channel, the Oblivious Channel, using the Channel Creation with Contact Device (CCCD) Protocol, an AKE protocol that is also run whenever a user associates a new device to their account. The Oblivious Channel is based on a ratchet mechanism similar to Signal's double ratchet, and encrypts all application and protocol messages exchanged by Olvid clients. All messages exchanged by users are sent encrypted through the Olvid server, which is hosted by AWS; client-server communications are secured by TLS 1.3, with TLS 1.2 as a fallback. To renew the root keys used in the Oblivious Channel, the Full Ratchet Protocol (FRP) is run.

In this work, we focus on the cryptographic core of Olvid, and assume that the out-of-band exchange of public-key material and authentication was successfully completed. This assumption is justified by Abdalla's prior analysis of the SAS-based protocol [1]. Specifically, we assume that Alice has Bob's authentic public-key package containing his verification key and long-term KEM public key, and vice versa.

Note that Olvid's terminology deviates from conventional cryptographic meaning. For example, the Oblivious Channel does not refer to Oblivious RAM (ORAM) or Oblivious Transfer (OT), and the ratchet mechanism does not follow the Signal protocol exactly. The Asymmetric and Oblivious Channels are not traditional cryptographic protocols, but rather protocols that encrypt and authenticate message keys via an Authenticated Encryption scheme: the Asymmetric Channel uses a KEM-based hybrid encryption scheme to encrypt messages to the receiver's public key, while the Oblivious Channel uses symmetric keys that evolve through two ratchet protocols. Both the CCCD Protocol and FRP output seeds for the ratchet rather than the final symmetric keys themselves.

3.1 Threat model

Secure messaging systems are typically evaluated against an active network adversary with extensive capabilities. This adversary is capable of intercepting, reading, modifying, reordering, replaying, and injecting messages, as well as impersonating users by sending new messages to any participant in the system. Additionally, the model must account for the possibility of malicious insiders, meaning that some users within the system may act dishonestly or attempt to subvert security guarantees. Furthermore, the server is assumed to be untrusted, meaning it could be compromised, malfunction, or collude with an adversary to extract metadata or manipulate communications. In such a scenario, a meticulously designed secure messaging system should prioritize minimizing reliance on the server for security-critical operations and ensure confidentiality, integrity, and authenticity solely through cryptographic mechanisms. Olvid explicitly claims to provide post-compromise security and forward secrecy, which suggests that its cryptographic design must be resilient against key compromise at some point in time. This includes protection against compromised randomness, ensuring that even if an adversary gains temporary access to encryption keys, they cannot decrypt past or future messages. However, since Olvid does not provide a formal threat model in its documentation or white papers, we will use the threat model above, which aligns with existing security research, to analyze its security guarantees.

3.2 The cryptographic core of Olvid

The starting point for the cryptographic core of Olvid as considered here is that two users **A** and **B** have obtained each other's authentic long-term key packages containing KEM public keys and signature verification keys. Hence, **A** starts with state

$$\gamma_A = (kempk_A,\ kemsk_A, signpk_A,\ signsk_A, kempk_B,\ signpk_B)$$

and **B** starts with state

$$\gamma_B = (kempk_B,\ kemsk_B, signpk_B,\ signsk_B, kempk_A,\ signpk_A).$$

High-level view. Olvid uses two main protocols, the Channel Creation with Contact Device Protocol and the Full Ratchet Protocol, and two sub-protocols, the Asymmetric Channel and the Oblivious Channel. To bootstrap, parties **A** and **B** first execute the CCCD Protocol, which is an authenticated key exchange, and agree on shared symmetric-key seeds s_A and s_B. The CCCD Protocol uses the Asymmetric Channel as a subroutine in its first part; in its second part, once shared secrets are established, it performs key-confirmation that uses the Oblivious Channel as subroutine. The symmetric key seeds s_A and s_B are subsequently used and updated by the Oblivious Channel for payload encryption; s_A is used and updated for messages sent from **A** to **B**; s_B is used and updated for messages sent from **B** to **A**. Informally, the Oblivious Channel includes a symmetric ratchet. Furthermore, when at least 100 messages have been sent by a party or when a timeout of one week is reached since the last execution of the Full Ratchet Protocol, the Full Ratchet Protocol is executed to derive a new key seed for the Oblivious Channel, and can be informally regarded as an asymmetric ratchet[2]. Fig. 1 depicts this high-level view as a diagram.

Asymmetric Channel. Olvid's Asymmetric Channel is a public-key encryption (PKE) scheme, built from a KEM and from an authenticated encryption (AE) scheme. The KEM is instantiated with the ECIES-KEM construction [74], a Diffie-Hellman-based KEM that is IND-CCA-secure. For the Diffie-Hellman routine, Olvid uses Curve25519 [15] but the source code also includes an implementation using the Million Dollar Curve (MDC) [5]. For the authenticated encryption, Olvid adopts the Encrypt-then-MAC approach, using AES-256-CTR combined with HMAC-SHA-256.

The Olvid specification describes this PKE scheme as a unidirectional communication channel, so we use corresponding notation in Fig. 2. Note that the Asymmetric Channel is not following the standard KEM/DEM construction [74]: rather than using k_1 to encrypt the message m, it first encrypts the message key mk_A with k_1 and then encrypts m with that message key.

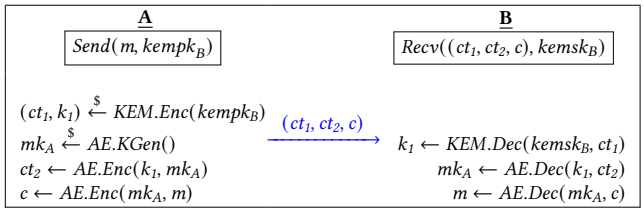


Figure 2: The Asymmetric Channel protocol

When the Asymmetric Channel is used as a subroutine of higher-level protocols, we will use notation

$$\xrightarrow[\text{Asymmetric Channel}]{m}$$

to mean that party **A** executes $Send(m, kempk_B)$ to encrypt message m and send the ciphertext c to **B**, followed by party **B** executing $Recv(c, kemsk_B)$ to receive c and decrypt to m.

Oblivious Channel. Olvid's Oblivious Channel is essentially an authenticated-encryption scheme with symmetric self-ratcheting of the key. Again, the authenticated encryption uses AES-256-CTR with HMAC-SHA-256. Additionally, the Oblivious Channel uses a PRNG and a KDF, which are instantiated with HMAC_DRGB as specified in [9], with SHA-256 as hash function.

[2]These are the thresholds given in the Olvid specification [8]; the Android implementation in the analyzed version 3.6 uses 500 messages and 24h as thresholds.

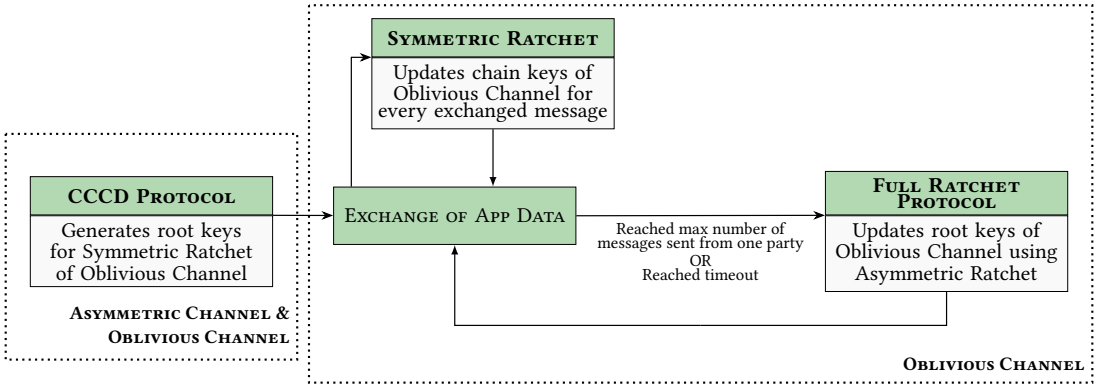


Figure 1: High-level view of the Cryptographic Core of Olvid. The CCCD Protocol is triggered when a user adds a new device or initiates a new session with a contact.

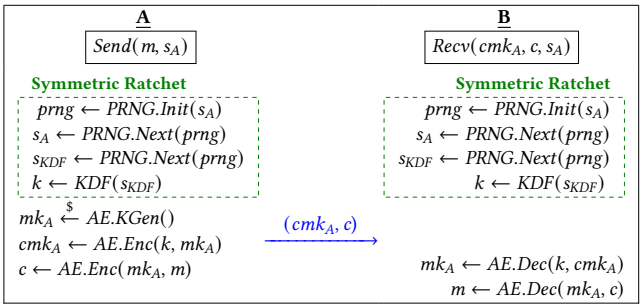


Figure 3: The Oblivious Channel protocol with the symmetric-ratchet component marked in green

Just like for the Asymmetric Channel, we use unidirectional-channel notation in Fig. 3. Note that there are multiple levels of "encryption indirection" in the Oblivious Channel. First, the key seed s_A is used as secret input to the PRNG to derive an updated key seed s_A and another seed s_{KDF}. This seed is then used as input to the KDF to generate a symmetric key k, which is used to encrypt the message key mk_A, which in turn is used to encrypt the message m.

For the Oblivious Channel usage, we similarly write

$$\xrightarrow[\text{Oblivious Channel}]{m}$$

to mean that party **A** executes $Send(m, s_A)$ to encrypt message m and send the resulting ciphertext c to **B**, followed by party **B** executing $Recv(c, s_B)$ to receive c and decrypt to obtain m.

Channel Creation with Contact Device Protocol. This protocol implements an authenticated key exchange between two parties A and B who have already obtained each other's authentic long-term KEM and signature keys. The output of the protocol are symmetric-key seeds s_A and s_B. Prior to executing the protocol, party A has associated a new device with their Olvid account and party B has run the contact discovery protocol and found that there is no Oblivious Channel between their device and A's new device. This generates a local message on B's client, containing A's identity id_A and device identifier dev_A. This message triggers the execution of the

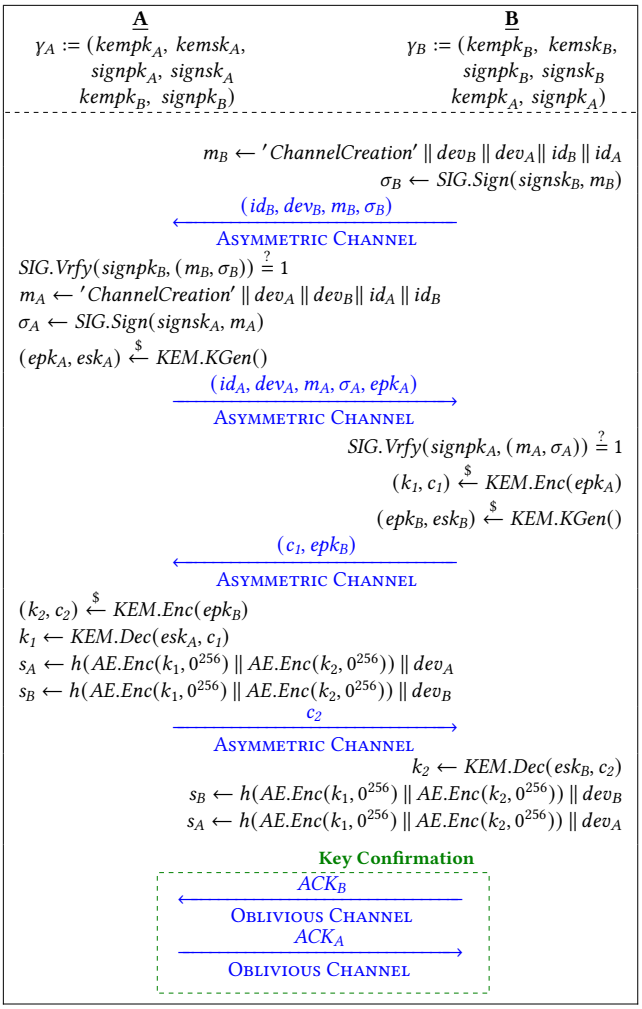


Figure 4: High-level overview of the CCCD Protocol

CCCD Protocol by B, provided that B has A in their contact list

and A is active. The KEM is again instantiated with ECIES-KEM on Curve25519, the signature algorithm SIG is the Edwards-Curve Digital Signature Algorithm (EdDSA) [16] over the MDC curve. The hash function h used in s_A and s_B is SHA-256.

We show a simplified version of the protocol in Fig. 4, and show in Appendix E an extended version incorporating implementation-specific modeling details. The first two exchanged messages of Fig. 4 are called *Ping* messages. The strings ACK_A and ACK_B exchanged in the key-confirmation phase contain additional information about the participants, e.g., an encoding of their profile picture. As it is the case for the Full Ratchet Protocol, every exchanged protocol message has a different message type identifier.

Full Ratchet Protocol. The Full Ratchet Protocol exchanges fresh key material through an already established Oblivious Channel to update the symmetric-key seed s_A (or s_B if roles are reversed). Together with the symmetric ratchet that is part of the Oblivious Channel, the protocol can be seen as Olvid's version of the double ratchet in Signal. While the symmetric ratchet alone would be sufficient to achieve forward security, the Full Ratchet Protocol is required for post-compromise security. All cryptographic primitives in the Full Ratchet Protocol are instantiated as in the CCCD Protocol. We show the protocol in Fig. 5.

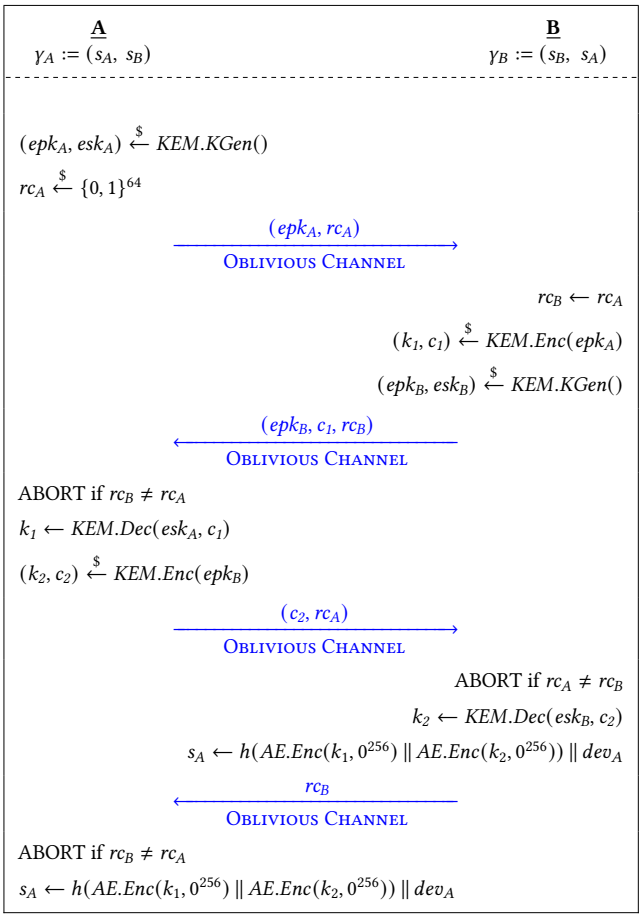


Figure 5: High-level view of the Full Ratchet Protocol

If A aborts the protocol after B has already updated s_A, this does not cause a desynchronization: A will simply reinitiate the Full Ratchet Protocol from scratch by sending the first message, and B will still be able to decrypt it because B retains a set of older seeds.

4 Formal analysis using Tamarin

In this section we first explain how we model the cryptographic core of Olvid in Tamarin. We then use this model to prove many security properties and show that other stronger modern properties do not hold.

4.1 Formally modeling Olvid

Translating a real-world protocol into an abstract model is an inherently challenging process. It requires an in-depth understanding of the protocol's design and implementation, as well as an evaluation of the extent to which the implementation aligns with the design. Moreover, any formal model inevitably involves a number of modeling choices that are dictated by the requirements and limitations of the verification methodology adopted.

Our decision to closely inspect the source code is motivated by the need to bridge the gap between abstraction and reality as much as possible, thereby providing users of the application with confidence that our models accurately reflect the behavior of the deployed system, whether when establishing a new conversation or when updating the keys used for exchanging messages securely.

In this subsection, we not only present the technical details of how we modeled the protocols and their subcomponents in Tamarin, but also offer insights that extend beyond the Olvid model and may be applicable to other protocols modeled adopting the same methodology or others. To describe the modeled parts, we follow the same bottom-up approach that we used to build our models. First, we provide a description of the primitives and channels, and then, at the end, a description of the protocols.

Primitives. As the implementation makes use of malleable signatures, we use the malleable signature models from [43], which allow for multiple distinct signatures to be valid for the same message and signing key. For authenticated encryption, we use Tamarin's built-in encryption model (see [24]). For KEMs, we use a variant of the model used in [48], which we adapt for our threat models.

Asymmetric Channel. Modeling the Asymmetric Channel and Oblivious Channel was one of the most challenging and interesting parts because of the non-standard nature of such channels. After trying several approaches, we identified two main ways to model them in Tamarin. The first consists of inlining the operations of encrypting and decrypting the sent and received messages within the rules of the protocol communicating parties. This means that if, for example, party A wants to send a message m to party B via the Asymmetric Channel, A will perform the operations of encryption of the Asymmetric Channel within their rule and then will send the resulting record via the standard channel (accessible to the adversary). In order to build the record, we use a macro that has the following structure:

```
AsymChanRecord(actor, peer, devActor, devPeer, kemPkPeer, k, rand,
      msgKey, msg) = <<actor, peer, devActor, devPeer, <encaps(k,
     kemPkPeer, rand), senc(msgKey, k)>>, senc(msg, msgKey)>
```

When a party `actor` sends a message `msg` to a party `peer` through the channel established between `devActor` and `devPeer`, it follows the `Send` procedure shown in Fig. 2. In the macro, the record includes additional header fields: `actor`, `peer`, `devActor`, `devPeer`. The component `<encaps(k, kemPkPeer, rand), senc(msgKey, k)>` corresponds to cmk_A in Fig. 2. Note that, following the standard notation used in Tamarin, the arguments of the encryption function appear in reversed order compared to conventional representations.

The second approach, which is the one we ultimately followed, is to consider the channel as an atomic entity, and introduce additional rules to model its behavior. These rules are `AsymmetricChannel_send` and `AsymmetricChannel_recv`, and they model the sending and receiving of a message via the Asymmetric Channel respectively. The former takes as input (i.e., on the left-hand side of the rule) the long-term KEM public key of the receiver, the `AsymChanSend($actor, $peer, $devActor, $devPeer, msg)` fact, and randomly generates the nonce and message key needed to encrypt the message. Then the rule builds the record and sends it out through the standard channel. Similarly, the `AsymmetricChannel_recv` rule is executed when a well-structured record is received via the standard channel and the rule computes the decryption of the message, giving on the right-hand side of the rule the `AsymChanRecv($actor, $peer, $devActor, $devPeer, msg)`. In this way, when a user sends a message using the Asymmetric Channel, they have the `AsymmetricChannel_send` fact on the right-hand side of their rules, and when they receive a message via this channel, they have the `AsymmetricChannel_recv` fact on the left-hand side of the rule. This mechanism allows the state to be passed between rules.

Oblivious Channel. Similarly to the Asymmetric Channel, we first opted for inlining the computation of the record sent via the channel in the rules of the communicating parties, using the macro:

```
OblvChanRecord(actor, peer, devActor, devPeer, seedActor, msg,
      msgKey) = <<actor, peer, devActor, devPeer, senc(msgKey, hkdf
      (PRNG(PRNG(seedActor, '1'), '2'), 'authEncKey'))>, senc(msg,
      msgKey)>
```

This models the encryption mechanism in Fig. 3. The labels `'1'` and `'2'` match the corresponding calls to *PRNG.Next* in the figure. Here, `seedActor` corresponds to s_A before being ratcheted and `hkdf(PRNG(PRNG(seedActor, '1'), '2'), 'authEncKey')` corresponds to k. The symmetric ratcheting of chain key seeds is handled directly within the fact representing the parties' states, $A(\cdot)$ and $B(\cdot)$: when a party A sends a message, their local Oblivious Channel state is updated from `A(A, B, devA, devB, ..., seedA, seedB, counter)` to `A(A, B, devA, devB, ..., PRNG(seedA, '1'), seedB, counter+1)`. Similarly, when the receiving party B processes the message, their Oblivious Channel state changes from `B(B, A, devB, devA, ..., seedB, seedA, counter)` to `B(B, A, devB, devA, ..., seedB, PRNG(seedA, '1'), counter)`. This mechanism ensures that seed values are ratcheted in synchrony with message transmission and reception, while maintaining the correct ratcheting number.

However, the solution we preferred and ultimately adopted is the one in which the channel is seen as a separate entity with its own rules: `ObliviousChannel_send` and `ObliviousChannel_recv`. The strategy is analogous to that of the Asymmetric Channel, with the difference that in the Oblivious Channel the user keeps the ratcheting state, and passes the seed needed to perform the encryption of the message to `OblChanSend($actor, $peer, $devActor, $devPeer, seedActor, msg)` and `OblChanRecv($actor, $peer, $devActor, $devPeer, seedPeer, msg)`, the first to encrypt the message to send, the second to decrypt the message received.

Channel Creation with Contact Device Protocol. In line with Olvid's specifications, we have modeled the protocol using seven rules, each of them corresponding to one step of the protocol. To understand this better, the protocol can be seen as a state machine, and each rule models how a party transitions from one state to another in the system. For example, looking at Fig. 4, the first step for party B consists of computing the message m_B and its signature σ_B, and sending the identifiers id_B and dev_B together with m_B and σ_B to party A via the Asymmetric Channel. In Tamarin, we translate this first step into a rule named `CCCDP_B_1`, which performs these operations and enables party B to proceed with their execution of the protocol once they have received a message back from party A, as shown in Fig. 4.

We use three additional rules to set up the long-term keys of each party and the role played by each party in executing the protocol.

Our model closely follows the more detailed, implementation-based protocol diagram provided in Appendix E. One interesting aspect of our model is how we handle the complex preconditions and checks present in Olvid's code. In Tamarin's modeling language, some preconditions for rules to be instantiated can be expressed by so-called restrictions, i.e., a rule can be *restricted* to only be enabled if a certain condition is met or some action previously happened. In Olvid's implementation, before party B starts executing the protocol, they delete any existing instance of the same protocol between the same parties and devices, as well as any Oblivious Channel already established between their device and A's device. To accurately model this behavior, we define the restrictions named `No_protocol_instance_exists` and `No_oblivious_channel_exists`:

```
restriction No_protocol_instance_exists:
 All A B devA devB #t.
  OnlyIfNoProtocolInstanceExists(A, B, devA, devB) @t
 ==>
  not(Ex #r p. ProtocolInstance(p, A, B, devA, devB) @r & #r<#t)


restriction No_oblivious_channel_exists:
 All A B devA devB #t.
  OnlyIfNoObliviousChannelExists(A, B, devA, devB) @t
 ==>
  not(Ex #r. CreateObliviousChannel(A, B, devA, devB) @r & #r<#t)
```

The first restriction models that when a rule containing the action `OnlyIfNoProtocolInstanceExists` is executed, the implementation checks that no instance of the same protocol between the same two parties and devices was running beforehand. The second restriction models that if a protocol step contains the action `OnlyIfNoObliviousChannelExists`, the implementation checks that no previous protocol execution reached the point where an Oblivious Channel was already generated by a party playing the role of `A` with `B`, using the same devices `devA` and `devB`. These checks also apply when `A` performs the corresponding *DELETE* operations.

With the following restriction, we model the *ABORT* behavior triggered by the reception of a signature that was already received previously:

```
restriction uniqueness_signature:
  All A B sigB #t.
    OnlyIfNoSameSignatureReceived(A, B, sigB) @t
 ==>
    not(Ex #r. StoreSignature(A, B, sigB) @r & (#r < #t))
```

This ensures that whenever a protocol trace includes a rule that contains the action `OnlyIfNoSameSignatureReceived`, there exists no prior step in the execution trace in which the same signature was stored by the receiving party, similar to the checking condition in the implementation.

In the Olvid implementation, received KEM ciphertexts undergo validation, and malformed inputs result in a *DecryptionException* and subsequent abort. To reflect this behavior, we model decapsulation checks using explicit deconstructors, in combination with the embedded restriction on the `ConditionCiphertext` predicate:

```
ConditionCiphertext(ctx)
<==> Ex k pk r. ctx = encaps(k, pk, r)
```

This models that decapsulating arbitrary malformed data will cause an abort, but valid encapsulations of arbitrary keys with arbitrary randomness will not cause an abort.

Full Ratchet Protocol. Following Olvid's specifications, we model the protocol from Fig. 5 using five rules: three for parties playing the role of A and two for parties playing the role of B. In this setting, A's root key is the key to update via the protocol. The execution of the protocol begins with two initialization rules, `FRP_Init_1` and `FRP_Init_2`. These rules ensure that the Oblivious Channel is synchronized on both sides before any further steps are taken. Then, we add two other rules named `FRP_SendMsg` and `FRP_RecvMsg`, which do not model steps of the Full Ratchet Protocol but are necessary to prove post-compromise security, as they allow the parties involved in the protocol to exchange messages after having healed.

Adversarial capabilities. As mentioned in the introduction, there is a strong difference between the technical security claims in [6], which are relatively weak, and informal claims in [55], which are extremely strong. To cover this, we both model the weaker technical claims about forward secrecy as well as the eCK security model, which is more representative of the strong informal claims. In the eCK setting, the adversary can corrupt long-term keys, such as KEM keys and signing keys, and reveal session keys (in our case, it can reveal session seeds and then retrieve the corresponding session keys). It can also reveal ephemeral KEM keys and the randomness values used in *KEM.Enc*. As the adversary is assumed to have full control over the network, it can intercept, modify, delete, and inject arbitrary messages. The model further allows the adversary to attempt Key Compromise Impersonation (KCI) attacks, whereby knowledge of a party's long-term key is exploited to impersonate other parties to that compromised party. Due to the fact that signatures in the implementation are malleable, in the CCCD Protocol the adversary is also able to replace valid signatures, as well as send crafted *Ping* messages to trigger replay attacks. Additionally, in the Full Ratchet Protocol, we consider the adversary capable of compromising either the sender's seed or the receiver's seed used to ratchet the chain keys.

We give the adversary all of these capabilities. When proving specific security properties, we selectively exclude certain adversarial capabilities by classifying the corresponding attacks as trivial within the rules of our model. For instance, when proving forward secrecy, which focuses on the compromise of long-term keys, we treat the compromise of ephemeral keys as a trivial attack pattern, and exclude all execution traces in which such a pattern occurs.

4.2 Verified security properties

This subsection contains all the security properties that we have successfully verified with the Tamarin prover.

Authentication. For both the CCCD Protocol and Full Ratchet Protocol, we model authentication between entities in Tamarin as an *injective agreement* [50]. This property captures that the intended participants engage in the protocol and that they share a consistent view of the roles and agreed-upon terms.

In Tamarin's syntax, such formal properties are specified as lemmas to be proven, and we show an example of the mutual injective agreement below. The final conjunct of the lemma is the injectivity condition, and it guarantees that each protocol instance of party A is uniquely matched to a single instance of party B. Informally, this injectivity corresponds to a uniqueness that models replay protection.

```
lemma mutual_authentication:
  All actor peer devActor devPeer pidActor k1 k2 #i.
    Commit(pidActor, actor, peer, devActor, devPeer, k1, k2) @i &
    Role(pidActor, <actor, peer, devActor, devPeer, 'B'>) @i &
    not(Ex p #r. RevKemKeys(p) @r & #r < #i)
    ==>
    Ex pidPeer #j.
        Running(pidPeer, peer, actor, devPeer, devActor, k1, k2)
              @j &
        Role(pidPeer, <peer, actor, devPeer, devActor, 'A'>) @j &
        #j < #i
    & not(Ex #t2.
        Commit(pidActor, actor, peer, devActor, devPeer, k1, k2)
              @t2 &
        not(#i=#t2))
```

The presentation above slightly simplifies the formal property in the artifact to provide more intuition.[3] The formula states that if an actor, executing in the role of party B, computes the keys *k1* and *k2* and no adversary has, prior to this computation, obtained any long-term KEM private key of any agent nor any ephemeral KEM private key associated with any session, then there must exist a peer, in the role of party A, such that the peer was already active prior to the actor computing the keys.

Forward secrecy. In the CCCD Protocol and Full Ratchet Protocol, Olvid asserts that it provides perfect forward secrecy (FS). In Tamarin, we developed a series of secrecy lemmas, with varying levels of strength, to elucidate the protocols' limitations related to FS. The analysis results are positive with respect to the standard notion of forward secrecy, in which the only patterns that lead to the disclosure of the session seed involve the disclosure of the long-term KEM keys of the participants, prior to the establishment of the seed by them. Additionally, we successfully prove a stronger

[3] The actual property analyzed includes a more complex threat model which leads to more complicated "freshness conditions".

definition of secrecy, which considers the additional adversarial capability of revealing the seed of the considered session or of another session that has the same seed:

```
lemma forward_secrecy:
 All actor peer devActor devPeer sidActor seed #i1 #i3.
  SessionSeedEstablished(sidActor, actor, peer, devActor, devPeer,
        seed) @i1 &
  Role(sidActor, <actor, peer, devActor, devPeer, 'A'>) @i1 &
  K(seed) @i3 &
  (All sid #i. RevRandomnessKem(sid) @ i  ==> F)
  ==>
  (Ex sid #i4. SessSeedRev(sid, seed) @i4) |
  (Ex #i4. RevKemLtk(actor) @i4 & #i4 < #i1) |
  (Ex #i4. RevKemLtk(peer) @i4 & #i4 < #i1)
```

However, we observe contrasting outcomes when considering security in stronger security models, as we describe in Section 4.3.

Post-compromise security. The Full Ratchet Protocol offers a set of security properties, one of which is Post-Compromise Security (PCS) [22]. PCS is a form of security that may be achieved after a device has been compromised. This is possible if the adversary remains passive for a short time after the compromise, which may allow the compromised party to "heal" by exchanging new ephemeral keys to bootstrap future secure communications.

We model a CKA-based [3] formulation of PCS: if the adversary is passive with respect to a specific run of the Full Ratchet Protocol (i.e., allowing the ratchet to heal), even if the adversary can compromise long-term secrets but not the specific seed or ephemerals of the current run, then the new seed is secret. We formally model this security property in Tamarin using the following lemma.

```
lemma PCS:
 All pidActor pidPeer actor peer devActor devPeer seedActor
      seedActor2 #t #t1 #t2 #t3 #t4.
  HealRecvSeed(pidPeer, peer, actor, devPeer, devActor, seedActor)
       @ t1 &
  HealSendSeed(pidActor, actor, peer, devActor, devPeer, seedActor
       ) @ t2 &

  not(Ex #l. AttackerCompromise() @l & #l < #t2 ) &
  not(Ex pid seed #l. SessSeedRev(pid, seed) @l) &
  not(Ex pid #l. RevEphAny(pid) @l) &

  UpdateSendSeed(pidActor, actor, peer, devActor, devPeer,
       seedActor2) @ t3 &
  UpdateRecvSeed(pidPeer, peer, actor, devPeer, devActor,
       seedActor2) @ t4 &

  K(seedActor2) @ t &

  ((#t2 = #t3) | (#t2 < #t3)) &
  ((#t1 = #t4) | (#t1 < #t4))
  ==>
  ((Ex #i. CompromiseSendSeed(actor, peer, devActor, devPeer) @ i
       & (#t2 < #i )) |
   (Ex #i. CompromiseRecvSeed(peer, actor, devPeer, devActor) @ i
        & (#t2 < #i)))
```

The verification confirms that the Full Ratchet Protocol can effectively heal the ratcheted session. However, this guarantee only holds for the specific session, and not for sessions that get started afterwards (possibly by an attacker that obtained long-term keys) as previously noted [25, 28].

Note that Olvid's session-specific PCS property could have been strengthened: Olvid's ratchet mechanism does not "accumulate" entropy across successive ratchets; the new seed replaces the old seed. This means that compromising the ephemeral keys $(K1, K2)$ of the current session would always compromise the next seed. This would be avoided if Olvid were to chain the previous seed into a PRF-PRNG-like construction [3] as Signal does.

Replay protection. In the CCCD Protocol, Olvid ensures replay protection by embedding a signature within the *Ping* message. Upon receiving this message, a participant checks whether the signature has been previously received. If it has, the protocol is aborted; otherwise, the signature is stored to prevent future replays. To verify the effectiveness of this strategy, we modeled a replay protection lemma in Tamarin. Our analysis confirms that the approach works but also highlights certain drawbacks, which we discuss in Section 5. Following Olvid's source code, we model malleable signatures.

The replay protection lemma specifies that once a participant receives and stores a signature associated with a specific pair of devices, any forged version of the same message's signature created by an adversary cannot be stored. In other words, only valid signatures are accepted and retained.

```
lemma replay_protection:
 All pidActor actor peer devActor devPeer msg sig1 #t1 #t2.
  Ping(pidActor, actor, peer, devActor, devPeer, sig1) @ t1 &
  StoreSignature(peer, actor, msg, sig1) @t2
  ==>
  not(Ex sig2 #t3. StoreSignature(peer, actor, msg, sig2) @t3 &
        not(sig1=sig2))
```

KCI resistance. Key Compromise Impersonation (KCI) considers the scenario where an attacker compromises a party X's long-term key, and then exploits the protocol to impersonate as an arbitrary party towards X. The below property models this by checking for agreement while allowing compromise of the actor, which is reflected in the fact that the property should hold *unless* the long-term key of the peer is compromised. Agreement should thus even hold when the actor is compromised to ensure the absence of KCI attacks. We successfully proved this property with respect to an adversary that can reveal long-term keys but not short-term secrets.

```
lemma KCI_resistance:
 All pidActor actor peer devActor devPeer k1 k2 #t1.
  Commit(pidActor, actor, peer, devActor, devPeer, k1, k2) @t1 &
  not(Ex #t. RevKemLtk(peer) @t & #t < #t1) &
  not(Ex #t. RevSigningLtk(peer) @t & #t < #t1) &
  not(Ex pid #t. RevEphAny(pid) @t)
  ==>
  Ex pidPeer #t2.
    Running(pidPeer, peer, actor, devPeer, devActor, k1, k2) @t2 &
    #t2 < #t1
```

4.3 Stronger properties that are not met

We next present stronger security properties that are met by some protocols, such as HMQV, Naxos, XHMQV [33, 45, 46] (and to some extent by Signal [21]), but as our analysis shows, not by Olvid.

Session keys computed as output of authenticated key exchange are a function of four secret keys, namely the static keys and the ephemeral keys of each of the two parties. Strong security models like the extended Canetti-Krawczyk (eCK) model introduced in [46] not only consider security of *past* sessions if some of those keys are compromised, but also mandate key secrecy if certain patterns of secret keys are compromised *during the attacked session*. This property is sometimes referred to as security under *maximal exposure (MEX)* attacks [35].

Ephemeral keys reveal. One property mandated by eCK and MEX security is that the session key remains secret, even when both

parties' ephemeral keys are revealed. This property models an attack scenario in which the devices running the protocol rely on a compromised or low-entropy randomness source. The practical relevance of this scenario has been demonstrated by real-world incidents such as Debian's OpenSSL randomness disaster [81].

We modeled this security property by stating that even if both peers' ephemeral secret keys and randomness are compromised during a protocol execution trace, the adversary is still unable to compute the session seed required for computing the session key:

```
lemma key_secrecy_under_compromise_both_ephemeral:
All I R devI devR pidI pidR seed #t1 #t2 #t3 #t4 #tx #ty.
  SessionSeedEstablished(pidI, I, R, devI, devR, seed) @ t1 &
  SessionSeedEstablished(pidR, R, I, devR, devI, seed) @t2 &
  RevEKemSk(pidI, I, R, devI, devR) @ t3 & #t1 < #t3 &
  RevEKemSk(pidR, R, I, devR, devI) @ t4 & #t2 < #t4 &
  RevealRandomness(pidI, I, R, devI, devR) @tx & #t1 < #tx &
  RevealRandomness(pidR, R, I, devR, devI) @ty & #t2 < #ty &
  not(Ex seed #t6. SessSeedRev(pidI, seed) @ t6) &
  not(Ex seed #t6. SessSeedRev(pidR, seed) @ t6) &
  not(Ex #t6. RevKemLtk(I) @ t6) &
  not(Ex #t6. RevKemLtk(R) @t6)
  ==>
  not(Ex #t5. K(seed) @ t5 & #t3 < #t5 & #t4 < #t5)
```

The four not-statements in the premise ensure that only traces relevant to the specific threat model are considered: we exclude adversarial capabilities that are not about ephemeral key reveal.

Key secrecy does not hold under this attack pattern. Intuitively, this attack exists because both the CCCD Protocol and the Full Ratchet Protocol run ephemeral KEM-based key exchanges through an encrypted and authenticated channel, from which they inherit authentication. However, the session seed produced by these protocols is independent of all static keys, so the static keys do not contribute to session-key confidentiality in any way. One can consider this attack as "dual of KCI", where instead of compromising the target's long-term keys, the adversary compromises *only* the target's randomness used for non-long term secrets.

Ephemeral public key reveal. An important aspect of Olvid's design is that ephemeral public keys are not accessible to the adversary. This assumption is justified in the protocol flow, since these keys are transmitted exclusively over the encrypted and authenticated Asymmetric Channel (in the CCCD Protocol) or Oblivious Channel (in both CCCD Protocol and Full Ratchet Protocol). However, requiring public keys to be kept secret is an unusual assumption and may prove fragile when interfacing with implementation security. Implementors of cryptographic libraries typically assume that it is no problem to leak information about public keys through, e.g., timing information. We therefore also model the scenario where, in addition to the static key of one party and the ephemeral key of the other party, all public keys are revealed to the adversary.

```
lemma key_secrecy_under_compromise_of_ephemeral_public_key:
All I R pidI pidR devI devR seed #t1 #t2 #t3.
  SessionSeedEstablished(pidI, I, R, devI, devR, seed) @t1 &
  not(Ex seed #t. SessSeedRev(pidI, seed) @t) &
  not(Ex seed #t. SessSeedRev(pidR, seed) @t) &
  not(Ex sid pa pb deva devb #tx #ty. RevEKemSk(sid, pa, pb, deva,
        devb) @tx & RevKemLtk(pa) @ty) &
  not((Ex #tx #ty. RevEKemSk(pidI, I, R, devI, devR) @tx &
       RevEKemSk(pidR, R, I, devR, devI) @ty)) &
  RevealRandomness(pidI, I, R, devI, devR) @t2 &
  RevEKemPk(pidR, R, I, devR, devI) @t3
  ==>
  not(Ex #t4. K(seed) @t4)
```

Analogously to the previous lemma, the four not-statements in the premise restrict Tamarin to the relevant threat model.

Tamarin finds an attack trace for this scenario. In contrast, protocols like Signal's X3DH have no requirements for ephemeral public keys to be secret, and in fact are not encrypted.

4.4 Analysis summary

Modeling the protocols ensuring alignment with the specifications and implementation took approximately 8–9 person-months, accounting for several iterations of model refinement. We ran our proofs sequentially using the Tamarin prover version 1.10.0 on an Apple M4 Processor machine with 24 GB of RAM. The total execution time was around eleven hours and thirty minutes. We provide the proving times of the main security properties in Appendix D.

Olvid's unique protocol structure meant that we could not use the standard breakdowns in works for other protocols. This posed a new challenge and required us to revisit the way we would usually construct such models. Because Olvid has a more intertwined structure in how its different sub-protocols interact, it is harder to cleanly isolate components in a way that would simplify the analysis while still having meaningful results. For example, the Asymmetric Channel resembles a form of hybrid encryption to a party, rather than a standard channel abstraction. The Oblivious Channel reminds of TLS's record layer, but is used for multiple purposes in a non-standard way.

4.5 Reusability beyond Olvid

At this point, we highlight key takeaways on how our modeling approach and security lemmas can inform the analysis of other protocols, extending beyond the specific case of Olvid. There are three main aspects for re-use for other works: (i) specific sub-protocol models, (ii), security property specifications, and (iii) tailored modeling approaches developed for this work.

Sub-protocols models. At present, we are not aware of any other protocol that shares sub-protocol structures with Olvid. As a result, the exact models we developed in this work cannot be reused in other contexts. Nevertheless, they may still serve as reference points for similar designs should such protocols emerge.

Security property specifications. In contrast, our security properties formalized as lemmas can be directly used by other protocol models. In fact, we developed some new property variants, such as a symbolic variant of the CKA-based PCS property. These property specifications (lemmas) can be re-used by other protocol developers and analysts for comparison or serve as a baseline.

Modeling methodology. Finally, our modeling approach offers direct suggestions and rationales for others that apply to similar protocols, such as our inlining and refactoring.

5 Observations on Olvid's implementation

In order to derive the formal model of Olvid described in Section 4.1, we closely inspected the Java source code of the Android client, version 3.6 [56]. Overall, the codebase is clean and easy to read. In particular, we acknowledge the effort invested in ensuring a close correspondence between the source code and the specification [8]. Specifically, the pseudocode in the specification and the actual

implementation use the same function and variable names where applicable. This close alignment significantly helped with deriving our formal model of Olvid's cryptographic core.

However, while inspecting the code to accurately reflect the protocols in our Tamarin model, we identified several security issues. We report and discuss our findings in this section.

5.1 Non-canonical values

On both Android and iOS, Olvid does not use established libraries for cryptographic primitives, but instead relies on its own implementations of all cryptographic primitives (written in Java and Swift, respectively). In the implementations of elliptic-curve-based primitives, we noticed several instances of assuming that values are in a (fully reduced) canonical representation without actually ensuring first that they are. Most notably, Olvid implements the Elliptic Curve Based Schnorr Digital Signature Algorithm (ECSDSA) [36], but omits range checks. This means that signatures in Olvid are malleable. As Olvid's replay protection relies on storing all signatures received in *Ping* messages at the beginning of the CCCD Protocol in a database, this malleability could result in an attack against replay protection. However, interestingly, we were not able to leverage this to an actual attack. What saves Olvid's replay-protection mechanism is that all messages of the CCCD Protocol are sent through the Asymmetric Channel.

5.2 DoS vulnerability from replay protection

Storing signatures for replay protection, however, yields another attack vector, namely for a DoS attack that aims at filling up the database and thereby eventually exhausting the victim's memory and storage. This process is relatively slow. In an experiment with a Google Pixel 4a device as a victim, we were able to fill up the database at a rate of about 100 KB per minute from a single attacker device. We were able to roughly double this rate when adding a second attacker device. Notably, this DoS attack is persistent; restarting the application or device will not resolve the issue. Regularly emptying the database would clearly help, but also eliminate the replay protection. In Section 6 we discuss a possible mitigation, which consists of replacing signatures by hashes.

5.3 Timing leakage in scalar multiplication

Finally, we found two instances in the open-source implementation of Olvid not following the constant-time programming policy [2], which is widely accepted as state-of-the-art for cryptographic implementations. Specifically, two scalar multiplications, *scalarMultiplication* and *scalarMultiplicationWithX*, implemented for EdwardCurve are not constant-time. Function *scalarMultiplication* is used when encapsulating the long-term private key (Github Link) and function *scalarMultiplicationWithX* is used when deriving the public key from the private key during signature generation (Github Link). Both of them branch on scalar bits and violate the constant-time programming policy by having secret-dependent control flow. Although the two paths of each conditional branch have almost identical bodies, an attacker can still monitor the cache behavior to learn the instructions on which path are executed using the `Flush+Reload` technique [83]. Consequently, it is possible for an attacker to reconstruct the long-term KEM private key and the secret signing key by monitoring the two scalar multiplications.

As a proof-of-concept, we show that it is practical for an attacker to monitor which path of the secret-dependent branch is executed. We present the vulnerable code snippet in Listing 1. The input `n`, which is the scalar, is evaluated by a conditional branch at line 4. The bodies of the branch are almost identical, except that the operands of some operations are different; we highlight differences in red. We first compile the exact same scalar multiplication taken from Olvid's Java source code, to native binary on Intel i7-10710U that runs Ubuntu 20.04.06LTS. Then we monitor two instruction locations, one for each branch body, with the classic `Flush+Reload` technique [83]. That is, the spy process keeps flushing the monitored instructions out of the cache and measuring the cost of reloading them in a periodic manner. If the reload cost is small, it indicates that the victim process has accessed the flushed instruction, which brings the instruction back to the cache. Otherwise, the victim has not accessed the monitored instruction.

Listing 1: Vulnerable Scalar Multiplication

```
1  scalarMultiplication(n, Y) {
2    ...
3  Loop on n :
4    if (n.testBit(i)) {
5      t3 = uR.add(wR).modPow(TWO, p);
6      t4 = uR.subtract(wR).modPow(TWO, p);
7      t5 = t3.subtract(t4).mod(p);
8      u2R = t3.multiply(t4).mod(p);
9      w2R = t5.multiply(t4.add(
10         c.multiply(t5))).mod(p);
11     uQ = uQplusR; wQ = wQplusR;
12     uR = u2R; wR = w2R;
13   } else {
14     t3 = uQ.add(wQ).modPow(TWO, p);
15     t4 = uQ.subtract(wQ).modPow(TWO, p);
16     t5 = t3.subtract(t4).mod(p);
17     u2Q = t3.multiply(t4).mod(p);
18     w2Q = t5.multiply(t4.add(
19         c.multiply(t5))).mod(p);
20     uQ = u2Q; wQ = w2Q;
21     uR = uQplusR; wR = wQplusR;
22   }
23 }
```

We pin the victim process to one core and the spy process to another core. We first execute the spy process, which keeps monitoring the two locations without knowing when the victim starts executing the scalar multiplication. Then, we execute the victim process which processes a random scalar n. Depending on the value of each scalar bit, the victim process executes one of the two branches and the spy process should observe the timing difference of reloading the two locations. If the scalar bit is one, the attacker will observe a small reload cost for instruction location in the then-branch path, and a large reload cost for instruction location under the else-branch path. Fig. 6 presents partial of the results, where the x-axis is the scalar bit index and the y-axis is the reload cost. As shown in the figure, the timing difference between executing different branches is significant: if the victim executes the monitored instruction, the reload cost is always below 50 cycles, otherwise it is always above 100 cycles. According to the figure, the recovered bits are 1011110110.

While our PoC is shown on an Intel processor, previous works demonstrated similar `Flush+Reload` attacks on ARM processors and Android applications that are not constant-time [47, 49, 84]. They typically assume a threat model where a malicious application

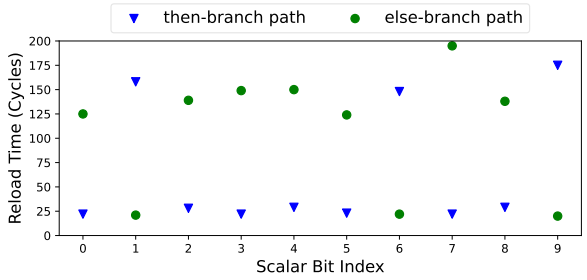


Figure 6: `Flush+Reload` attack on the scalar multiplication. It takes shorter time to reload instructions on the then-branch path if the secret bit is *one*, and shorter time to reload instructions on the else-branch path if the secret bit is *zero*.

(i.e., the spy process) is installed as a usual benign application that requires no additional permission, while the victim application is compiled to native binary, which is common nowadays [47, 84]. This threat model also applies to an adversary targeting Olvid on Android devices. Extending our PoC to an end-to-end exploit would require additional engineering effort and is beyond the scope of this paper. Nevertheless, the Olvid team acknowledged the issue and plan to fix it in future releases, see Appendix A.

6 Further discussion

In this section, we discuss higher-level aspects of Olvid.

6.1 Olvid's anonymity claims

Olvid's advertisement includes questionable statements about anonymity guarantees, for example *"Olvid is the only system that also encrypts metadata, thus guaranteeing the anonymity of interlocutors"*, or *"No third party could ever identify the participants, not even the server. No trace of any metadata."*. In reality, servers see some metadata including recipient public keys (required to deliver messages), message sizes, and timing information of messages. The link between public keys and individuals is not immediately public, but will in many cases be rather trivial to establish, for example if server operators are at the same time also users of Olvid with their own contact list. Following common cryptographic terminology [65], it would certainly be more appropriate to speak about *pseudonymity* rather than anonymity and refrain from making statements about unavailability of metadata to the server. See also the reply by the Olvid designers regarding this criticism, in Appendix A.

Formal security specifications. The Olvid specifications [8] only describe *how* Olvid functions, not *why* protocols are designed the way they are. For formal security analyses such as the one we present in this paper, but also for potential future work, it would be very helpful if the specifications formally stated what assumptions they make about inputs, employed sub-protocols, and primitives, and what security goals they aim to achieve, for *each* sub-protocol.

As one example, the use of the Full Ratchet Protocol clearly hints at post-compromise security (PCS) as a design goal of Olvid, but this is not stated or formalized anywhere. Also, as shown in [25], Olvid does not actually achieve PCS at application level. If PCS *is* a design goal, then changes will be required to extend the property from protocol to application level; if it is not a design goal, an obvious simplification of Olvid's cryptographic core would be to drop the Full Ratchet Protocol altogether.

As another example, the CCCD Protocol at a first glance looks like an authenticated key exchange built from signatures and ephemeral KEM-based key exchange. However, the signatures do not actually contribute to authentication; instead, authentication is inherited from the encrypted and authenticated Asymmetric Channel through which all messages are sent. It is still not entirely clear to us if this is an intentional design decision or a "happy accident".

6.2 Unclear purpose of signatures

One use of signatures in Olvid is to provide replay protection during the CCCD Protocol. In principle, this use case would require signatures to satisfy *strong* existential unforgeability under chosen message attacks (SUF-CMA). However, as explained in Section 5, this condition is not met by Olvid's current implementation. Nevertheless, the replay-protection mechanism is saved by the fact that all messages in the CCCD Protocol are transmitted via the Asymmetric Channel.

A more robust and efficient alternative would be to replace signatures with second-preimage-resistant hashes. Using Tamarin, we analyzed the security properties of the protocol after replacing signatures with such hashes and found that all the security properties that we proved for Olvid with signatures remained intact.

Additional benefits of using hashes instead of signatures would be improved computational performance and smaller storage requirements, as signatures have a size of 80 bytes, whereas hashes would occupy only 32 bytes. The latter would help as a first step towards mitigating the DoS attack vector described in Section 5. However, at the moment, a signed *Ping* message also triggers tearing down already existing channels [7]. When replacing signatures with hashes, this mechanism would need to be updated to use the authentication provided by the Asymmetric Channel.

6.3 Client open-source, server closed-source

At the time of our analysis, Olvid's client code was open source, but the server code was not. Back then, Olvid listed four reasons why they did not yet make the server source code available. First, they argued that because the server is untrusted, *"making the code open source would have no beneficial security impact"*. While we agree that having the client code available is much more important, we disagree that the server code is not relevant for security analysis. For example, it would be very useful to study the code to get a better understanding of what privacy or "anonymity" users can expect if a server is seized by law enforcement, e.g., how it stores and deletes information regarding recipient public keys. The next two reasons were mostly related to the fact that users are discouraged from running their own server until APIs have stabilized. This makes sense, but could also be achieved by clear warnings while still making the server code available for independent security analysis. The last reason was essentially that Olvid would like to have some security by obscurity against DoS attacks.

Overall, we did not find the reasons fully convincing and encouraged Olvid to release the server source code. The Olvid authors

corroborated that they received many such requests; the server source code was released as open source in January 2026.

7 Conclusion

Our work shows that the cryptographic core of Olvid gets many things right, but still falls somewhat short of the goal of matching or even exceeding the state of the art in terms of security.

On the positive side, our formal analysis provides proof that our Olvid models meet core security properties against network attackers. No such results were previously established for Olvid's core key-exchange mechanisms, so our work provides higher assurance in the core design.

On the negative side, our formal analysis also shows that Olvid does not meet stronger properties met by other protocols, or relies on non-standard assumptions. Improving this will require updates to the Olvid protocol stack, which will break backwards compatibility. As minor recommendations, we suggest including the key output of the encapsulations to static keys in the session-key derivation in the CCCD Protocol, and including the previous session keys in the derivation of fresh keys in the Full Ratchet Protocol. We are in a constructive dialogue with the Olvid developers that we hope will eventually result in a concrete proposal; see also Appendix A.

Some of the issues we found with the source code are fairly easy to address and have to a large extent already been addressed as a reaction to our disclosure. This includes eliminating the timing leakage in elliptic-curve scalar multiplication (at least on source level; robust solutions to preserve this property through compilation for mainstream languages are still lacking [37, 40, 72]). Also, more careful canonicalization through modular reductions before range checks or other comparisons is easy to include and have already been included at least in one spot.

Unlike Signal and iMessage, Olvid does not yet provide post-quantum security, or resilience to the harvest-now-decrypt-later threat model. Given that the design is KEM-based, this should be relatively straightforward to update, but has not yet been done.

Acknowledgments

This work was supported by the Australian Research Council Discovery Project DP210102670; the Dutch Ministry of Education, Culture, and Science through Gravitation project "Challenges in Cyber Security – 024.006.037"; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and through project number 560392681;

References

[1] Michel Abdalla. 2020. Security Analysis of Olvid's SAS-based Trust Establishment Protocol. Cryptology ePrint Archive, Paper 2020/808.
[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security 2016*.
[3] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT 2019*.
[4] Joël Alwen, Daniel Jost, and Marta Mularczyk. 2022. On the Insider Security of MLS. In *CRYPTO 2022*.
[5] Thomas Baignères, Cécile Delerablée, Matthieu Finiasz, Louis Goubin, Tancrède Lepoint, and Matthieu Rivain. 2015. Trap Me If You Can – Million Dollar Curve. Cryptology ePrint Archive, Report 2015/1249.
[6] Thomas Baignères and Matthieu Finiasz. 2019. Security models of mobile messaging apps and applications of cryptography in communications. https://olvid.io/assets/documents/2019-06-12_Olvid_GDR-Securite-Informatique.pdf. Accessed: 2026-06-04.
[7] Thomas Baignères and Matthieu Finiasz. 2025. Personal communication.
[8] Thomas Baignères and Matthieu Finiasz. 2024. Specifications of Olvid – Application and Server. https://olvid.io/assets/documents/2024-10-07_Olvid-specifications.pdf. Published: 2024-10-07. Accessed: 2026-06-04.
[9] Elaine Barker and John Kelsey. 2025. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. NIST SP 800-90A.
[10] Jean-Noël Barrot. 2023. Tweet About Olvid. https://x.com/jnbarrot/status/1729964589742219712. Published: 2023-11-29. Accessed: 2026-06-04.
[11] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. 2018. A Formal Analysis of 5G Authentication. In *ACM CCS 2018*.
[12] David Basin, Ralf Sasse, and Jorge Toro-Pozo. 2021. The EMV Standard: Break, Fix, Verify. In *2021 IEEE Symposium on Security and Privacy*.
[13] Mihir Bellare and Phillip Rogaway. 1993. Entity Authentication and Key Distribution. In *CRYPTO 1993*.
[14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2014. "Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way. In *CHES 2014*.
[15] Daniel J. Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *PKC 2006*.
[16] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2011. High-speed high-security signatures. In *CHES 2011*.
[17] 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*.
[18] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies 2017*.
[19] Ran Canetti and Hugo Krawczyk. 2001. Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels. In *EUROCRYPT 2001*.
[20] Sofia Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. 2022. A Tale of Two Models: Formal Verification of KEMTLS via Tamarin. In *ESORICS 2022*.
[21] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2017. A Formal Security Analysis of the Signal Messaging Protocol. In *2017 IEEE European Symposium on Security and Privacy*.
[22] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. 2016. On Post-Compromise Security. In *CSF 2016*.
[23] Ronald Cramer and Victor Shoup. 2003. Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack. *SIAM J. Comput.* (2003).
[24] Cas Cremers, Alexander Dax, Charlie Jacomme, and Mang Zhao. 2023. Automated Analysis of Protocols that use Authenticated Encryption: How Subtle AEAD Differences can impact Protocol Security. In *USENIX Security 2023*.
[25] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. 2020. Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice. In *ACM CCS 2020*.
[26] 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*.
[27] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. 2016. Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. In *2016 IEEE Symposium on Security and Privacy*.
[28] 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 2023*.
[29] Euclidia. 2022. Olvid: the truly secure european messenger. https://www.euclidia.eu/publications/FDL-Cloud.Success.Case.Olvid. Accessed: 2026-06-04.
[30] Euronews/AFP. 2023. France's government ministers to ditch using WhatsApp for French app Olvid over security fears. https://www.euronews.com/next/2023/11/30/frances-government-ministers-to-ditch-using-whatsapp-for-french-app-olvid-over-security-fe. Published: 2023-11-30. Accessed: 2026-06-04.
[31] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. 2016. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *ACM CCS 2016*.
[32] Rune Fiedler and Felix Günther. 2025. Security Analysis of Signal's PQXDH Handshake. In *PKC 2025*.
[33] Rune Fiedler, Felix Günther, Jiaxin Pan, and Runzhi Zeng. 2025. XHMQV: Better Efficiency and Stronger Security for Signal's Initial Handshake based on HMQV. In *CRYPTO 2025*.
[34] Rune Fiedler and Christian Janson. 2024. A Deniability Analysis of Signal's Initial Handshake PQXDH. *PoPETs 2024* (2024).
[35] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. 2012. Strongly secure authenticated key exchange from factoring, codes, and lattices. In *PKC 2012*.
[36] Bundesamt fur Sicherheit in der Informationstechnik. 2018. Elliptic Curve Cryptography. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publ

ications/TechGuidelines/TR03111/BSI-TR-03111_V-2-1_pdf.pdf. Accessed: 2026-06-04.
[37] Antoine Geimer and Clémentine Maurice. 2025. Fun with flags: How Compilers Break and Fix Constant-Time Code. arXiv, Report 2507.06112.
[38] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-By Key-Extraction Cache Attacks from Portable Code. In *Applied Cryptography and Network Security*.
[39] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *ACM CCS 2017*.
[40] Lukas Gerlach, Robert Pietsch, and Michael Schwarz. 2025. Do Compilers Break Constant-time Guarantees?. In *International Conference on Financial Cryptography and Data Security*.
[41] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security 2015*.
[42] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses – RAID 2014*.
[43] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. 2019. Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures. In *ACM CCS 2019*.
[44] Marc Joye and Sung-Ming Yen. 2002. The Montgomery Powering Ladder. In *CHES 2002*.
[45] Hugo Krawczyk. 2005. HMQV: A High-Performance Secure Diffie-Hellman Protocol. Cryptology ePrint Archive, Paper 2005/176.
[46] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. 2007. Stronger Security of Authenticated Key Exchange. In *Provable Security*.
[47] Yan Lin, Joshua Wong, Xiang Li, Haoyu Ma, and Debin Gao. 2024. Peep With A Mirror: Breaking The Integrity of Android App Sandboxing via Unprivileged Cache Side Channel. In *USENIX Security 2024*.
[48] Felix Linker, Ralf Sasse, and David Basin. 2025. A Formal Analysis of Apple's iMessage PQ3 Protocol. In *USENIX Security 2025*.
[49] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security 2016*.
[50] Gavin Lowe. 1997. A Hierarchy of Authentication Specifications *(CSFW 1997)*.
[51] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification*.
[52] Le Monde. 2023. Olvid, the French government's 'secure' messaging app, already under fire. https://www.lemonde.fr/en/economy/article/2023/12/15/olvid-the-french-government-s-secure-messaging-app-already-under-fire_6346249_19.html. Published: 2023-12-15. Accessed: 2026-06-04.
[53] Peter L. Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.* (1987).
[54] Katsuyuki Okeya, Hiroyuki Kurumatani, and Kouichi Sakurai. 2000. Elliptic Curves with the Montgomery-Form and Their Cryptographic Applications. In *PKC 2000*.
[55] Olvid. 2025. Olvid – Technology: One giant leap for messaging. https://olvid.io/technology/en/. Accessed: 2026-06-04.
[56] Olvid. 2025. Olvid Android Client, Version 3.6. https://github.com/olvid-io/olvid-android/releases/tag/v3.6_263. Published: 2025-02-10. Accessed: 2026-06-04.
[57] Olvid. 2025. Olvid Keycloak Configuration Guide. https://www.olvid.io/keycloak/. Accessed: 2026-06-04.
[58] Olvid. 2025. President Macron renews his trust in Olvid. https://olvid.io/news/en/. Published: 2025-11-17. Accessed: 2026-06-04.
[59] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA 2006*.
[60] Sylvain Pasini. 2009. *Secure communication using authenticated channels*. Ph. D. Dissertation. EPFL, Switzerland. https://infoscience.epfl.ch/server/api/core/bitstreams/91734ac6-1e59-4ff4-b2d3-15b32883282e/content. Accessed: 2026-06-04.
[61] Sylvain Pasini and Serge Vaudenay. 2006. SAS-based authenticated key agreement. In *PKC 2006*.
[62] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. 2023. Three Lessons From Threema: Analysis of a Secure Messenger. In *USENIX Security 2023*.
[63] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan '05*.
[64] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *ACM CCS 2016*.
[65] Andreas Pfitzmann and Marit Köhntopp. 2001. Anonymity, Unobservability, and Pseudonymity – A Proposal for Terminology. In *Designing Privacy Enhancing Technologies 2001*.
[66] Politico. 2023. France bans ministers from WhatsApp, Signal; demands French alternatives. https://www.politico.eu/article/france-requires-ministers-to-swap-whatsapp-signal-for-french-alternatives/. Published: 2023-11-30. Accessed: 2026-06-04.
[67] Reuters. 2023. Stop using WhatsApp, get Paris-made alternative, French PM tells ministers. https://www.reuters.com/world/europe/stop-using-whatsapp-get-paris-made-alternative-french-pm-tells-ministers-2023-11-29/. Published: 2023-11-29. Accessed: 2026-06-04.
[68] Phillip Rogaway. 2002. Authenticated-encryption with associated-data. In *ACM CCS 2002*.
[69] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. 2018. Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure. In *ACM CCS 2018*.
[70] Paul Rösler, Christian Mainka, and Jörg Schwenk. 2018. More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema. In *2018 IEEE European Symposium on Security and Privacy*.
[71] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. 2012. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *CSF 2012*.
[72] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. 2025. Breaking Bad: How Compilers Break Constant-Time Implementations. In *ACM CCS 2025*.
[73] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. 2021. Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite. In *USENIX Security 2021*.
[74] Victor Shoup. 2001. A Proposal for an ISO Standard for Public Key Encryption. Cryptology ePrint Archive, Paper 2001/112.
[75] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security 2019*.
[76] Douglas Stebila. 2024. Security analysis of the iMessage PQ3 protocol. Cryptology ePrint Archive, Paper 2024/357.
[77] Synacktiv. 2020. Cible de Sécurité CSPN – Olvid Android. https://messervices.cyber.gouv.fr/visas/ANSSI-CSPN-2021-12-cible.pdf, version 1.1. Accessed: 2026-08-03.
[78] Synacktiv. 2020. Rapport technique d'évaluation – Olvid iOS. https://www.olvid.io/assets/documents/Synacktiv-Olvid-CSPN-Olvid-0.8.2-RTE-public.pdf, version 1.2. Accessed: 2026-08-03.
[79] Synacktiv. 2021. Rapport technique d'évaluation – Olvid Android. https://olvid.io/assets/documents/Synacktiv-Olvid-CSPN_Olvid-0.9.2-RTE-v1.2.pdf, version 1.2. Accessed: 2026-06-04.
[80] Kien Tuong Truong, Noemi Terzo, and Kenneth G. Paterson. 2026. Message Injection Attacks Against Signal. In *USENIX Security 2026*.
[81] Florian Weimer. 2008. New openssl packages fix predictable random number generator. Debian Security Advisory 1571-1. https://lists.debian.org/debian-security-announce/2008/msg00152.html, bug discovered by Luciano Bello. Published: 2008-05-13. Accessed: 2026-06-04.
[82] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. 2020. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security 2020*.
[83] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security 2014*.
[84] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *ACM CCS 2016*.

A Ethical Considerations

Responsible disclosure is critical for our research, since our object of study is used by high-risk individuals. Our main results are positive symbolic proofs of the exact type of security that Olvid provides, but we also note that (i) it does not provide some stronger security guarantees provided by others, (ii) the use of non-constant time code can lead to a potential timing attack, and (iii) we find some potential points to improve the security of the code.

After we established our main results, we reached out to the main developers of Olvid, Baignères and Finiasz, on April 2, 2025 and shared a high-level view of our findings. We followed up with more details in a video call with them on April 9. The reaction by the Olvid developers was very positive, not only regarding our formal proofs, but also regarding the aspects of Olvid that we believe can be improved. We also received a written reply on April 10. Importantly, they did not object to us submitting our manuscript to scientific conferences.

In particular, regarding our observations on non-optimal security guarantees and implementation, they wrote that they would be *"glad to exchange with you on the best solutions to fix them and work on plans to deploy them in our applications. Backward compatibility constraints may slow down the roll out, but we definitely want to improve our protocols anywhere this can reasonably be done."*

With respect to our observations on the (lack of) anonymity, they responded that Olvid's anonymity claims *"should be understood in the "non technical" sense of anonymity, that is, our server does not know the name or other personal information of the users. Olvid definitely does not provide "cryptographic anonymity" as each user is uniquely identified by their public key: at best, Olvid achieves pseudonymity."*, thereby confirming our observations. Moreover, in this context, they point out that Olvid has no user directory and can operate without name, phone number, or e-mail address of users, unlike others.

In November 2025 we exchanged more details. The Olvid authors explicitly noted that we can publish our article whenever it is done since, as they put it, *"you gave us plenty of time to implement fixes"*. They wrote that in the first instance they aim to to: (i) release the open source code of the server, (ii) make the EC computations constant-time, and (iii) fix the non-canonical values.

In May 2026, we provided additional suggestions to the Olvid authors on how to address the timing leaks and commented on the limitations that a Java-level solution has due to compiler optimizations. We also confirmed that we consider the proposed solution to removing non-canonical values in signature verification appropriate but stressed the need for a more systematic code review with regards to the use of non-canonical values beyond this specific occurrence.

B Open Science

The artifact associated with this paper is available as a Zenodo record at https://doi.org/10.5281/zenodo.20647106. See the artifact appendix for more information about the artifact.

C Cryptographic primitives

The cryptographic core of Olvid makes use of several cryptographic primitives, which we briefly introduce here. Since our symbolic analysis treats primitives as ideal, we omit detailed discussion of computational security notions and only briefly mention the security properties that we need for our discussion of Olvid.

Key encapsulation mechanisms. A key encapsulation mechanism (KEM) [23] is a triple of algorithms $(KGen, Enc, Dec)$ that can be used to establish a shared key between two parties. The key generation algorithm $KGen$ probabilistically generates key pairs (pk, sk), where pk is the public key and sk is the secret key. The encapsulation algorithm Enc receives as input a public key pk and probabilistically computes a shared key k and a ciphertext ct. The deterministic decapsulation algorithm Dec takes as input a secret key sk and a ciphertext ct, and computes a shared key k or returns a failure symbol $\perp$. A KEM is correct if $P(k = k' \mid (pk, sk) \leftarrow KGen(), (k, ct) \leftarrow Enc(pk), k' \leftarrow Dec(sk, ct)) = 1$. The KEM employed in Olvid achieves indistinguishability under chosen-ciphertext attacks (IND-CCA); for a formal definition see, e.g., [23, Sec. 7.1].

Authenticated Encryption. An authenticated-encryption (AE) scheme is a triple of algorithms $(KGen, Enc, Dec)$. The $KGen$ routine produces a value from the keyspace of the AE scheme. For the purpose of this paper, we assume this to be a fresh random bit string. However, recent versions of Olvid employ a specific routine involving a PRNG that also takes the padded plaintext as input. For details, see [8, Sec. 23.1]. Note that also output of PRNG and KDF are used as key for the AE scheme in Olvid (see next paragraph). The algorithm Enc, on input a key k and a plaintext m, computes an authenticated ciphertext c. The algorithm Dec, on input an authenticated ciphertext c and a key k, returns the plaintext or a failure symbol $\perp$. An AE algorithm is correct if $P(m = m' \mid c \leftarrow Enc(k, m), k' \leftarrow Dec(k, c)) = 1$. Informally, we require that it is impossible for an attacker who has no knowledge about k to learn information about m given c (confidentiality), or to generate a c with $Dec(k, c) \neq \perp$ (authentication). For formal definitions, see, e.g., [68].

PRNGs and KDFs. A pseudorandom-number generator (PRNG) is a pair of algorithms $(Init, Next)$. The $Init$ algorithm, on input a seed s, produces a PRNG state $prng$. The $Next$ algorithm, on input a PRNG state $prng$, outputs a bitstring k and updates the state $prng$ in place. For the purpose of this paper, a key-derivation function (KDF) is simply a non-incremental version of a PRNG; i.e., $k \leftarrow KDF(s)$ is equivalent to $prng \leftarrow PRNG.Init(s), k \leftarrow PRNG.Next(prng)$.

Digital signatures. A digital signature scheme is a triple of algorithms $(Gen, Sign, Vrfy)$. The probabilistic key-generation algorithm Gen outputs a key pair (pk, sk). The signature generation algorithm $Sign$ takes as input the signing key sk and a message m and outputs a signature α. The verification algorithm $Vrfy$ takes as input the verification key pk and the signed message (m, α) and returns a boolean value. A signature scheme is correct if $P(Vrfy(pk, (m, \alpha)) = 1 | (pk, sk) \leftarrow Gen(), \alpha \leftarrow Sign(sk, m)) = 1$.

Two security notions for digital-signature schemes are relevant in this paper: existential unforgeability under chosen-message attacks (EUF-CMA) and strong unforgeability under chosen-message attacks (SUF-CMA). In both notions, an attacker is allowed to adaptively query for signatures on chosen messages. In order to break EUF-CMA, the attacker then needs to produce a signature α on a fresh (i.e., not previously queried) message m such that

$Vrfy(pk, (m, \alpha)) = 1$. For the stronger SUF-CMA notion, the attacker needs to produce a message-signature pair (m, α) that was not an input-output pair of one of the earlier queries. In other words, producing a *different signature* on a message the attacker queried before is sufficient to break SUF-CMA. Signatures produced by schemes that are EUF-CMA secure but not SUF-CMA secure are often referred to as *malleable*.

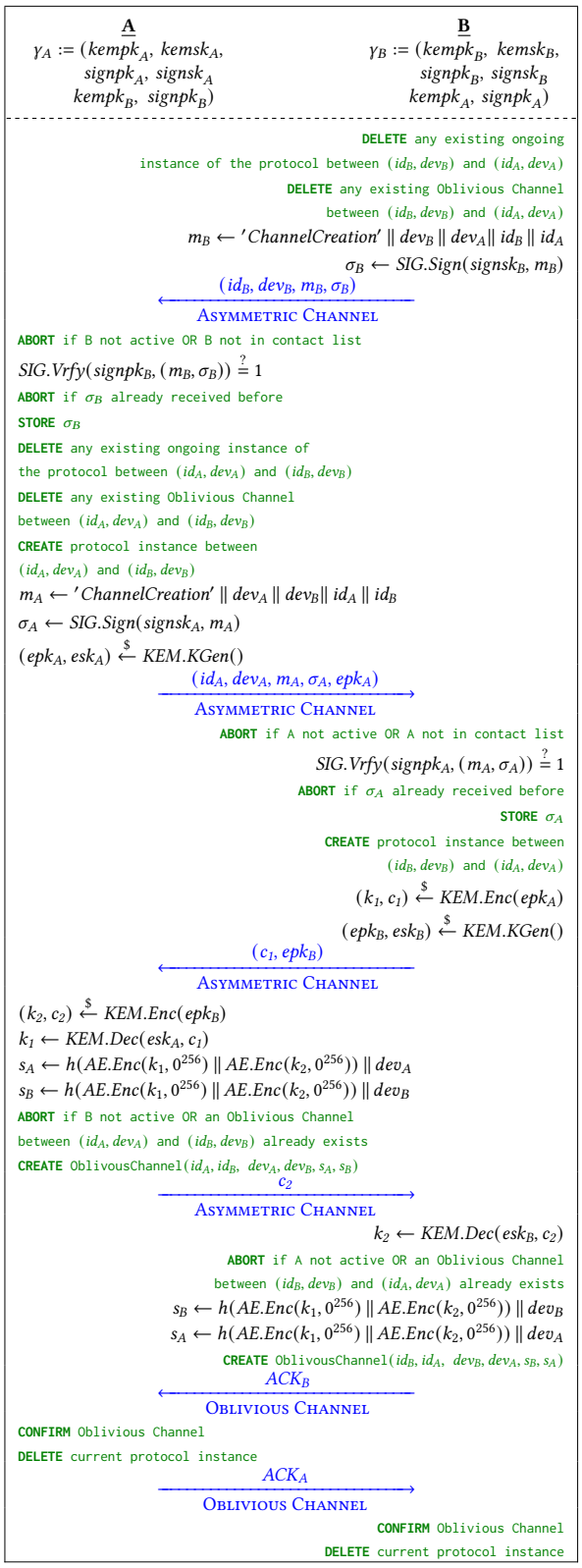
D Summary of formal analysis results

All lemmas in Table 1 are analyzed automatically in Tamarin. The runtimes indicate the time Tamarin required to analyze each property. We write ✓ to indicate that Tamarin verifies the property, i.e., we obtain a proof in our symbolic model; ✗ indicates that Tamarin finds a counterexample, i.e., an attack on the property.

Table 1: Tamarin analysis results and runtimes

Protocol	Property	Result	Time (s)	#Steps
CCCDP	Mutual Authentication	✓	288	1350
CCCDP	KCI Resistance	✓	30576	102970
CCCDP	Replay Protection	✓	5	16
CCCDP	PFS	✓	68	325
CCCDP	Stronger Key Secrecy	✓	68	325
CCCDP	Key Secrecy under Static-Ephemeral Reveal	✓	1980	6201
CCCDP	MEX Resistance	✗	312	29
CCCDP	Weak eCK PFS	✗	112	32
CCCDP	eCK PFS	✗	116	32
CCCDP	Key Secrecy under ephemeral keys reveal	✗	38	29
CCCDP	Key Secrecy under ephemeral public key reveal	✗	2065	29
FRP	PCS	✓	4069	42379
FRP	Mutual Authentication	✓	1483	22320
FRP	Weak eCK PFS	✗	69	21
FRP	eCK PFS	✗	26	14

E Extended CCCD Protocol



F Artifact Appendix

This appendix assists readers who wish to reproduce our results by running our Tamarin proofs for the CCCDP and FRP, or the proof-of-concept (PoC) of the timing side-channel attack.

F.1 Description & Requirements

The artifact is organized as follows:

- `cccdprotocol/` and `frprotocol/` — Tamarin models of the CCCDP and of the FRP. The folders contain the models of participants, adversary, channels, restrictions and lemmas.
- `proof_results/` — our reference proof results, containing the files `cccdp.out` and `frp.out`.
- `PoC_Flush_Reload/` — PoC of the timing side-channel attack against Olvid's scalar multiplication, with its own README file describing how to replicate the results.

The top-level `README.md` is the main entry point and documents how to install Tamarin and run the proofs. A `Dockerfile` and a `Makefile` automate the construction of the environment and the execution of the proofs. To understand more about the structure of our models, see Section 4 and the comment lines in our .spthy and .splib Tamarin specification files.

F.1.1 Security, privacy, and ethical concerns. The Tamarin part of the artifact performs no destructive step and disables no security mechanism on the evaluator's machine. The PoC is a microarchitectural Flush+Reload attack that runs entirely locally, against a victim process that the evaluator launches on their own machine: it targets no third party and exfiltrates no data. Building it requires installing GraalVM and using `gdb` to inspect a locally compiled binary; no other security mechanism is disabled.

F.1.2 How to access. The artifact is permanently and publicly available on Zenodo at https://doi.org/10.5281/zenodo.20647106. After downloading the tarball, extract it (`tar -xvjf olvidgbu.tar.bz2`); all commands in the remainder of this appendix assume that the working directory is the root of the extracted artifact (`olvidgbu/`).

F.1.3 Hardware dependencies. The Tamarin proofs do not require any special hardware feature. Our reference measurements were obtained on an Apple M4 machine with 24 GB of RAM, all allocated to Docker since the heavier lemmas benefit from additional memory (see the troubleshooting note in Section F.2). The PoC requires an x86-64 CPU exposing the `clflush` instruction; since cache behavior and clock frequency differ across processors, the timing parameters may need CPU-specific tuning (see `PoC_Flush_Reload/README.md`). We ran the PoC on an Intel i7-10710U machine with 32 GB of RAM, running Ubuntu 20.04.

F.1.4 Software dependencies. The Tamarin part relies on version 1.10.0 of the Tamarin prover, which in turn depends on Maude (up to 3.5), Graphviz and Haskell Stack. The provided Docker image (built on `debian:stable-slim`) installs all dependencies automatically; alternatively, Tamarin can be built from source (see Section F.2). The timing side-channel PoC requires GraalVM (to compile the Java code into a native binary), a C compiler (`gcc`), and the `gdb` debugger together with `objdump` (to locate the offset of the vulnerable branch in the compiled binary).

F.1.5 Benchmarks. None. The Tamarin analysis does not depend on any external dataset, and the PoC generates its own inputs at runtime.

F.2 Set Up

This section describes how to prepare an environment for the Tamarin proofs; for the PoC, see experiment (E4) of Section F.3 and the corresponding `README.md`. Detailed instructions for both supported options — using the provided Dockerfile (strongly recommended) or building Tamarin from source — are given in the top-level `README.md`.

F.2.1 Installation. The Dockerfile sets up a container with Tamarin 1.10.0 and the artifact contents; for installing Docker, see https://docs.docker.com/get-docker/. To reproduce all our results, we recommend allocating 24 GB of RAM to Docker. On Linux, to access the proof results outside the container, first create the `out` directory and assign its ownership to the container user (`mkdir out; sudo chown 1001:1001 out`); this is not required on macOS. Then build and run the container:

```
docker build -t olvidgbu .
docker run -it -v $(pwd)/out:/home/ccs2026/out olvidgbu
```

F.2.2 Basic test. After installation (with either Docker or from source), verify that Tamarin works with `tamarin-prover --version`; the expected output starts with `tamarin-prover 1.10.0`. As a first functional check of the proving pipeline, a single lemma can be proved with:

```
make prove-lemma LEMMA=<lemma> PROTOCOL=<
↪ protocol_file.spthy>
```

For example, proving `model_mex_resistance` in `cccdprotocol/cccdp.spthy` should print `model_mex_resistance (all-traces): falsified - found trace (29 steps)`, i.e., Tamarin found a counterexample showing that CCCDP does not provide resistance against maximal exposure (MEX) attacks (see Section 4.3). The result is written under `out/`; the file-naming convention and a troubleshooting section for proofs that run out of Docker memory are documented in the `README.md`.

F.3 Evaluation Workflow

F.3.1 Major claims.

(C1): The CCCDP satisfies the security properties analyzed in Section 4.2. This is proven by experiment (E1), whose lemmas `model_mutual_authentication`, `model_KCI_resistance`, `model_replay_protection`, `model_PFS`, `model_stronger_key_secrecy` and `model_key_secrecy_under_static_ephemeral_reveal` all terminate with the result `verified`.

(C2): The CCCDP does *not* meet a number of stronger security notions discussed in Section 4.3. This is proven by experiment (E1), whose corresponding lemmas (e.g., `model_mex_resistance`, `model_eCK_PF_seed_secrecy`) terminate with the result `falsified`, i.e., Tamarin exhibits a counterexample.

(C3): The FRP provides post-compromise security (PCS) and injective mutual entity authentication (see Section 4.2), but does not meet the stronger eCK PFS properties (see Section 4.3). This is proven by experiment (E2).
(C4): All the security properties that we prove for the CCCDP with signatures also hold when signatures are replaced by hashes, as discussed in Section 6.2. This is proven by experiment (E3).
(C5): Olvid's scalar multiplication contains an input-dependent branch that can be exploited through a Flush+Reload timing side-channel. This is demonstrated by experiment (E4).

F.3.2 Experiments. All Tamarin results can be cross-checked against the table in the artifact's `README.md`, which reports, for every lemma, the corresponding paper section, the expected result, the proving time and the number of proof steps (measured inside the Docker container on an Apple M4 machine with 24 GB of RAM; for the times obtained running the proofs directly on the host, see the table in Appendix D).

(E1): *CCCDP proofs* [a few person-minutes + several compute-hours]: reproduces claims (C1) and (C2). By default, the lemmas of the CCCDP are proved with signatures.
Preparation: see Section F.2.
Execution: run `make prove-cccdp`; the output is stored in `out/cccdp_sign.out`.
Results: the `sanity_*` lemmas should be `verified`; the `model_*` lemmas should match the `verified`/`falsified` value listed in the table. The reference results are also available in `proof_results/cccdp.out`.

(E2): *FRP proofs* [a few person-minutes + a few compute-hours]: reproduces claim (C3).
Preparation: same as (E1).
Execution: run `make prove-frp`; the output is stored in `out/frp.out`.
Results: compare the results against the FRP rows of the table. The reference results are also available in `proof_results/frp.out`.

(E3): *CCCDP with hashes* [a few person-minutes + several compute-hours]: reproduces claim (C4). We replace the signatures of the CCCDP with hashes and verify that the same security properties still hold.
Preparation: same as (E1).
Execution: run `make prove-cccdp SIGNORHASH=HASH`; the output is stored in `out/cccdp_hash.out`.
Results: the verified and falsified properties match the ones obtained with signatures in (E1).

(E4): *Timing side-channel PoC* [about 1 person-hour + a few compute-minutes]: reproduces claim (C5). The PoC targets the scalar multiplication function in `Hello.java` using the attacker code in `attacker.c`.
Preparation: run all commands from the `PoC_Flush_Reload/` directory. Install GraalVM (https://www.graalvm.org/latest/getting-started/linux/) and set the corresponding environment variables.
Execution (automated, recommended): three helper scripts automate the whole pipeline: `setup_graalvm.sh` installs the packages and GraalVM, `find_offset.sh` compiles `Hello.java` and patches the vulnerable branch offset into `attacker.c`, and `auto_run.bash` runs the attack. The manual steps are given in the README.
Results: the spy process recovers the input-dependent behavior of the branch through cache-timing measurements, demonstrating the side-channel (the timing parameters in `attacker.c` may need CPU-specific tuning, see the README). A reference run, obtained on our Intel i7-10710U machine with the scripts of this artifact, is provided in `sample_output/`.