A Comprehensive Reassessment of Matrix Security Protocols through Automated Formal Analysis

Hao Li[1], Xiaofeng Liu[1], Yu Wang[1], Chengyu Hu[1], Shanqing Guo[12*]
[1]Shandong University, China. [2]Quancheng Laboratory, China.
202020883@mail.sdu.edu.cn, guoshanqing@sdu.edu.cn

***Abstract*—Matrix, at the forefront of decentralized instant messaging communication, offers robust end-to-end cryptographic security protocols encompassing key-level key distribution, request and backups, device-level SAS-based device verification, and session-level Olm cryptographic ratchet. The Matrix security protocols ensure various security properties, such as confidentiality, authenticity, forward secrecy, and backward secrecy.**

It is crucial to study the feasibility of automating the verification process for Matrix security protocols to ensure the fulfillment of their security properties, uncovering any potential security vulnerabilities. In this paper, we introduce a modularized and automated formal verification approach aimed at identifying issues within Matrix security protocols. To formalize the protocols, we create diverse threat models tailored to address various attack scenarios, namely, "Server Security Compromised", "External Security Compromised", and "Compulsory Access". Subsequently, we conduct a comprehensive examination and abstraction of the protocol flow for distinct Matrix security protocols, and we modularize and implement the modeling of these protocols using ProVerif. Our formal models encompass a range of threat models and security properties, defined through event queries, to enhance the efficiency of protocol validation. Following the formal verification process, we identified multiple security breaches within the Matrix protocols, spanning man-in-the-middle, impersonation, and cloning attacks. Moreover, our study involved a comprehensive examination of the attack traces, aiming to offer a more detailed formal analysis of the Matrix security protocol.

***Index Terms*—Matrix protocol, security, formal verification**

I. Introduction

Traditional centralized communication platforms (e.g., Facebook, Twitter, WhatsApp, etc.) typically control information collaboration and data sovereignty centrally on a centralized server or platform [1]. This centralized architecture has a number of potential problems and challenges such as, data sovereignty, single point of failure, monitoring and privacy issues, and limited openness. With the development of Web3, decentralized communication protocols are becoming more and more popular [2], and most decentralized instant messaging protocols tend to rely on distributed systems, which can help combat government censorship of data and the monopoly of tech giants. As an energetic promoter of the decentralized communication protocol, Matrix [3] gets the favor of massive customers and claims to have 80M+ accounts [4], including individuals, enterprises, government departments, universities, etc. For instance, the French government embraced Matrix to build Tchap Agent, a secure messenger and collaboration tool, whose user base reached 160,000 amid the Coronavirus pandemic [5].

Matrix is an open standard decentralised real-time communication system based on a federation specification protocol [6], and the entire network has no central data point. Anyone can participate in the federation via a self-hosted `Homeserver`, and Matrix `User` communicates with each other in chat `Room`. Matrix protocol enhances the security of communications by implementing end-to-end encryption [7], which ensures that communications are encrypted in transit and can only be decrypted by participating parties. To protect end-to-end encrypted messages, Matrix has developed and implemented security protocols [6], [8], [9], which includes key distribution, request and backup protocols for key-level security, device authentication protocol for device-level security, and cryptographic ratchet protocol for hardened message session-level security.

Motivation. The motivations of this work to investigate the Matrix security protocols are as following:

- Presently, to the best of our knowledge, there exists a deficiency in the automated formal verification and security analysis of Matrix security protocols, encompassing aspects such as key-level, device-level, and session-level protocols.
- The Matrix instant messaging network functions in a decentralized fashion, meaning there are no centrally controlled servers (e.g., Alice and Bob are belonging to different homeservers). The decentralized structure, which includes various roles and participants, poses increased challenges and complexity when formally modeling the Matrix security protocol. Therefore, it is crucial to establish a threat model capable of addressing the intricacies inherent in a decentralized scenario.
- Security protocols within the Matrix, encompassing diverse sub-protocols, specifically, key distribution, device authentication, cryptographic ratchets, key requests, and key backups protocols, which require the development and design of automated formalisms with modular capabilities to generate attack traces during the formalization process.

What we have done and findings. We have formulated threat models and security properties specifically tailored for various Matrix security protocols, as outlined in Section III-A and III-B. On the one hand, our proposed threat models cover scenarios such as "*Server Security Compromised*", "*External Security Compromised*", and "*Compulsory Access*". On the

*Corresponding author.

other hand, the security properties encompass *Confidentiality*, *Authenticity*, *Perfect Forward Secrecy* (PFS), and *Post-Compromised Security* (PCS). Building upon the aforementioned groundwork, we constructed a modular formal model to comprehensively verify the security of the Matrix security protocol, that to ensure the fulfillment of the designated security properties under our defined threat models.

Specifically, in Section IV, we kick off by modeling the key distribution protocol (Sec. IV-A), covering generate, upload, claim, and query. We also analyze and model the key request protocol (Sec. IV-D) and key backup protocol (Sec. IV-E) at the key-level, this ensures a solid grasp of these critical protocols. Following that, we conduct a detailed analysis and modeling of the device-level SAS (Short Authentication String) device authentication protocol [10] in Sec. IV-B. This step involves scrutinizing the authentication process, acting like a security checkpoint for device legitimacy. Finally, we wrap up with a thorough analysis and modeling of the end-to-end encrypted (E2EE) message session-level Olm cryptographic ratchet protocol (Sec. IV-C). In essence, our approach helps verify the security of the Matrix protocol by systematically peeling back the layers of these protocols.

After conducting the formal verification method that we designed and implemented, we uncovered security property violations within different Matrix security protocols under diverse threat models, revealing distinct attack traces. Specifically, the key distribution and key backups at the key-level protocol exhibit vulnerabilities to man-in-the-middle (MitM) attacks, compromising the authenticity security property. Additionally, the key requests protocol exposes security risks, including man-in-the-middle and clone attacks, leading to violations of authenticity, confidentiality, PFS and PCS. Furthermore, the device-level SAS device verification protocol reveals susceptibility to an impersonation attack, undermining authenticity. Finally, our proposed formal authentication model also can expose an impersonation attack within the communication session-level Olm encryption ratchet protocol, resulting in violations of authenticity, PFS and PCS security properties. For a detailed exploration of these findings, refer to Sec. V.

Contributions. The main contributions are summarized as follows:

- To the best of our knowledge, we are pioneering a comprehensive formal model for the Matrix security protocols, encompasses five security protocols of the Matrix, namely key distribution, key requests, key backups, SAS device verification, and Olm-based cryptographic ratchet protocol.
- We've designed and implemented a modular modeling approach with the primary objective of compartmentalizing each Matrix security protocol, which is versatile, and accommodating various threat models and security properties.
- Leveraging our proposed formal verification model, we have effectively pinpointed security vulnerabilities in several Matrix security protocols. These vulnerabilities encompass man-in-the-middle (MitM) attacks, impersonation attacks, and clone attacks. It's important to highlight that the MitM attack is a previously known vulnerability, whereas the other issues are newly discovered security concerns. We responsibly disclosed the protocol vulnerabilities and issues we discovered to the Matrix team. They confirmed that the Matrix security encryption system indeed has a vulnerability causing encrypted message leakage, attributing it to the initial trust establishment in the protocol.

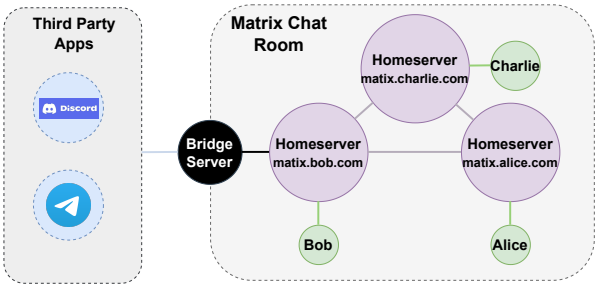

Fig. 1. Overview of Matrix federation framework.

Roadmap. In Sec. II, we provide the background of our work, followed by an elucidation of the threat models and security properties proposed for the Matrix security protocol (Sec. III). Sec. IV articulates the tangible implementation of the formal model we devised, while Sec. V delves into a comprehensive detailing of the research results. Sec. VI sheds light on the limitations of our approach, while in Sec. VII, we compile related work. To conclude, a concise summary of this research work is presented in Sec. VIII.

II. Background

A. Framework of Matrix

Matrix is a federated system that mainly includes the homeservers, rooms, and users, and the framework as illustrated in Fig. 1. All communication between Matrix users takes place within the room, suppose a user (`Alice`) wants to communicate with others in the room, she first needs to send messages to her homeserver (*matrix.alice.com*) via the Client-Server API [11]. The homeserver will then use the Server-Server API [12] to synchronize messages to other homeservers in the room. Eventually, the destination users' clients (`Bob` and `Charlie`) will receive the synchronized messages from their homeserver. Besides, if there are users of other platforms in the room, e.g., a Telegram user bridged in by the bridge server deployed on Bob's homeserver, Bob's homeserver also demands to leverage the Application Service API [13] to sync messages to the bridge server. And the bridge will relay the data to third-party application clients.

Besides, for privacy-minded customers, Matrix rooms also support communication with E2EE based on the Olm cryptographic ratchets protocol [9], [14]. Every registered user in Matrix has a unique "user ID" (`MXID`) in the format `@user_id:domain`, and a user device (e.g., Android, iOS) that supports E2EE has its `DeviceKeys`, the public part of the DeviceKeys will be uploaded to their homeserver, and the private part will be retained in the local device. The DeviceKeys encompass an Ed25519 [15] fingerprint key pair

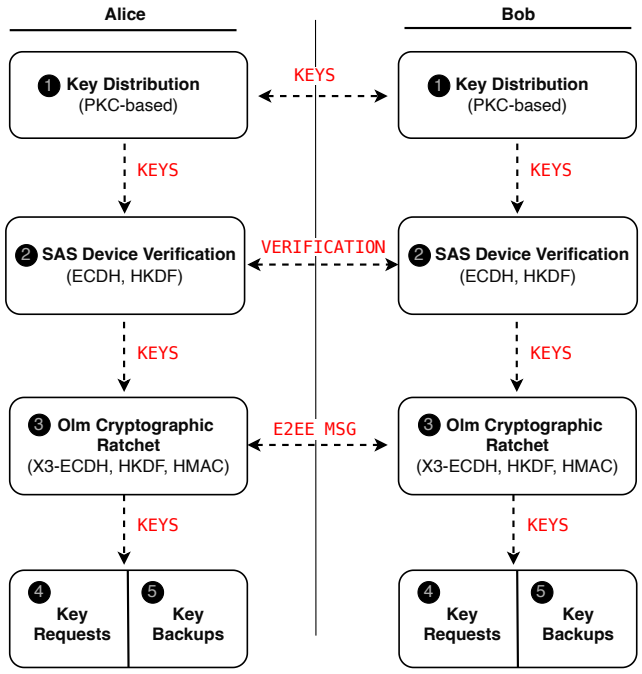

Fig. 2. Overview of the Matrix Security Protocols.

for signing messages, a Curve25519 [16] identity key pair, and Curve25519 one-time keys respectively. In particular, long-term Curve25519 identity keys and Curve25519 one-time keys enable authentication and establishment of shared key within Olm [8] sessions.

B. Security Protocols in Matrix

Matrix has designed a variety of security protocols at different levels to safeguard the overall network security. These levels include key-level security, device-level security, and encrypted message session-level security protocols among users. As illustrated in Figure 2, the security protocols at the key level primarily encompass key distribution protocols (❶) founded on public key cryptography (PKC) [17]. These protocols involve key generation, upload, claim, query, etc., to guarantee the secure dissemination of keys. Additionally, key requests protocol (❹) and key backups protocol (❺) are employed to verify the identity authenticity and ensure the security of user keys within the network. Besides, the security protocol at the device level employs a Short Authentication String (SAS) based device verification protocol [10] (as depicted in ❷, Fig. 2) to authenticate user devices and ensure their authenticity. Regarding the encrypted message session level among users, Matrix also has devised a cryptographic ratchet protocol based on Olm [8] (❸ of the Fig. 2). This protocol ensures the confidentiality, PFS, and PCS of end-to-end encrypted message transmission within the system.

Key Distribution. The Key distribution protocol primarily encompasses key generation using public key cryptography, key upload for identity keys generated, key claiming for one-time keys generated, and key querying to retrieve the keys of the other device. The Matrix client employs key generation algorithms like Ed25519 and Curve25519 to create a key pair. Subsequently, it uploads or claims the public part of the user-generated identity key pair and the one-time key pair to its homeserver for future use in key agreement protocols. To establish secure end-to-end encrypted communication with another user, Bob, Alice must initially query Bob's homeserver via her own homeserver to obtain Bob's public key.

Device Verification Protocol. Device verification based on SAS (which is heavily inspired by Phil Zimmermann's ZRTP key agreement handshake [18]) ensures the authenticity of both parties involved. The underlying concept involves the two parties querying each other's public key, creating a shared key through a key agreement protocol, and subsequently generating a unique shared key using the SAS algorithm, which may include emojis or numbers for distinctiveness. Finally, both parties verify the consistency of the generated SAS through an out-of-band method, such as in-person or video-based verification. If the SAS values are consistent, it confirms the authenticity of both parties' devices.

Olm Protocol and Megolm Protocol. To ensure the confidentiality of end-to-end encrypted messages and achieve both perfect forward secrecy (where the compromise of the current key does not impact the confidentiality of previously encrypted messages) and post-compromise security (where the compromise of the current key does not affect the confidentiality of subsequent encrypted messages), Matrix has introduced a cryptographic ratchet protocol based on Olm [8]. The core idea revolves around using the identity key and the one-time key to generate a shared key through multiple key agreements. Following this, the ratchet key and message encryption key are derived from this shared key, ensuring the security of end-to-end encryption. The inclusion of a one-time key adds an additional layer of assurance for both forward and backward confidentiality of encrypted messages. For a detailed explanation of the Olm protocol procedures, refer to Section IV-C. On the flip side, the Megolm cryptographic ratchet protocol [9] is designed to secure the confidentiality of end-to-end encryption, particularly in situations with multiple users' group communication within an encrypted room. It's crucial to highlight that Megolm is built upon the framework of Olm [19]. Therefore, in this study, our primary focus is on analyzing the security aspects of the Olm protocol.

Key Requests. In the event that a user transitions to a new device and wishes to access encrypted messages from their previous device, Matrix has implemented a key request protocol. In this sub-protocol, the new device can request the keys by dispatching a message, such as `m.room_key_request`, to other devices. Following the successful verification of the new device, the old device will forward the room session keys (`m.forwarded_room_key`) to the former.

Key Backups. To enhance resilience against the potential loss of keys, the Matrix security protocol incorporates a dedicated key backup protocol. This mechanism empowers users to encrypt their session key and securely transmit it to the homeserver for storage. The users can then utilize the key API (e.g.,

`GET /_matrix/client/v3/room_keys/keys`) to retrieve the stored session key from the homeserver, ensuring a resilient and secure method for safeguarding keys against the risk of loss.

Next, we will present the threat models and the essential security properties needed for the formal analysis of the Matrix security protocols.

III. Proposed Threat Models and Security Properties

Threat models delineate the array of potential threats and attacks encountered during the design, analysis, and utilization of cryptographic protocols and algorithms. A meticulously crafted threat model serves as a valuable tool in assessing the potential security risks inherent in a protocol, guiding the formulation and implementation of requisite security measures. Concurrently, security properties denote essential characteristics, such as confidentiality, that a cryptographic system or protocol should embody to ensure robust resilience against diverse forms of attacks. Subsequently, we will provide a comprehensive exploration of the threat models confronting the Matrix security protocol (Sec. III-A), and additionally, we are going to delve into the security properties integral to the protocols (Sec. III-B).

A. Threat Models

Within the realm of standard cryptographic protocol analysis, the Dolev-Yao attack model [20] stands out as one of the prevailing threat models. And given the decentralized nature of the Matrix security protocol, which involves multiple participants and roles, we have introduced three threat models in distinct scenarios based on the Dolev-Yao model. This approach allows for the modular analysis of various security protocols, leading to a more comprehensive formal verification of the entire protocol. In Fig. 3, we present the "Server Security Compromised", "External Security Compromised", and "Compulsory Access" threat models in a detailed manner.

- The **Server Security Compromised** threat model unfolds in scenarios where the server undergoes compromise or functions as a Byzantine server. In such situations, attackers can attempt to infiltrate the server through various avenues, seeking unauthorized access and subsequently engaging in malicious activities, such as the substitution or alteration of user key information. This threat model operates under the assumption of a weakened compromised precondition. Within the spectrum of risk vulnerabilities encountered by the Matrix security protocol, this particular threat model is acknowledged as prevalent. It manifests notably in protocols like the SAS-based device authentication protocol and the Olm-based password ratchet protocol.
- The **External Security Compromised** threat model addresses scenarios where external transmissions are compromised, leading to potential vulnerabilities in the communication link within the system. Adversaries may attempt interception, tampering, or eavesdropping on sensitive information during data transmission. This model also includes potential attacks on key management systems, seeking unauthorized access to encrypted communications through methods like key guessing, theft, or forgery. It is a recurrent concern in Matrix security protocols, particularly impacting key distribution, key requests, and key backup processes.
- The **Compulsory Access** threat model envisions scenarios involving the brief takeover of an endpoint device (full-compromised) or compromise by malicious apps on the phone (semi-compromised). This takeover could happen during border inspections or situations where adversaries use aggressive techniques like brute force attacks, malware, or other forceful means to gain unauthorized access to the endpoint. In the context of Matrix protocols security, the key request protocol is crucial for securing communication channels by facilitating the exchange and verification of cryptographic keys. The Compulsory Access threat model is relevant here as it assesses the protocol's resilience in scenarios where endpoints could face temporary control or persistent unauthorized access attempts.

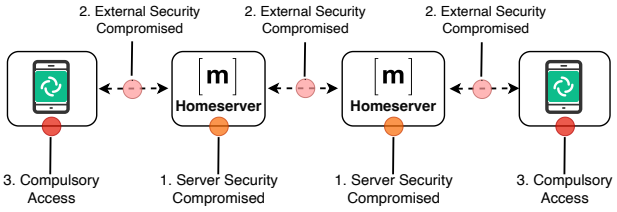

Fig. 3. Threat models of the Matrix security protocols.

In our endeavor to bolster the security analysis of the Matrix security protocol, we have meticulously crafted diverse threat models. These models are intricately tailored to encapsulate the distinctive characteristics inherent in the Matrix security protocol. By doing so, we aim to provide a more granular and insightful representation of potential adversaries and the specific attack methodologies they might deploy. Furthermore, leveraging the well-defined security properties, we are empowered to discern and delineate the potential traces and methodologies that an attacker could exploit.

B. Security Properties

In our investigation of Matrix security protocols through formal analysis, our primary focus centers on evaluating the security properties of Confidentiality, Authenticity, Perfect Forward Secrecy (PFS), and Post-Compromise Security (PCS). The details are outlined below.

- **Confidentiality** ensures that communication information is only accessible to authorized entities, preventing unauthorized parties from understanding or obtaining sensitive data. In the Matrix security protocol, a robust approach to confidentiality is achieved through the use of public key cryptography. This not only ensures secure communication but also prevents eavesdropping or unauthorized data access. Advanced cryptographic protocols, including key

distribution and cryptographic ratchet protocols, contribute to realizing confidentiality in the Matrix security protocol.

- **Authenticity** guarantees the genuineness of an entity's identity involved in communication, be it the sender of a message or the generator of data. This verification process serves as a safeguard against impersonation and identity theft, ensuring the trustworthiness of the communication participants. Within the Matrix security protocol, upholding authenticity stands out as a paramount security imperative. The Matrix security protocol employs a diverse array of mechanisms to secure this critical property. Notable among these mechanisms is the SAS-based device authentication protocol, a robust authentication framework designed to establish the authenticity of devices within the network. Additionally, the protocol leverages the Diffie-Hellman-based key agreement protocol, further fortifying the assurance of authenticity during key exchange processes.
- **Perfect Forward Secrecy** (PFS) is a crucial security feature in the Matrix security protocol, ensuring the confidentiality of past communications even if long-term keys are compromised. Implemented meticulously in the Olm-based cryptographic ratchet protocol, PFS is integral to securing sensitive information across the Matrix network. In each end-to-end encrypted session between Matrix users, a careful approach is taken to achieve PFS by utilizing independent temporary keys, isolating and safeguarding communication content. The integration of a key exchange protocol, such as the Diffie-Hellman key exchange, further strengthens PFS by negotiating session-specific keys.
- **Backward Secrecy** or **Post-Compromise Security** (PCS) is a crucial security measure that guarantees the privacy of past communications, even in the event of a compromise to certain keys within the system, such as long-term identity keys. In the pursuit of achieving this robust security property, the Matrix security protocol introduces the Olm-based key ratchet protocol to ensure the PCS. By employing a sophisticated mechanism, this protocol negotiates session keys between temporary keys and long-term keys.

Next, we are poised to present the tailored and implemented formal modeling method, designed to tackle the distinct threat models and security properties inherent in our proposed Matrix security protocol.

IV. A COMPREHENSIVE FORMAL MODEL OF MATRIX SECURITY PROTOCOLS

Given the intricate nature of the decentralized Matrix security protocol, which encompasses a multitude of protocols, and necessitates the involvement of diverse roles and participants in each specific protocol, the formalization of the Matrix security protocol is faced with heightened challenges. Hence, we have meticulously developed and implemented a formal verification methodology rooted in modularity. This modular-based approach has been strategically designed to accommodate diverse threat models, offering the versatility needed to evaluate and verify the unique security properties associated with different Matrix security protocols. And our proposed modular-based formal verification method is to ascertain whether different Matrix security protocols exhibit the requisite security properties under various threat models.

Following this, we delve into a comprehensive analysis of the protocol flow for five specific Matrix security protocols, namely, *Key Distribution*, *SAS-based Device Verification*, *Olm Cryptographic Ratchet*, *Key Requests*, and *Key Backups* protocols. Subsequent to this meticulous analysis, we embark on the formulation and implementation of their formal models utilizing ProVerif [21]. ProVerif is extensively employed within the realm of computer science for formal verification tasks, notably for protocols and systems. Its utilization has played a pivotal role in driving progress across multiple domains, such as IoT security [22], and Bluetooth security [23].

A. Modeling Protocols for Key Distribution

In this subsection, we distill the protocol logic from the Matrix specification [6] and provide a comprehensive depiction of the intricate key distribution protocol process integrated into the Matrix security protocol sub-protocol, as illustrated in Figure 4. This complex procedure evolves through various crucial stages, including key generation, identity key upload, one-time key claim, and key query. Each of these stages plays a pivotal role in safeguarding the resilience and effectiveness of the overarching security framework.

As depicted in Figure 4, the protocol involves four distinct participants: the sender, the receiver, the sender's homeserver, and the receiver's homeserver. Furthermore, the communication exchange among these participants adheres to our proposed External Security Compromised threat model. In the context of this model, every entity engages within an insecure transmission channel, underscoring the significance of addressing potential external security threats. The core of information exchange among participants is centered on the primary dissemination of public key components emanating from the generated key pairs, and we denoted as `pk_A_I`, `pk_A_O`, `pk_B_I`, and `pk_B_O`. This fundamental aspect forms a critical facet of the communication process, establishing a secure foundation for the transmission and reception of cryptographic keys within the networked environment. Hence, we undertook a comprehensive formalization of the various operations inherent in the protocol, meticulously capturing and modeling each distinct behavior. Our overarching goal is to scrutinize and assess whether the key distribution protocol aligns harmoniously with the security properties articulated in our earlier discussions.

Listing 3 in Appendix offers a visual representation that delineates the intricacies of the highly efficient implementation process within our formal model for the key distribution protocol using ProVerif. Every participant, meticulously crafted and thoughtfully designed in our framework, is individually instantiated as a process macro, exemplified by entities such as `clientA`. Within the modeled framework, every participant is endowed with a suite of functions designed to seamlessly handle both the transmission and reception of messages (e.g., `out` and `in`). These functions empower each participant

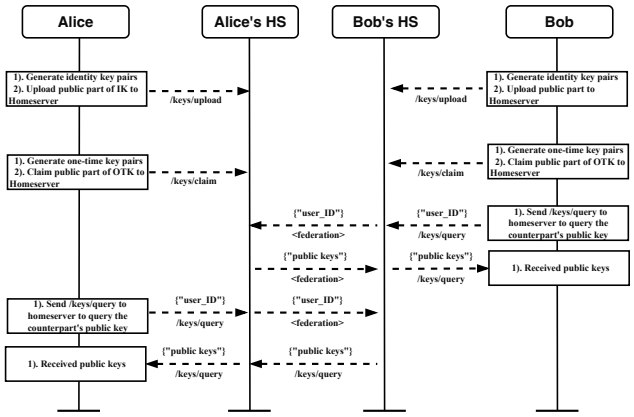

Fig. 4. We condensed the particulars of the Matrix key distribution protocol to direct and influence the modeling of this security protocol module in ProVerif.

to actively engage in the intricate communication processes inherent in the key distribution protocol. Moreover, the client role assumes a pivotal position in the ecosystem. Beyond its fundamental role as a message conduit, it possesses the unique capability to generate cryptographic keys, as demonstrated by the introduction of the `new` operation in Line 13. This pivotal functionality aligns with the dynamic nature of the key distribution protocol, where secure key generation is a fundamental aspect. The client is further empowered to execute a diverse array of actions within the protocol. Notably, Lines 14-16 showcase its ability to `upload` keys, contributing to the secure dissemination of cryptographic information. And the client role demonstrates a sophisticated capacity to `query` keys, as exemplified by operations conducted in Lines 20-24.

Besides, we establish various security properties for validation within the protocol's verification model, illustrated by lines 56-57 of Listing 3. For the involved parties in the protocol, the model delineates events such as Bob receiving Alice's identity key and the public key segment of the one-time key (`pk_I`, `pk_E`), termed as `evRecieveAliceKeys`, aimed at authenticating the key distribution protocol. Conversely, events pertaining to Alice's uploading of the identity key and the public key segment of the one-time key are labeled `evUploadAliceKey` and `evClaimAliceKey`. Our verification model ensures authenticity across different protocol entities by validating the correspondence between the public key received by Bob and the key uploaded or declared by Alice. In the last of our formal modeling, we intricately intertwine the unique functionalities of each participant, encapsulating the entirety of the system within the construct denoted as `!clientA|!serverA|!serverB|!clientB`. This amalgamation ensures a cohesive representation to query a basic correspondence assertion of the key distribution protocol. Besides, the modular design of our framework adds a layer of versatility, allowing for the seamless migration of this module to contribute to the modeling implementation of various other Matrix security protocols.

B. Modeling Protocols for SAS-Based Device Verification

We have streamlined and formalized the SAS-based device verification protocol, and its procedure is illustrated in Figure 5. In our considerations, we explored various scenarios. Ideally, the server should be limited to the receiving, querying, and forwarding functions of the Matrix API. Meanwhile, the client should possess additional capabilities, including generating temporary keys, completing key negotiation, and generating short authentication strings (emoji or number).

At the beginning of the process, client Alice initiates a device verification request to Bob. Once Bob agrees and is prepared, both parties mutually confirm the chosen SAS method, hash function, key agreement protocol, and message authentication code algorithm. Following that, both entities acknowledge and produce their respective temporary key pairs (e.g., `skey_B_E`, `pkey_B_E`). They transmit the public key component to the other party (m.key.verification.key) and generate a shared key based on the key agreement protocol. Ultimately, both entities create a SAS using the information received from the other party (including public key, user ID, device ID, etc.). They employ an out-of-band method to verify the legitimacy of the other party's device. Once confirmed, both parties utilize `ourinfo` and `theirinfo` to independently generate Message Authentication Codes (MAC) and transmit them to each other, concluding the device authentication process.

Utilizing the representation in Figure 5, we have developed and executed a formal model for SAS-based device verification. We have delved into diverse threat models, including Server Security Compromised and External Security Compromised. The abstraction of deployment has been conducted, as exemplified in Listing 1. Within our system, we define a dedicated private out-of-band SAS acknowledgment channel, explicitly outlined in Line 2 of the Listing 1. This channel serves as a secure communication pathway for the acknowledgment of Short Authentication Strings. Additionally, we designate a shared key specifically tailored for private attributes, denoted in Line 8 of the Listing 1, ensuring the confidentiality and integrity of sensitive information. Moreover, both `clientA` and `clientB` are endowed with a versatile set of capabilities. They possess the capacity to engage in a myriad of operations, such as initiating and responding to requests, commencing and accepting processes, managing cryptographic keys, facilitating the generation of a shared key (as delineated in Line 29), and undertaking the computation of SAS (outlined in Line 33), among other functionalities. On the other hand, `serverA` and `serverB` are restricted to the roles of mere recipients and forwarders within the system.

Ultimately, we specify events characterized by distinct verifiable security properties, including authenticity and confidentiality, as illustrated in Lines 49-52 of Listing 1. More precisely, the events `evClientAliceCalcShareSecret` and `evClientAliceCalcSAS` ascertain whether the shared key and SAS computed by Alice, the communicating entity, indeed utilize the temporary public key transmitted by

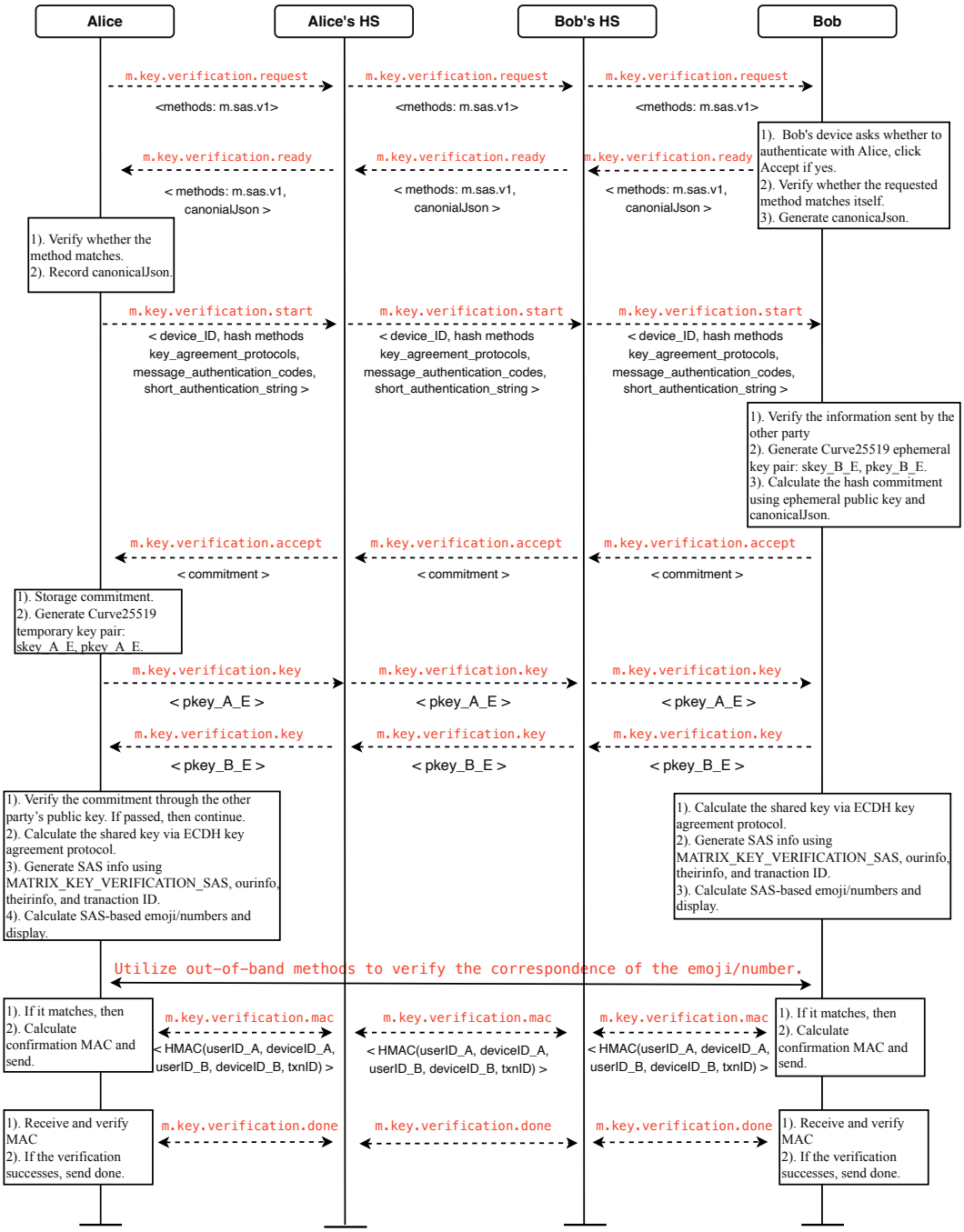

Fig. 5. The specifics of the Matrix SAS device verification protocol and employed as a guide for implementing the modeling of the security protocol module.

Bob, the counterpart entity. This process aims to authenticate the different entities involved in the protocol. Additionally, our modeling approach includes verifying the confidentiality of the SAS device authentication protocol. This is achieved by introducing the query event `attacker(sas)` to assess whether the SAS value is compromised or leaked. The process is initiated using `!clientA|!serverA|!serverB|!clientB` as the designated execution entry point.

C. Modeling Protocols for the Olm Cryptographic Ratchet

The procedural steps of the Olm cryptographic ratchet protocol within the Matrix security framework are consolidated and presented in Figure 6. This protocol is designed to uphold both forward secrecy and backward secrecy (as shown in Sec. III-B) for end-to-end encrypted messages exchanged between users. The Olm protocol is primarily segmented into three key components: the initialization phase, the encryption phase, and the decryption phase. The protocol's initialization

```
// SAS out-of-bound (oob) public key channel
free oob_sas_ch: channel [private].
// type definition
type methods.  (* m.sas.v1 *)
type msgtype.  (* m.key.verification.XX *)
...
// Elliptic Curve Diffie-Hellman Key Exchange.
free sharedsecret_A: key [private].
free sharedsecret_B: key [private].
letfun get_ecdhkey(sk_E_local:skey, pk_E_remote:
    pkey) = sca(..., ...).
// Authenticity events
event evClientAliceSendSAS(bitstring).
event evClientAliceCalcShareSecret(key).
...
// Client Alice can send and receive.
let clientA(skey_A_E:skey) =
(
// (* Alice send m.key.verification.request *)
    out(ch_c2s, (actionRequest, methodA));
// (* Alice send m.key.verification.start *)
    out(ch_c2s, (actionStart, methodA, KGP_A,
    hash_method_A, mac_method_A, sas_method_A));
// (* Alice receive m.key.verification.accept
    from Bob *)
    in(ch_s2c, (actionAccept: event_action, ...,
    commitmentB: hash));
// (* Alice send m.key.verification.key to Bob *)
    out(ch_c2s, (actionKey, pkey_A_E));
// (* Generate and verify commitment *)
    let calc_commitmentB = SHA256(...) in
// (* calculate ECDH *)
    let sharedsecret_A = G2key(get_ecdhkey(
    skey_A_E, pkey_B_E)) in
    event evClientAliceCalcShareSecret(
    sharedsecret_A);
// (* Generate SAS info *)
    let sasInfo = generate_sasInfo(
    MATRIX_KEY_VERIFICATION_SAS, ourInfo,
    theirInfo, txnId) in
// (* calculate SAS *)
    let bytes_6_A = generate_bytes(sharedsecret_A
    , sas_method_A, sasInfo) in
// Send SAS in out-of-bound channel
    out(oob_sas_ch, emojis_A);
    event evClientAliceSendSAS(emojis_A);
...).
// Server Alice can receive and forward messages.
let serverA() =
( ... ).
// Server Bob can receive and forward messages.
let serverB() =
( ... ).
// Client Bob can generate, compute, send, and
    receive messages just like client Alice.
let clientB(skey_B_E:skey, transactionId:
    bitstring) =
( ... ).
// (* authenticity properties *)
query pkey_E:pkey, sharekey:key; event(
    evClientAliceCalcShareSecret(sharekey)) ==>
    event(evClientBobSendKeys(pkey_E)).
query pkey_E:pkey, sas:bitstring; event(
    evClientAliceCalcSAS(sas)) ==> event(
    evClientBobSendKeys(pkey_E)).
// (* confidentiality properties *)
query attacker(sas).  (* SAS info *)
...
// Runs the process.
process ( !clientA(sk_A_E) | !serverA() | !
    serverB() | !clientB(sk_B_E) )
```

Listing 1. Implementation of formal modeling of SAS-based device verification module.

encompasses a key distribution process, details of which have been expounded upon in Section IV-A.

When user Alice seeks to establish end-to-end encryption with user Bob, a key negotiation process is initiated. The Olm protocol facilitates this negotiation by executing Elliptic-Curve Diffie-Hellman (ECDH) three times. This involves utilizing one's own identity key and the private component of the temporary key, along with the other party's identity and the public component of the temporary key, ultimately generating the shared secret. Following this, both entities can employ the shared key to derive the root key, chain key, and message key for the cryptographic ratchet using the HKDF algorithm. At last, leveraging the message key, the HKDF function is once again executed to derive encryption keys, including the AES key, HMAC key, and AES IV. Notably, for encrypting new messages, the client must generate a fresh one-time key to facilitate the negotiation of a shared secret and update the ratchet chain key. This mechanism ensures both forward and backward secrecy for encrypted messages exchanged between users. Upon receiving a message encrypted with the encryption key from the counterparty, the user can decrypt it using the locally derived asymmetric encryption's decryption key.

As depicted in in Listing 4 of the Appendix, we present a section of the ProVerif code that has been employed to model the Olm cryptographic ratchet protocol for the Matrix framework. Our modeling approach incorporates the use of `table` as an interface for delineating various procedural steps. This design enables the seamless integration of information from one step to the next, with the table content from preceding steps serving as input for subsequent stages in the formal modeling process. By employing this structured and interconnected approach, we aim to accurately capture the intricacies of the Olm cryptographic ratchet protocol within the formal framework, facilitating a comprehensive analysis of its security properties and potential vulnerabilities.

Within the protocol, a crucial element is the Triple-ECDH key agreement algorithm, serving as a linchpin for secure communication. In our modeling process, we intricately delineate the specific mechanisms governing this algorithm in `step1client`, meticulously capturing the intricacies and nuances involved (as illustrated in Line 16 of Listing 4). This granular representation allows us to dissect and understand the algorithm's behavior at a detailed level. Building upon this foundation, we seamlessly integrate the output of `step1client` into `step2client`, recognizing the pivotal role it plays in the larger scheme of the protocol. Drawing from the negotiated shared key, the subsequent step, `step2client`, involves the computation of various keys essential for encryption. This encompasses the derivation of keys such as `rootKey`, `aesKey`, `hmacKey`, `aesIV`, and others, ensuring a comprehensive suite of cryptographic elements to facilitate secure communication.

To finalize the encryption process, we utilize the derived keys, as previously established, to employ the `senccbc` (AES-256 in CBC mode) at Line 40 of Listing 4. This step ensures the secure transformation of the plaintext message into

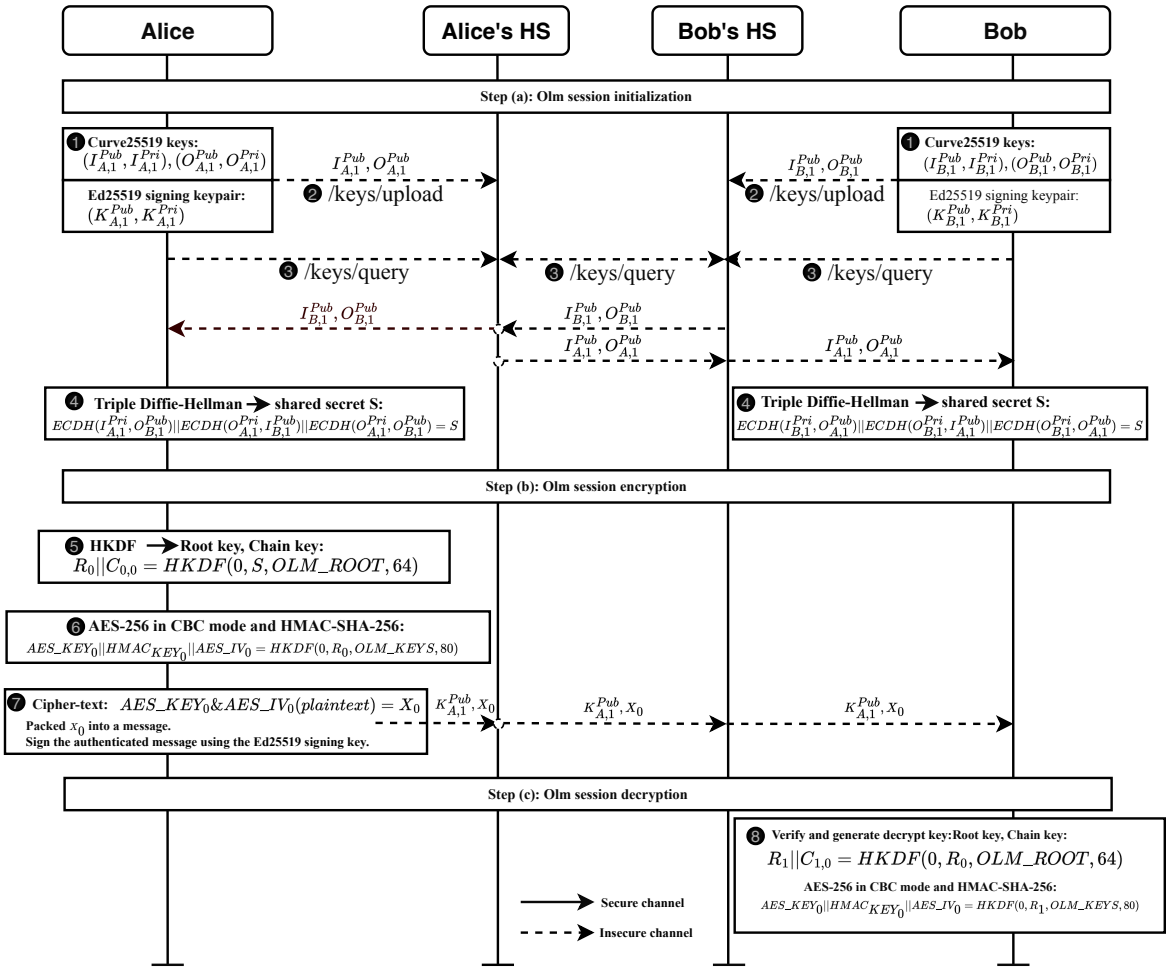

Fig. 6. The specifics of initializing, encrypting, and decrypting within the Matrix Olm cryptographic ratchet protocol are succinctly summarized.

a ciphertext, safeguarding its confidentiality during transmission. On the receiving end, decryption is orchestrated using the `sdeccbc`, as indicated at Line 51. This cryptographic operation ensures the secure retrieval and reconstruction of the original plaintext message from the received ciphertext.

In order to verify various security properties, encompassing confidentiality, authenticity, PFS, and PCS, we incorporate a series of query statements (Lines 52-55 in Listing 4) within the Olm cryptographic ratchet protocol. These queries function as a pivotal mechanism for gauging the protocol's alignment with and fulfillment of the predefined security attributes. Ultimately, we utilize the `process` command to execute the modular formal verification program that we have meticulously designed. This execution step serves as the conclusive phase in our comprehensive evaluation of the Olm cryptographic ratchet protocol, ensuring the thorough examination of its adherence to the specified security properties.

D. Modeling Protocols for Key Requests

Suppose a user possesses multiple devices and the new device does not possess the decryption key for a message, it has the option to seek the session key. This is achieved by transmitting an `m.room_key_request` to the old device, which is the device holding the session key. If the device holding the key decides to grant access to the new device, it can achieve this by transmitting an encrypted `m.forwarded_room_key` message to the requesting device. Thus, within this subsection, we formal two distinct threat models for key request protocols. The initial model addresses situations where external security is compromised (External Security Compromised), while the second model addresses threat scenarios involving forced access (Compulsory Access threat model).

1) Modeling Key Request Protocol in External Security Compromised: As depicted in Listing 7 of Appendix, we have streamlined and illustrated the key request protocol procedure under an External Security Compromised scenario. And in our implementation of modeling, we initiate the room session key request action, `RoomKeyRequest` (Line 9 of Listing 7), and send it to the recipient. Upon reception by the recipient, if the requester is identified as an authenticated device (as indicated on Line 24), which indicates that it has been authenticated by the SAS device verification, the key will be relayed through `RoomForwardedKey` (Line 25 of Listing 7). Ultimately, we

define various security properties, such as the confidentiality of the session key and the authenticity of the requesting device. We examine whether the key request protocol meets these properties, as exemplified by Lines 32-33 of the Listing 7.

2) Modeling Key Request Protocol in Compulsory Access: Within this section, we introduce an additional formal modeling scenario for the key request protocol, specifically, the threat model involves the hijacking of user devices. This involves a case where an attacker has the capability to clone the victim device. Subsequently, we will proceed to model the design and deployment considerations within this threat model.

Our modeling approach begins by assuming that Alice and Bob have established a session. Under our proposed threat model, the attacker has the capability to inject basic information from Alice's device (including the `identityKey`, `deviceID`, `userID`, etc., but have not permission to access the administrator password) into the `table` (Line 8 of Listing 5 in Appendix), treating it as clone information for the new device. Subsequently, when Alice Cloner initiates the client for the new device and is unable to decrypt the preceding message, she sends a `m.room_key_request` (Line 29) to Alice's original device. Upon receiving the request, Alice responds by sending `m.forwarded_room_key` (Line 23). Furthermore, to comprehensively model the Compulsory Access threat model, as demonstrated in Lines 36-38 of Listing 5, we also modeling the Alice Cloner can establish a new session with Bob, Bob transmits an encrypted message to both Alice and Alice Cloner, and message encryption and decryption by Alice Cloner. Ultimately, we articulate distinct `query` events to assess security properties, including authenticity, confidentiality, perfect forward secrecy, and post-compromise security as specified in lines 52-58 of Listing 5.

E. Modeling Protocols for Key Backups

In this subsection, we depicts our modeling of the key backup protocol as shown in Listing 6 of Appendix. Alice's device (`aliceDevice1`, as shown in Line 14 of Listing 6) can upload an encrypted room session keys to her homeserver. In the event that another device affiliated with Alice (`aliceDevice2` at Line 30) endeavors to access a message without possessing the requisite key, said device is empowered to request the necessary key from the server and subsequently decrypt the message. Unlike key requests, server-side key backup doesn't rely on another device being online. Nevertheless, due to the encrypted storage of the session key on the server, the user is required to input the decryption key in order to decrypt the session key. On one hand, to create a backup of the room session keys, the device `POST /_matrix/client/v3/room_keys/version` request to homeserver, and define the encryption algorithm for room session keys through the backup's `auth_data` (Lines 15-18 of Listing 6). Then the keys are encrypted via the `auth_data`, and `PUT /_matrix/client/v3/room_keys/keys` (Lines 20-24) request to the server. On the other hand, devices belonging to Alice can retrieve the backup by invoking `GET /_matrix/client/v3/room_keys/version` (Lines 31-34). and can be acquired via the `GET` request to `/_matrix/client/v3/room_keys/keys` (Lines 35-39). In our designed formal module, we implement authenticity and confidentiality properties (Lines 41-47 of Listing 6) to verify the key backups protocol.

V. Formal Verification Results and Findings

An overview of the formal verification results of the Matrix security protocols under different security properties are showing in Table I. Next, we will specifically describe the security properties violated and specific attack traces for Matrix security protocols under different threat models.

A. Key Distribution

Matrix's key distribution protocol encompasses essential operations, including generation, upload, claim, and query. Leveraging public key cryptography, these operations uphold security properties such as authenticity, ensuring the protection of encrypted messages exchanged among Matrix users [24]. To assess the protocol's adherence to these security properties, we implemented an automated formal modeling verification approach using ProVerif (as detailed in Section IV-A). This method was implemented within an external threat model that simulated security compromises. The formal model primarily focuses on verifying the key upload and key claim aspects of the key distribution protocol. Additionally, it examines the authenticity of users, ensuring that the keys queried by users, such as Alice and Bob, genuinely belong to one another.

As exemplified by Line 57 in Listing 3 of Appendix, we provide a detailed illustration by considering a scenario where Bob queries Alice's key. Upon conducting an automated model verification, a critical revelation emerged, exposing a deficiency in the authenticity property within the Matrix key distribution protocol. This discrepancy is explicitly highlighted in Line 14 of Listing 8. Specifically, the recorded event of receiving Alice's keys (`evReceiveAliceKeys`) does not align with the events of Alice uploading (`evUploadAliceKey`) and claiming (`evClaimAliceKey`) those keys.

Besides, we are analyzing the attack trace generated by ProVerif, as outlined in Listing 8, unveils a potential security threat where an adversary can compromise the authenticity property of the key distribution protocol through specific steps. On the one hand, within the threat model of an external security compromise, an attacker possesses the capability to intercept the transmission of crucial elements, including the public key segment of the identity key, the public key segment of the one-time key, and the user ID (depicted as `attacker((userID,pk_I,pk_E))`) exchanged between the involved parties. On the other hand, when client Bob requests Alice's key, the attacker can manipulate and provide Bob with the `(userID, pk_I, pk_E)` information (Lines 10-12 of Listing 8). This manipulation paves the way for the successful execution of a man-in-the-middle (MitM) substitution attack, undermining the authenticity of the key

TABLE I
AN OVERVIEW OF THE FORMAL VERIFICATION RESULTS OF THE MATRIX SECURITY PROTOCOL UNDER DIFFERENT SECURITY PROPERTIES. ✗ INDICATES THAT A SECURITY PROPERTY HAS BEEN COMPROMISED, ✓ INDICATES NOT.

#	Matrix Security Protocols	Security Properties				Attacks
		Authenticity	Confidentiality	PFS	PCS	
1	Key Distribution	✗	✓	✓	✓	MitM
2	SAS-based Device Verification	✗	✓	✓	✓	Impersonation Attack
3	Olm Cryptographic Ratchet	✗	✓	✗	✗	Impersonation Attack
4	Key Requests – 1	✗	✓	✓	✓	MitM
	Key Requests - 2	✗	✗	✗	✗	Clone Attack
5	Key Backups	✗	✓	✓	✓	MitM

exchange and posing a significant security risk within the key distribution.

Finding 1: The key distribution protocol of Matrix is vulnerable to man-in-the-middle attack, particularly in the key query phase, jeopardizing the authenticity security attributes of the protocol.

B. SAS-based Device Verification Protocol

The Matrix specifications [6] introduce a SAS-based device verification protocol with the goal of authenticating devices on both ends. The protocol involves comparing the generated SAS (represented by emojis or numbers) using out-of-band methods. If both parties reach an agreement, the verification process is considered successful. When authenticating user devices, this protocol ensures security properties encompassing confidentiality and authenticity. Specifically, the outcome of the negotiation involving the `sharekey` and `sas` is both confidential and authentic. To validate the aforementioned security properties, we formulated rules outlined in lines 48-52 of Listing 1 within the framework of the "Server Security Compromised" threat model. In particular, authenticity is confirmed by validating whether the shared key and SAS computed by both users indeed result from negotiations with the counterpart. Simultaneously, the verification involves checking the potential leakage of generated SAS information to ensure confidentiality.

Following formal model verification, it was observed that the protocol adheres to the security guarantee of confidentiality but does not meet the security property of identity authenticity. Consider the example where Alice's device verify Bob (whose homeserver has been compromised). In this case, the SAS calculated by Alice (`evClientAliceCalcSAS`) is legitimately derived from Bob's key (`evClientBobSendKeys`). The corresponding attack trace is illustrated in Listing 2. To exploit vulnerabilities and compromise the authenticity security properties outlined in Table I, an attacker can strategically execute a series of steps. Initially, the compromised homeserver gains access to the Short Authentication String (SAS) method (`SAS_V1`) utilized by both communication parties, along with obtaining the `Ready` status of both entities (depicted in Lines 4-5 of Listing 2). Furthermore, the attacker acquires critical information exchanged between the parties, including `key_agreement_protocols`, `hash_method`, `mac_method`, `sas_method`, and other details within the `m.key.verification.start` action.

Following this reconnaissance, when both parties signal acceptance and proceed to exchange the public key of the ephemeral key pair via the `m.key.verification.key` action, the compromised homeserver seizes the opportunity to initiate an impersonation attack. At this juncture, the attacker cunningly substitutes the victim's temporary key. Consequently, the key received by Alice (`evClientAliceReceiveKeys`, Line 15) is effectively controlled by the compromised homeserver, exerting influence over the authenticity of the shared key (`evClientAliceCalcShareSecret`, Line 16) and SAS (`evClientAliceGetSASInfo`, Line 17).

In the concluding phase, during the SAS calculation process by Alice (`evClientAliceCalcSAS`), the computation is grounded on the key surreptitiously replaced by the compromised homeserver. In this scenario, the attacker successfully assumes the identity of the victim's device, thereby compromising the authenticity attribute of the SAS-based device verification protocol.

Finding 2: The SAS-based device verification protocol is prone to a homeserver impersonation attack. The attacker can mimic the victim, associating the SAS with a different user and compromising the protocol's authenticity.

C. Olm Cryptographic Ratchet Protocol

The foundation for securing end-to-end encrypted messages among Matrix users lies in the Olm encryption ratchet protocol designed by Matrix. This protocol is intricately crafted to deliver various security properties, with a particular emphasis on ensuring confidentiality, authenticity, perfect forward secrecy, and post-compromise security. The Olm protocol relies on both one-time keys and long-term keys, utilizing triple elliptic curve Diffie-Hellman key agreement as the foundation for advancing two-party encrypted sessions. Utilizing newly

```
1  new skey_A_E: skey creating skey_A_E_2 at {1}
2  new skey_B_E: skey creating skey_B_E_2 at {2}
3  new transactionId: bitstring creating transactionId_3 at {3}
4  out(ch_c2s, (~M,~M_1)) with ~M = Request, ~M_1 = SAS_V1 at {9} in copy a
5  in(ch_s2c, (Ready,SAS_V1)) at {10} in copy a
6  new KGP_A: key_agreement_protocols creating KGP_A_5 at {12} in copy a
7  new hash_method_A: hashes creating hash_method_A_5 at {13} in copy a
8  new mac_method_A: message_authentication_codes creating mac_method_A_5 at {14} in copy a
9  new sas_method_A: short_authentication_string creating sas_method_A_5 at {15} in copy a
10 out(ch_c2s, (~M_2,~M_3,~M_4,~M_5,~M_6,~M_7)) with ~M_2 = Start, ~M_3 = SAS_V1, ~M_4 = KGP_A_5, ~M_5 =
       hash_method_A_5, ~M_6 = mac_method_A_5, ~M_7 = sas_method_A_5 at {17} in copy a
11 in(ch_s2c, (Accept,a_1,~M_4,a_2,~M_6,~M_7,SHA256(concat2(sca(a_3,P),a_1)))) with ~M_4 = KGP_A_5, ~M_6
       = mac_method_A_5, ~M_7 = sas_method_A_5 at {18} in copy a
12 event evClientAliceSendKeys(pk(skey_A_E_2)) at {24} in copy a
13 out(ch_c2s, (~M_8,~M_9)) with ~M_8 = Key, ~M_9 = pk(skey_A_E_2) at {25} in copy a
14 in(ch_s2c, (a_4,sca(a_3,P))) at {26} in copy a
15 event evClientAliceReceiveKeys(sca(a_3,P)) at {27} in copy a
16 event evClientAliceCalcShareSecret(sca(skey_A_E_2,sca(a_3,P))) at {32} in copy a
17 event evClientAliceGetSASInfo(concat4(MATRIX_KEY_VERIFICATION_SAS,concat3(userID_A,v,deviceID_A,v,pk(
       skey_A_E_2),v),concat3(userID_B,v,deviceID_B,v,sca(a_3,P),v),transactionId_3)) at {37} in copy a
18 event evClientAliceCalcSAS(generate_bytes(sca(skey_A_E_2,sca(a_3,P)),sas_method_A_5,concat4(
       MATRIX_KEY_VERIFICATION_SAS,concat3(userID_A,v,deviceID_A,v,pk(skey_A_E_2),v),concat3(userID_B,v,
       deviceID_B,v,sca(a_3,P),v),transactionId_3))) at {39} in copy a (goal)
19 The event evClientAliceCalcSAS(generate_bytes(sca(a_3,sca(skey_A_E_2,P)),sas_method_A_5,concat4(
       MATRIX_KEY_VERIFICATION_SAS,concat3(userID_A,v,deviceID_A,v,pk(skey_A_E_2),v),concat3(userID_B,v,
       deviceID_B,v,sca(a_3,P),v),transactionId_3))) is executed at {39} in copy a.
20 A trace has been found.
21 RESULT event(evClientAliceCalcSAS(sas)) ==> event(evClientBobSendKeys(pkey_E)) is false.
22 ------------------------------------------------------
23 Verification summary:
24 Query event(evClientAliceCalcSAS(sas)) ==> event(evClientBobSendKeys(pkey_E)) is false.
25 ------------------------------------------------------
```

Listing 2. Attack trace of the impersonation attack attack against the SAS-based device verification protocol.

generated encryption keys such as `aesKey`, `hmacKey`, and `aesIV`, the encrypted messages exchanged between users remain secure. Even in the event of a compromise to the current encryption key, the integrity of both prior and subsequent encrypted messages remains unaffected.

Thus, in order to validate the specified security properties, we developed a formal model targeting the Olm cryptographic ratchet protocol module, specifically under the "Server Security Compromised" threat model. This involved the establishment of query event rules, as illustrated by those delineated in Lines 52-55 of Listing 4. These rules are designed to scrutinize aspects such as verifying the sender authenticity of an encrypted message received by Alice from Bob and assessing the potential for an adversary to decrypt messages exchanged among users. Within our investigation, we uncovered a significant vulnerability within the Olm cryptographic ratchet protocol. In the event of a compromised homeserver, there exists the potential for executing impersonation attacks on its users. This involves the homeserver posing as the legitimate user, engaging in encrypted communication with other users. Such an exploit not only compromises the confidentiality and authenticity safeguards provided by the protocol but also undermines both PFS and PCS.

In this section, we focus on the event outlined in Line 21 of Listing 11 in Appendix as a representative example. In this scenario, the encrypted message received by Alice (`evClientAliceReceiveCipertext`) cannot be conclusively determined to be the same as the encrypted message sent by Bob (`evClientBobSendCipertext`). Examining the attack traces exposes a potential security breach where Bob's compromised homeserver possesses the capability to intercept keys sent by Alice and replace them with the attacker's keys, as demonstrated in events `evClientAliceSendKeys` (Line 4) and `evClientAliceReceiveKeys` (Line 7).

Further scrutiny reveals that during the Triple Elliptic Curve Diffie-Hellman (Triple ECDH) key agreement protocols between users Alice and Bob, the shared key undergoes manipulation by the attacker at the key agreement stage (`evClientAliceCalcShareSecret` of Line 8). This manipulation extends to the attacker gaining control over essential cryptographic parameters, including the `rootKey`, `chainKey`, `ratchetKey`, and the encryption key (`aesKey`), all calculated by the unsuspecting victim, Bob. The consequences of this orchestrated attack become more apparent as the attacker intercepts the encrypted message dispatched by Alice (`evClientAliceSendCipertext` in Line 13 of Listing 11) and successfully decrypts it. Moreover, the attacker seizes the opportunity to manipulate the ratchet keys for both parties by initiating `evClientAliceReceiveRatchetKey`, enabling them to impersonate Bob and transmit encrypted messages under his identity.

Finding 3: The Olm cryptographic ratchet protocol is vulnerable to impersonation attacks on compromised homeservers. An assailant has the capability to substitute the victim's identity key, allowing them to forge communications with others surreptitiously, compromising authenticity, as well as PFS and PCS.

D. Key Requests

The key request protocol [25] is beneficial in scenarios with users employing multiple devices. In cases where a new device struggles to decrypt messages from an old device, requesting the room session key from the old device aids in decryption. Maintaining device identity authenticity and key confidentiality is crucial for ensuring the security of message confidentiality in both forward and backward directions. The following presents verification results for two threat models in the analysis of the formal model of the key request protocol.

1) Key Requests in External Security Compromised: As depicted in Listing 10 of Appendix, we designed and implemented a formal model to verify the confidentiality of the protocol (ensuring the attacker cannot acquire the `sessionKey` during key requests, as indicated in Line 17) and to validate the authenticity (verifying the accuracy of the `session_key` request against the true identity of the key). The results from formal analysis of attack traces indicate a vulnerability in the protocol, specifically in its resistance against man-in-the-middle attacks. In instances where a new device initiates a key request action (`RoomKeyRequest`), the protocol is exposed to interception by an attacker. Subsequently, the man-in-the-middle assailant gains the ability to replace the transmitted information and proceed to request the room session key. While our assessment indicates a perceived low level of risk associated with this type of vulnerability, it nevertheless erodes the intended security properties inherent to the protocol.

2) Key Requests in Compulsory Access: In the presence of the compulsory access threat model illustrated in Figure 3, as examined through the formal model in Listing 5 of Appendix, it becomes evident that the key request protocol is susceptible to attacks (defined as clone attack). The clone attack can compromise the authenticity, confidentiality, perfect forward secrecy, and post-compromise security properties of the key requests protocol.

In delving into the nuanced dynamics of the security analysis, we find, on one hand, a distinctive proficiency exhibited by the cloner's device, as meticulously detailed in Line 53 of Listing 5 in Appendix. Notably, this device showcases the remarkable ability to decrypt messages (`evClonerCanDecryptFormerMessage`) that were encrypted in preceding interactions. Furthermore, the cloner's device can adeptly initiate a request to the victim's device, as evidenced by the event `evClonerGetRoomForwardedKey`. The primary objective of this request is to acquire the room session key associated with the aforementioned previously encrypted message. Adding a layer of complexity to this scenario, the victim, under the impression that the cloned device is verified, unwittingly transmits the key directly to the attacker through the channel `m.forwarded_room_key`. On the other hand, a thorough verification of the capabilities inherent in Alice's cloned device reveals a multifaceted nature. This device not only possesses the capability to initiate a new session (`evClientBobBuildSessionWithCloner`) with another user, such as Bob, but also demonstrates the remarkable ability to synchronize the encrypted messages exchanged between Alice and Bob thereafter.

Emphasizing a crucial aspect of our proposed attack methodology, it is imperative to note that our type of attack does not trigger the generation of an attack trace in ProVerif. To further validate and substantiate our findings, we devised a comprehensive validation experiment, drawing inspiration from [26]. In this experiment, we meticulously implemented a Linux virtual machine using VMware, meticulously replicating the procedures outlined in the referenced work. Subsequently, within this virtual environment, we installed Element-Web and Element-Linux [27], meticulously configuring the setup to mirror real-world scenarios. Leveraging VMware's replication function, we orchestrated a series of clone attacks to emulate and assess the vulnerabilities within the system. This experimental setup allowed us to observe and analyze the effects of clone attacks on the security attributes of the protocol. The culmination of our validation experiment revealed noteworthy outcomes, indicating that the security attributes of the protocol, encompassing confidentiality, authenticity, PFS, and PCS, are indeed susceptible to compromise through clone attacks.

Finding 4: The key requests protocol in Matrix exhibits vulnerability to clone attacks, wherein an attacker not only has the capability to send session key requests to the victim but can also initiate sessions with other users. This dual capability poses a significant threat, compromising both the PFS and PCS properties of the protocol.

E. Key Backups

Within the key backup protocol, it is imperative for the client to ensure the local security of its room session keys through encryption. Following this security measure, the client proceeds to backup these keys to the homeserver, thereby ensuring the prevention of any potential key loss [28]. As outlined in our formal model of Listing 6, we establish two security property queries within the context of the "External Security Compromised" threat model. These queries specifically address the authenticity of the key backup request received by user Alice's homeserver and the confidentiality of the data encapsulated in the client's `key_backup_data`, ensuring its resilience against any potential leakage. Following the verification of the formal model, it was determined that the protocol can ensure the confidentiality of key backups. However, it revealed latent vulnerabilities in terms of authenticity. As illustrated in Listing 9 of Appendix, the attack trace generated by ProVerif highlights a potential

vulnerability wherein a man-in-the-middle attacker could intercept and substitute information (e.g., `BackupAlgorithm` and `auth_data`) returned by the user sending API `/_matrix/client/v3/room_keys/version`. Hence, when the server receives a key backup request, the protocol lacks a guarantee that it must originate from a benign user.

It is important to emphasize that, in our assessment, the associated risks with this protocol are deemed insignificant and are unlikely to result in significant vulnerabilities within the broader Matrix security protocol.

VI. Limitations

It is worth noting that, for the current scope of this paper, more intricate group communication scenarios have been intentionally omitted, although it is acknowledged that one-to-one messaging inherently constitutes a subset of group messaging. As an illustrative example, the formal modeling of the Megolm protocol is not addressed in Sec. IV-C, as Olm stands as an indispensable component within the former.

Besides, we have synthesized the instant messaging security design document from the Matrix specification, delineating key security aspects such as confidentiality, authenticity, perfect forward secrecy, and post-compromise security. While this design effectively fulfills the requirements for a secure message protocol, it is imperative to note that our formal model's security properties do not encompass factors such as reachability and availability (resistance to denial-of-service attacks). It is crucial to underscore that our designed formal method, specifically crafted for the purpose of verification, offers only partial support in validating additional security properties. This limitation arises, impacting the overall completeness of the verification process.

VII. Related Work

A. Decentralized Matrix E2EE Communication Network

The analysis of Matrix, a representative decentralized communication federation protocol, has also attracted extensive attention from researchers [1], [19], [29]–[31]. [1] proposed a forensic approach to dissecting the Riot.im (Element [32]) application and the Matrix protocol to fill the knowledge gap in forensics and analysis for the Matrix community of investigators. [30] have evaluated the structure of the public part in a Matrix federation as a basis for looking into the middleware scalability. The authors identify scalability issues for communication mechanisms in federations with different structures by investigating the network load distribution. Martin et al. [19] report several exploitable cryptographic vulnerabilities in the Matrix protocol used for federated communication and its flagship client (Element [32]) and prototype implementations. The disclosed vulnerabilities are varied (insecure design, protocol obfuscation, and lack of domain separation) and are widely spread across different subprotocols and libraries of the cryptographic core. Hao et al. [31] have not only conducted theoretical analyses but have also implemented a comprehensive study. They explore the Matrix ecosystem from various viewpoints, specifically focusing on the landscape, network threats, and implementation vulnerabilities. The paper aims to demystify the development and security implications observed in real-world scenarios.

B. Formal Analysis Tools

In the panorama of prior academic investigations, a range of sophisticated tools, including but not limited to Proverif [21], [33], Tamarin [26], [34], [35], Deepsec [36], and others [37], [38], has been effectively employed. [23] introduces a formal model of the Bluetooth protocol suite using ProVerif in order to attain automated Bluetooth security analysis with formal assurances. This model encompasses both the key sharing and data transmission phases across all three Bluetooth protocols, facilitating the automatic detection of potential design flaws. [36] introduces the DEEPSEC verification tool, providing new complexity results for static equivalence, trace equivalence, and marked similarity. The paper also offers a decision-making procedure for these equivalences in scenarios with a limited number of sessions. Furthermore, in Papers [26] and [35], Tamarin Prover was utilized to conduct formal analysis on Signal's Sesame session management protocol. These papers present concrete instances where Post-Compromise Security (PCS) can be compromised during secure messaging, even with Signal's implementation of the double-ratchet protocol. Besides, [39] conducted a thorough formal analysis of the 12th version of the EDHOC draft. The research employed the SAPIC+ protocol platform, which integrates PROVERIF, TAMARIN, and DEEPSEC, and various compromised scenarios were considered during the analysis, resulting in the modeling of weaknesses.

We opted for the ProVerif primarily due to our familiarity with its functionality, and also, it provides easily comprehensible output, aiding us in the analysis and comprehension of the protocol's nature and potential security risks.

VIII. Conclusions

In this paper, we undertake the inaugural automated formal verification of the security protocol within the decentralized communication network Matrix. The study pioneers an automated formal verification of Matrix security protocols, encompassing pivotal facets such as key distribution, key request, key backup, the SAS device verification protocol, and the intricate Olm-based cryptography ratchet protocol. As part of our methodological approach, we introduce adaptable threat models and a nuanced set of security properties tailored to diverse security protocols. This adaptability ensures a comprehensive analysis that can be flexibly applied across different aspects of the security framework. Building upon these considerations, we articulate and implement a modular modeling method, enhancing the versatility and scalability of our verification process. Through the utilization of our formal verification model, we systematically scrutinize various Matrix security protocols, aiming to identify and understand potential vulnerabilities. And our comprehensive analysis reveals instances of security threats, including but not limited to, the susceptibility to man-in-the-middle (MitM) attacks, the

risk of impersonation attacks, and the potential vulnerabilities associated with clone attack.

REFERENCES

[1] G. C. Schipper, R. Seelt, and N. Le-Khac, "Forensic analysis of Matrix protocol and riot.im application," *Digit. Investig.*, vol. 36 Supplement, p. 301118, 2021.

[2] G. Korpal and D. Scott, "Decentralization and web3 technologies," 2022. [Online]. Available: https://www.techrxiv.org/articles/preprint/Decentralization_and_web3_technologies/19727734

[3] Matrix.org, "An open network for secure, decentralized communication." https://matrix.org/, 2023.

[4] Element.io, "Customers," https://element.io/customers, 2024.

[5] ——, "France embraces Matrix to build Tchap," https://element.io/case-studies/tchap, 2023.

[6] Matrix.org, "Matrix Specification," https://spec.matrix.org/latest/, 2022.

[7] ——, "End-to-End Encryption implementation guide," https://matrix.org/docs/matrix-concepts/end-to-end-encryption/, 2024.

[8] ——, "Olm: A cryptographic ratchet," https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/olm.md, 2019.

[9] ——, "Megolm group ratchet," https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/megolm.md, 2022.

[10] ——, "Short Authentication String (SAS) verification," https://spec.matrix.org/v1.9/client-server-api/#short-authentication-string-sas-verification, 2024.

[11] ——, "Client-Server API," https://spec.matrix.org/v1.9/client-server-api/, 2024.

[12] ——, "Server-Server API," https://spec.matrix.org/v1.9/server-server-api/, 2024.

[13] ——, "Application-Server API," https://spec.matrix.org/v1.9/application-service-api/, 2024.

[14] T. Perrin and M. Marlinspike, "The Double Ratchet Algorithm," https://signal.org/docs/specifications/doubleratchet/, 2016.

[15] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "Ed25519: high-speed high-security signatures," https://ed25519.cr.yp.to/, Access in 2023.

[16] D. J. Bernstein, "A state-of-the-art Diffie-Hellman function," https://cr.yp.to/ecdh.html, Access in 2023.

[17] Matrix.org, "Key Distribution," https://spec.matrix.org/v1.9/client-server-api/#key-distribution, 2024.

[18] P. Zimmermann, "ZRTP: Media Path Key Agreement for Unicast Secure RTP," https://philzimmermann.com/docs/zrtp/ietf/rfc6189.html, 2011.

[19] M. R. Albrecht, S. Celi, B. Dowling, and D. Jones, "Practically-exploitable cryptographic vulnerabilities in matrix," Cryptology ePrint Archive, Paper 2023/485, 2023, https://eprint.iacr.org/2023/485. [Online]. Available: https://eprint.iacr.org/2023/485

[20] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[21] V. C. Bruno Blanchet, "ProVerif: Cryptographic protocol verifier in the formal model," https://bblanche.gitlabpages.inria.fr/proverif/, 2023.

[22] T.-Y. Wu, L. Wang, X. Guo, Y.-C. Chen, and S.-C. Chu, "Sakap: Sgx-based authentication key agreement protocol in iot-enabled cloud computing," *Sustainability*, vol. 14, no. 17, p. 11054, 2022.

[23] J. Wu, R. Wu, D. Xu, D. J. Tian, and A. Bianchi, "Formal model-driven discovery of bluetooth protocol design vulnerabilities," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 2285–2303. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833777

[24] S. S. Al-Riyami and K. G. Paterson, "Certificateless public key cryptography," in *International conference on the theory and application of cryptology and information security*. Springer, 2003, pp. 452–473.

[25] Matrix.org, "Key Requests," https://spec.matrix.org/v1.9/client-server-api/#key-requests, 2024.

[26] C. Cremers, J. Fairoze, B. Kiesl, and A. Naska, "Clone detection in secure messaging: Improving post-compromise security in practice," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM, 2020, pp. 1481–1495. [Online]. Available: https://doi.org/10.1145/3372297.3423354

[27] Element.io, "Get the Element app to communicate on your own terms." https://element.io/download, Access in 2023.

[28] Matrix.org, "Server-side key backups," https://spec.matrix.org/v1.9/client-server-api/#server-side-key-backups, Access in 2023.

[29] F. Jacob, C. Beer, N. Henze, and H. Hartenstein, "Analysis of the Matrix event graph replicated data type," *IEEE Access*, vol. 9, pp. 28 317–28 333, 2021.

[30] F. Jacob, J. Grashöfer, and H. Hartenstein, "A glimpse of the Matrix (extended version): Scalability issues of a new message-oriented data synchronization middleware," *CoRR*, vol. abs/1910.06295, 2019. [Online]. Available: http://arxiv.org/abs/1910.06295

[31] H. Li, Y. Wu, R. Huang, X. Mi, C. HU, and S. Guo, "Demystifying decentralized matrix communication network: Ecosystem and security," in *29th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2023, Haikou, China, December 17-21, 2023*. IEEE, 2023.

[32] Element.io, https://element.io/, 2023.

[33] B. Blanchet, V. Cheval, and V. Cortier, "Proverif with lemmas, induction, fast subsumption, and much more," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 69–86. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833653

[34] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer, 2013, pp. 696–701.

[35] C. Cremers, C. Jacomme, and A. Naska, "Formal analysis of Session-Handling in secure messaging: Lifting security from sessions to conversations," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1235–1252. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/cremers-session-handling

[36] V. Cheval, S. Kremer, and I. Rakotonirina, "DEEPSEC: deciding equivalence properties in security protocols theory and practice," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 529–546. [Online]. Available: https://doi.org/10.1109/SP.2018.00033

[37] B. Blanchet, "Cryptoverif: Computationally sound mechanized prover for cryptographic protocols," in *Dagstuhl seminar "Formal Protocol Verification Applied*, vol. 117, 2007, p. 156.

[38] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele, "Dy*: A modular symbolic verification framework for executable cryptographic protocol code," in *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 523–542. [Online]. Available: https://doi.org/10.1109/EuroSP51992.2021.00042

[39] C. Jacomme, E. Klein, S. Kremer, and M. Racouchot, "A comprehensive, formal and automated analysis of the EDHOC protocol," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5881–5898. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/jacomme

APPENDIX

A. Formal Models

B. Findings of the Formal Model

```
// Various actions of the key distribution
type action.
const Upload: action.
const Claim: action.
const Query: action.
// Define the correlation of individual events
event evUploadAliceKey(pkey).
event evClaimAliceKey(pkey).
event evQueryAliceKey(bitstring).
...
// Client Alice generates, uploads, claims and queries keys
let clientA(sk_A_I: skey) = (
    new sk_A_E: skey; let pk_A_I = pk(sk_A_I) in
    let pk_A_E = pk(sk_A_E) in
    let actionUpload = Upload in
    event evUploadAliceKey(pk_A_I);
    out(ch_c2s, (actionUpload, pk_A_I));
    let actionClaim = Claim in
    event evClaimAliceKey(pk_A_E);
    out(ch_c2s, (actionClaim, pk_A_E));
    let actionQuery = Query in
    event evQueryBobKey(userID_B);
    out(ch_c2s, (userID_B, actionQuery));
    in(ch_s2c, (userID_B:bitstring, pk_B_I: pkey, pk_B_E: pkey));
    event evRecieveBobKeys(userID_B, pk_B_I, pk_B_E);
...
).
// Alice's homeserver relays action
let serverA() = (
in(ch_c2s, (actionUpload: action, pk_A_I: pkey));
in(ch_c2s, (actionClaim: action, pk_A_E: pkey));
in(ch_c2s, (user: bitstring, actionQuery: action));
...
).
// Bob's homeserver relays action
let serverB() = (
in(ch_c2s, (actionUpload: action, pk_B_I: pkey));
in(ch_c2s, (actionClaim: action, pk_B_E: pkey));
in(ch_c2s, (user: bitstring, actionQuery: action));
...
).
// Client Alice generates, uploads, claims and queries keys
let clientB(sk_B_I: skey) = (
...
    let actionUpload = Upload in
    event evUploadBobKey(pk_B_I);
    out(ch_c2s, (actionUpload, pk_B_I));
    let actionClaim = Claim in
    event evClaimBobKey(pk_B_E);
    out(ch_c2s, (actionClaim, pk_B_E));
    let actionQuery = Query in
    event evQueryAliceKey(userID_A);
    out(ch_c2s, (userID_A, actionQuery));
...
).
// Correspondence events to verify protocol security properties.
query userID: bitstring, pk_I: pkey, pk_E: pkey;
event(evRecieveAliceKeys(userID, pk_I, pk_E)) ==> event(evUploadAliceKey(pk_I)) && event(evClaimAliceKey(pk_E)).
...
// Runs the process.
process ( !clientA(sk_A_I) | !serverA() | !serverB() | !clientB(sk_B_I) )
```

Listing 3. Implementation of formal modeling of Key distribution module.

```
// (* OLM Algorithm tables to transport data *)
table table1_client_alice(...).
table table1_client_bob(...).
// (* AES-CBC *)
fun senccbc(bitstring, key, iv): bitstring.
reduc forall m:bitstring, k: key, i: iv; sdeccbc(senccbc(m, k, i), k, i) = m.
// (* Client send sensitive data *)
free plain_text_A: bitstring [private].
// (* Keys exchange events *)
event evClientAliceCalcShareSecret(sharedkey).
...
// (* step 1 ECDH key exchange in client Alice *)
let step1clientA(skey_A_I:skey) =
( ...
// Client Alice calculate shared key
    let SharedSecret_AB = concat3(...) in
    event evClientAliceCalcShareSecret(bitstring2sharedkey(SharedSecret_AB));
    insert table1_client_alice(...) ).
// step 1 ECDH key exchange in homeserver Alice
let step1serverA() = ( ... ).
// step 1 ECDH key exchange in homeserver Bob
let step1serverB() = ( ... ).
// step 1 ECDH key exchange in client Bob
let step1clientB(skey_B_I:skey) =
( // Client Bob receive Alice's key and calculate shared key
...
    event evClientBobCalcShareSecret(bitstring2sharedkey(SharedSecret_BA));
    insert table1_client_bob(...))
).
// step 2 client Alice encrypt message.
let step2clientA() =
(   get table1_client_alice(...) in
// generate root key, chain key, and ratchet key, and messgae key.
    let (rootKey: key, chainKey: key) = h1(SALT, SharedSecret_AB, INFO, 64) in
    let ratchetKey_A = rk(skey_A_I) in
    let msgKey = HMAC_SHA256(...) in
// encrypt plain-text.
    let (aesKey: key, hmacKey: key, aesIV: iv) = h2(SALT, msgKey, INFO, 80) in
    let cipher_text_A = senccbc(plain_text_A, aesKey, aesIV) in
... ).
// step 2 server Alice receive and send.)
let step2serverA() = ( ... ).
// step 2 server Bob receive and send. *)
let step2serverB() = ( ... ).
// (* step 2 client Bob decrypt message. *)
let step2clientB() =
(   get table1_client_bob(...) in
// (* generate root key, chain key, and ratchet key, and messgae key *)
...
    let plain_text_A = sdeccbc(cipher_text_A, aesKey, aesIV) in
... ).
// Authenticity and Confidentiality properties
query ...; event(evClientAliceReceiveCipertext(ciper_text)) ==> event(evClientBobSendCipertext(ciper_text)).
query attacker(plain_text).
...
// Run the process.
process(!step1clientA|!step1serverA|!step1serverB|!step1clientB|!step2clientA|!step2serverA|!step2serverB|!step2clientB)
```

Listing 4. Formal modeling of the Olm cryptographic ratchet.

```
// channel between server and cloner
free ch_clone2s: channel.
free RoomKeyRequest: action.
free RoomForwaredKey: action.
// session key between Alice and Bob
free sessionKey_AB: tp_sessionKey [private].
// insert clientA's infos to table
table table_clientA_cloner(...).
event evClonerGetRoomForwardedKey(tp_sessionKey).
event evClonerCanDecryptFormerMessage(bitstring).
event evClientBobBuildSessionWithCloner(...).
...
// here we simply define sessionkey
fun get_session_key_between_AB(...): sessionKey.
// sensitive data of Alice
free plain_text_A: bitstring [private].
let clientA() =
( ...
    insert table_clientA_cloner(...);
    in(ch_s2c, (deviceID_A_clone: bitstring,
    request_action: action));
// Alice forward room session keys
    let forward_action = Forwarded_Key in
    out(ch_c2s, (forward_action, sessionKey_AB));
... ).
// Cloner of Alice.
let clientA_cloner() =
(   get table_clientA_cloner(...) in
// Send m.room_key_request to deviceID_A to obtain
    the session key and decrypt it
    out(ch_clone2s, (deviceID_A_clone,
    request_action));
    in(ch_s2clone, (forward_action: action,
    sessionKey_AB: tp_sessionKey));
    event evClonerGetRoomForwardedKey(sessionKey_AB
    );
    let aesKey_clone, hmacKey_clone, aesIV_clone =
    get_aes_key_between_AB(...) in
    let plain_text_A = sdeccbc(cipher_text_A,
    aesKey_clone, aesIV_clone) in
    event evClonerCanDecryptFormerMessage(...);
// Cloner can receive and decrypt.
...
    event evClonerCanDecryptBobMessage(...);
... ).
// Alice's server
let serverA() =
( ... ).
// Bob's server
let serverB() =
( ... ).
// client Bob
let clientB() =
( // Alice Cloner can build session with Bob
...
    let sessionKey_BA_clone_new =
    get_session_key_between_AB(...) in
    event evClientBobBuildSessionWithCloner(...);
... ).
// (* authenticity properties *)
query ...; event(evClonerCanDecryptFormerMessage(
    plain_text)) ==>
( event(evClonerGetRoomForwardedKey(sessionKey))
    ==> event(evCloneTable(...)) ).
query ...;
event(evClientBobBuildSessionWithCloner(sharedKey))
     ==> event(evCloneTable(userID, deviceID,
    access_token, skey_I, skey_E)).
...
process (!clientA|!serverA|!serverB|!clientB|!
    clientA_cloner)
```

Listing 5. Formal modeling of the key requests in Compulsory Access.

```
type sessionkey.
// algorithm of key backup
const BackupAlgorithm: bitstring.  (* m.
    megolm_backup.v1.curve25519-aes-sha2 *)
// Sensitive data
free roomSessionKey: sessionkey [private].
free key_backup_data: object [private].
// Alice device 1 can generate backup keys data
fun gen_key_backup_data(number, number,
    verification, object): object.
...
// Alice's homeserver receive keys
event evDevice2GetRoomKeysVersion(action,
    access_token).
event evServerAliceReceiveRoomKeysVersion(action,
    access_token).
//
let aliceDevice1(sk_A_I:skey, pk_B_I:pkey, pk_B_E:
    pkey) = (
// POST /_matrix/client/v3/room_keys/version
...
    let auth_data = get_authdata(...)) in
    out(ch_c2s, (POST, actionRoomKeysVersion,
    BackupAlgorithm, auth_data));
...
// (* PUT /_matrix/client/v3/room_keys/keys *)
...
    let session_data = encrypt_session_keys(sk_A_I,
     roomSessionKey) in
    let key_backup_data = gen_key_backup_data(...,
    session_data) in
    out(ch_c2s, (PUT, (RoomKeysKeys, version),
    sessionID, key_backup_data)); ).
// Alice's homeserver receives backup keys.
let serverA() = (
    in(ch_c2s, (GET:urlrequesttype,
    actionRoomKeysVersion:action, authentication:
    access_token));
    event evServerAliceReceiveRoomKeysVersion(
    actionRoomKeysVersion, authentication); ).
// Another device of Alice
let aliceDevice2(sk_A_I:skey) = ( ...
// (* GET /_matrix/client/v3/room_keys/version *)
    event evDevice2GetRoomKeysVersion(
    actionRoomKeysVersion, authentication);
    out(ch_c2s, (GET, actionRoomKeysVersion,
    authentication));
    in(ch_s2c, (BackupAlgorithm:bitstring,
    auth_data:object, count:number, etag:bitstring
    , version:bitstring));
// (* GET /_matrix/client/v3/room_keys/keys *)
    let actionRoomKeysKeys = RoomKeysKeys in
    out(ch_c2s, (GET, (actionRoomKeysKeys, version)
    ));
    in(ch_s2c, (sessionID:bitstring,
    key_backup_data:object));
    let roomSessionKey = decrypt_session_keys(
    sk_A_I, key_backup_data) in
).
// (* authenticity properties *)
query ...;
event(evServerAliceReceiveRoomKeysVersion(
    actionRoomKeysVersion, authentication)) ==>
event(evDevice2GetRoomKeysVersion(
    actionRoomKeysVersion, authentication)).
// (* Confidentiality properties *)
query attacker(roomSessionKey).
query attacker(key_backup_data).
process (!aliceDevice1|!serverA|!aliceDevice2)
```

Listing 6. Formal modeling implementation of the key backups protocol.

```
free RoomKeyRequest: event_type.
free RoomForwaredKey: event_type.
free sessionKey: sesskey [private].
// request events
event evDeviceAReceiveKey(sesskey).
// Alice's device A
let deviceA() =
(
    let action = RoomKeyRequest in
    out(ch, (action, deviceID_A, deviceID_AA));
    in(ch, (action:event_type, from_device:
    bitstring, to_device:bitstring));
    in(ch, sessionKey:sesskey);
    event evDeviceAReceiveKey(sessionKey);
...
).
// Alice's server recevie and send msg
let serverA() =
( ... ).
// Alice's device AA
let deviceAA() =
(
    in(ch, (action:event_type, from_device:
    bitstring, to_device:bitstring));
    if action = RoomKeyRequest && to_device =
    self_deviceID then
    let (CV:verification, deviceID_A:bitstring) =
    check_verification(from_device) in
    let action = RoomForwaredKey in
    out(ch, (action, deviceID_AA, deviceID_A));
    out(ch, sessionKey);
    event evDeviceAASendKey(sessionKey);
...
).
// define Authenticity properties and
    Confidentiality properties
query session_key:sesskey; event(
    evDeviceAReceiveKey(session_key)) ==> event(
    evDeviceAASendKey(session_key)).
query attacker(sessionKey).
process ( !deviceA|!serverA|!deviceAA )
```

Listing 7. Formal modeling implementation of the key requests protocol in External Security Compromised.

```
new sk_A_I: skey creating sk_A_I_2 at {1}
new sk_B_I: skey creating sk_B_I_2 at {2}
new sk_B_E: skey creating sk_B_E_1 at {43} in
    copy a
event evUploadBobKey(pk(sk_B_I_2)) at {47} in
    copy a
out(ch_c2s, (~M,~M_1)) with ~M = Upload, ~M_1 =
    pk(sk_B_I_2) at {48} in copy a
event evClaimBobKey(pk(sk_B_E_1)) at {50} in
    copy a
out(ch_c2s, (~M_2,~M_3)) with ~M_2 = Claim, ~M_3
     = pk(sk_B_E_1) at {51} in copy a
event evQueryAliceKey(userID_A) at {53} in copy
    a
out(ch_c2s, (~M_4,~M_5)) with ~M_4 = userID_A, ~
    M_5 = Query at {54} in copy a
in(ch_s2c, (a_1,a_2,a_3)) at {55} in copy a
event evRecieveAliceKeys(a_1,a_2,a_3) at {56} in
     copy a (goal)
The event evRecieveAliceKeys(a_1,a_2,a_3) is
    executed at {56} in copy a.
A trace has been found.
RESULT event(evRecieveAliceKeys(userID,pk_I,pk_E
    )) ==> event(evUploadAliceKey(pk_I)) &&
    event(evClaimAliceKey(pk_E)) is false.
--------------------------------
new sk_A_I: skey creating sk_A_I_2 at {1}
new sk_B_I: skey creating sk_B_I_2 at {2}
new sk_A_E: skey creating sk_A_E_1 at {5} in
    copy a
event evUploadAliceKey(pk(sk_A_I_2)) at {9} in
    copy a
out(ch_c2s, (~M,~M_1)) with ~M = Upload, ~M_1 =
    pk(sk_A_I_2) at {10} in copy a
event evClaimAliceKey(pk(sk_A_E_1)) at {12} in
    copy a
out(ch_c2s, (~M_2,~M_3)) with ~M_2 = Claim, ~M_3
     = pk(sk_A_E_1) at {13} in copy a
event evQueryBobKey(userID_B) at {15} in copy a
out(ch_c2s, (~M_4,~M_5)) with ~M_4 = userID_B, ~
    M_5 = Query at {16} in copy a
in(ch_s2c, (a_1,a_2,a_3)) at {17} in copy a
event evRecieveBobKeys(a_1,a_2,a_3) at {18} in
    copy a (goal)
The event evRecieveBobKeys(a_1,a_2,a_3) is
    executed at {18} in copy a.
A trace has been found.
RESULT event(evRecieveBobKeys(userID,pk_I,pk_E))
     ==> event(evUploadBobKey(pk_I)) && event(
    evClaimBobKey(pk_E)) is false.
----------------------------------------------
Verification summary:
Query event(evRecieveAliceKeys(userID,pk_I,pk_E)
    ) ==> event(evUploadAliceKey(pk_I)) && event
    (evClaimAliceKey(pk_E)) is false.
Query event(evRecieveBobKeys(userID,pk_I,pk_E))
    ==> event(evUploadBobKey(pk_I)) && event(
    evClaimBobKey(pk_E)) is false.
----------------------------------------------
```

Listing 8. MitM Attack trace against the key distribution protocol.

```
new sk_A_I: skey creating sk_A_I_3 at {1}
new sk_B_I: skey creating sk_B_I_1 at {2}
new sk_B_E: skey creating sk_B_E_1 at {3}
in(ch_c2s, (a,a_1,a_2,a_3)) at {27} in copy a_4
out(ch_s2c, ~M) with ~M = Version at {30} in
    copy a_4
in(ch_c2s, (a_5,(a_6,a_7),a_8,a_9)) at {31} in
    copy a_4
new etag_1: bitstring creating etag_3 at {32} in
     copy a_4
out(ch_s2c, (~M_1,~M_2)) with ~M_1 = get_count(
    a_3), ~M_2 = etag_3 at {33} in copy a_4
in(ch_c2s, (a_10,a_11,a_12)) at {34} in copy a_4
event evServerAliceReceiveRoomKeysVersion(a_11,
    a_12) at {35} in copy a_4 (goal)
The event evServerAliceReceiveRoomKeysVersion(
    a_11,a_12) is executed at {35} in copy a_4.
A trace has been found.
RESULT event(evServerAliceReceiveRoomKeysVersion
    (...)) ==> event(evDevice2GetRoomKeysVersion
    (...)) is false.
--------------------------------------------
Verification summary:
Query event(evServerAliceReceiveRoomKeysVersion(
    actionRoomKeysVersion_4,authentication_2))
    ==> event(evDevice2GetRoomKeysVersion(
    actionRoomKeysVersion_4,authentication_2))
    is false.
--------------------------------------------
```

Listing 9. Attack trace of the MitM attack against the key backups protocol.

```
event evDeviceASendRequest(deviceID_AA) at {4}
    in copy a
out(ch, (~M,~M_1,~M_2)) with ~M=RoomKeyRequest,
    ~M_1 = deviceID_A, ~M_2 = deviceID_AA at {5}
     in copy a
in(ch, (a_1,a_2,a_3)) at {6} in copy a
in(ch, a_4) at {7} in copy a
event evDeviceAReceiveKey(a_4) at {8} in copy a
    (goal)
The event evDeviceAReceiveKey(a_4) is executed
    at {8} in copy a.
A trace has been found.
RESULT event(evDeviceAReceiveKey(session_key))
    ==> event(evDeviceAASendKey(session_key)) is
     false.
--------------------------------------------
in(ch, (RoomKeyRequest,a,deviceID_AA)) at {25}
    in copy a_1
event evDeviceAAReceiveRequest(a) at {26} in
    copy a_1
out(ch, (~M,~M_1,~M_2)) with ~M =
    RoomForwaredKey, ~M_1 = deviceID_AA, ~M_2 =
    deviceID_A at {31} in copy a_1
out(ch, ~M_3) with ~M_3 = sessionKey at {32} in
    copy a_1
event evDeviceAASendKey(sessionKey) at {33} in
    copy a_1
The attacker has the message ~M_3 = sessionKey.
A trace has been found.
RESULT not attacker(sessionKey[]) is false.
--------------------------------------------
Verification summary:
Query event(evDeviceAReceiveKey(session_key))
    ==> event(evDeviceAASendKey(session_key)) is
     false.
Query not attacker(sessionKey[]) is false.
--------------------------------------------
```

Listing 10. Attack trace of the MitM attack against the key requests.

```
new skey_A_I: skey creating skey_A_I_3 at {1}
new skey_B_I: skey creating skey_B_I_3 at {2}
new skey_A_E: skey creating skey_A_E_3 at {5} in
     copy a
event evClientAliceSendKeys(pk(skey_A_I_3),pk(
    skey_A_E_3)) at {8} in copy a
out(ch_c2s, (~M,~M_1)) with ~M = pk(skey_A_I_3),
     ~M_1 = pk(skey_A_E_3) at {9} in copy a
in(ch_s2c, (a_1,a_2)) at {10} in copy a
event evClientAliceReceiveKeys(a_1,a_2) at {11}
    in copy a
event evClientAliceCalcShareSecret(concat3(sca(
    skey_A_I_3,a_2),sca(skey_A_E_3,a_1),sca(
    skey_A_E_3,a_2))) at {18} in copy a
insert table1_client_alice(skey_A_I_3,skey_A_E_3
    ,pk(skey_A_I_3),pk(skey_A_E_3),a_1,a_2,
    concat3(sca(skey_A_I_3,a_2),sca(skey_A_E_3,
    a_1),sca(skey_A_E_3,a_2))) at {19} in copy a
get table1_client_alice(skey_A_I_3,skey_A_E_3,pk
    (skey_A_I_3),pk(skey_A_E_3),a_1,a_2,concat3(
    sca(skey_A_I_3,a_2),sca(skey_A_E_3,a_1),sca(
    skey_A_E_3,a_2))) at {83} in copy a_3
event evClientAliceSendRatchetKey(pk(rk(
    skey_A_I_3))) at {61} in copy a_3
out(ch_c2s, ~M_2) with ~M_2 = pk(rk(skey_A_I_3))
     at {62} in copy a_3
event evClientAliceSendCipertext(senccbc(
    plain_text_A,fist256bit(HKDF(SALT,
    HMAC_SHA256(next256bit(HKDF(SALT,concat3(sca
    (skey_A_I_3,a_2),sca(skey_A_E_3,a_1),sca(
    skey_A_E_3,a_2)),INFO,64)),PADDING_1),INFO
    ,80)),last128bit(HKDF(SALT,HMAC_SHA256(
    next256bit(HKDF(SALT,concat3(sca(skey_A_I_3,
    a_2),sca(skey_A_E_3,a_1),sca(skey_A_E_3,a_2)
    ),INFO,64)),PADDING_1),INFO,80)))) at {63}
    in copy a_3
out(ch_c2s, ~M_3) with ~M_3 = senccbc(
    plain_text_A,fist256bit(HKDF(SALT,
    HMAC_SHA256(next256bit(HKDF(SALT,concat3(sca
    (skey_A_I_3,a_2),sca(skey_A_E_3,a_1),sca(
    skey_A_E_3,a_2)),INFO,64)),PADDING_1),INFO
    ,80)),last128bit(HKDF(SALT,HMAC_SHA256(
    next256bit(HKDF(SALT,concat3(sca(skey_A_I_3,
    a_2),sca(skey_A_E_3,a_1),sca(skey_A_E_3,a_2)
    ),INFO,64)),PADDING_1),INFO,80))) at {64} in
     copy a_3
in(ch_s2c, a_4) at {65} in copy a_3
event evClientAliceReceiveRatchetKey(a_4) at
    {66} in copy a_3
in(ch_s2c, a_5) at {67} in copy a_3
event evClientAliceReceiveCipertext(a_5) at {68}
     in copy a_3 (goal)
The event evClientAliceReceiveCipertext(a_5) is
    executed at {68} in copy a_3.
A trace has been found.
RESULT event(evClientAliceReceiveCipertext(
    ciper_text)) ==> event(
    evClientBobSendCipertext(ciper_text)) is
    false.
--------------------------------------------
Verification summary:
Query event(evClientAliceReceiveCipertext(
    ciper_text)) ==> event(
    evClientBobSendCipertext(ciper_text)) is
    false.
--------------------------------------------
```

Listing 11. Attack trace of the impersonation attack against the Olm-based cryptographic ratchet protocol.