RADICALLY OPEN SECURITY

Cryptographic Analysis Report

Tox Instant Messenger

V 1.1
Amsterdam, June 5th, 2024

Document Properties

Client	Tox Instant Messenger
Title	Cryptographic Analysis Report
Target	The Noise-based handshake for the Tox instant messaging protocol.
Version	1.1
Pentester	Younes Talibi Alaoui
Authors	Younes Talibi Alaoui, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	May 1st, 2024	Younes Talibi Alaoui	Initial draft
0.2	May 5th, 2024	Marcus Bointon	Review
0.3	May 14th, 2024	Younes Talibi Alaoui	Minor edits
1.0	June 4th, 2024	Marcus Bointon	1.0
1.1	June 5th, 2024	Younes Talibi Alaoui	Minor edits

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Tox is a peer-to-peer instant-messaging protocol that aims to provide secure messaging without relying on centralised servers. The Tox project started in 2013. In 2017, Jason A. Donenfeld pointed out that Tox is vulnerable to a key compromise impersonation attack [1] due to a flaw in the handshake protocol for establishing a shared secret keys between the peers (see [2] for details).

In order to overcome this, Tobias Buchberger initiated the development of a new hanshake protocol for Tox, beginning with his master's thesis [2]. This handshake is based on Noise [4], a widely used framework for building secure channel protocols such as WhatsApp and WireGuard.

In this report, we provide a security analysis of the new Noise-based handshake for Tox [5], targeting the latest version available at the time of the start of the analysis[1].

1.2 Scope of work

The scope of the analysis was limited to checking the new Noise-based handshake for the Tox protocol, and *informally* to discuss the security guarantees provided with this new handshake. The analysis also included an inspection of the implementation's code.

Note that the analysis was constrained by a time limit of *10 person-days* (including reporting).

[1] commit `4f16bb2d0ace0bd890b4a7553b3986b742625297`

2 Preliminaries

2.1 Terminology

Throughout this report we will be referring to some cryptographic primitives often:

1. Key pair: A Diffie-Hellman key pair consisting of a private key and its corresponding public key.
2. `DH(private_key, public_key)`: Performs a Diffie-Hellman exchange using a `private_key` and a `public_key`, to derive a shared secret common to the party holding `private_key`, and the party holding the private key corresponding to `public_key`.
3. `HKDF(ck,ik,nb)` : A Key Derivation Function (KDF) based on hash functions, that takes as input two keys `ck` and `ik`, and a parameter `nb`, and returns `nb` number of keys.
4. `ENC(k; n; p; ad)`: Performs Authenticated Encryption with Associated Data (AEAD). That is, it takes plaintext `p`, and additional data `ad`, and encrypts it with key `k` and nonce `n` to generate an authenticated ciphertext. The field that takes plaintext `p`, can also accept a vector of plaintexts.
5. `DEC(k; n; c; ad)`: Performs Authenticated Decryption over the ciphertext `c`, using key `k`, nonce `n`, and additional data `ad`.
6. `H`: A hash function that takes data of arbitrary length, and returns an output of `HASHLEN` bytes.

We also refer to the empty string by `\epsilon`, this is used for instance in `ENC` in the field that takes additional data, when no additional data is associated to the plaintext being encrypted.

2.2 Tox

Here we provide a high-level description of the Tox protocol, targeting the parts that are relevant for the new handshake. For more details about the Tox protocol, please refer to the corresponding documentation [3].

Within the Tox network, peers generate session keys using ephemeral keys, which are communicated in an encrypted form through their respective long-term key pairs. The handshake for deriving session keys incorporates a cookie mechanism. That is, a Peer A in Tox holds the following keys:

1. `(S_A^pub, S_A^priv)`: A long term static pair of keys.
2. `(DHT_A^pub, DHT_A^priv)`: A pair of keys used to derive the key used to encrypt the cookie request packets. This pair is renewed every time the Tox instance is closed.
3. `K_A^C`: The key used to encrypt the cookies generated by Peer A.

Peer A also generates an ephemeral key pair `(E_A^pub, E_A^priv)` for each session with Peer B.

The cookie created by Peer B for Peer A is of the form : `C_B = N_B^C||ENC(K_B^C; N_B^C;t_B, S_A^pub, DHT_A^pub; \epsilon)` where `N_B^C` is a random nonce created by Peer B, and `t_B` is a timestamp marking the time of the creation of the cookie.

The Tox handshake between Peers A and B starts with a cookie request, where Peer A sends a packet of the form: `DHT_A^pub, N_A^R`, and the vector `S_A^pub, pad, echoID`, encrypted with key `K_AB^R = DH(DHT_A^priv,DHT_B^pub)` and nonce `N_A^R`. `pad` is a string consisting of the concatenation of 32 bytes of zeros, and `echoID` is a random number used to check the validity of the cookie response packets.

Peer B responds with a cookie response packet of the form: `N_B^R`, and the vector `C_B, echoID`, encrypted with the key `K_AB^R = DH(DHT_B^priv,DHT_A^pub)` and nonce `N_B^R`. Peer A then creates a handshake packet that includes `C_B` and sends it to Peer B. Peer A also includes `C_A` in the packet, so that Peer B can include it in its response (if `C_A` is not sent here, Peer B would have to send a cookie request packet to Peer A).

The two peers proceed with the handshake, where the ephemeral public key of each peer is communicated to the other peer, encrypted with the shared key generated from applying Diffie-Hellman over the long term static key pairs. The handshake is terminated if either peer finds the cookie invalid upon verification, or a decryption fails. The cookie check consists of verifying whether the Timestamp t exceeded the time limit (15 seconds), and verifying whether the static public key included in the cookie belongs to a friend of the peer receiving the cookie (for more details about this, refer to [3]). If the handshake completes, the shared key derived by the peers is `K_AB^data = DH(E_A^priv, E_B^pub) = DH(E_B^priv, E_A^pub)`. As was noted in [2], this handshake is vulnerable to a Key Compromise Impersonation (KCI) attack. That is, if such an attack succeeds on Peer A, the attacker will be able to impersonate any other Peer B to A. For more details about the attack, refer to [2].

2.3 Noise handshake

Noise [4] is a framework designed by Trevor Perrin, for establishing secure channel protocols. In Noise, we distinguish two parties, an *Initiator* and a *Responder*. Each of these parties has a long-term static key pair and/or an ephemeral key pair. At the end of the handshake, these keys are used to derive shared session keys that can be used to encrypt transport messages. The core of the handshake involves Diffie-Hellman and the usage of a `HKDF` function.

The framework provides a set of *handshake patterns* that one can choose while designing an application based on the Noise framework. A handshake pattern is composed of message patterns, which consist of a sequence of *tokens* designated with a direction (Initiator to Responder or Responder to Initiator). These tokens determine which Diffie-Hellman operations the parties perform, and which messages are exchanged. Messages consist of Diffie-Hellman public keys, and *payloads*, whose content is application-specific.

The possible tokens in the Noise framework are:

1. `"e"` and `"s"`: Indicate transmitting the ephemeral public key (for `"e"`) or the static public key (for `"s"`), in accordance with the token's direction, from the sender to the recipient.

2. `"ee"`, `"es"`, `"se"`, and `"ss"`: Indicate performing a Diffie-Hellman operation, between either the Initiator static or ephemeral key (determined by the first letter of the token), and the Responder static or ephemeral key (determined by the second letter of the token).

The instantiation of the cryptographic primitives involved in the exchange is application specific. This includes the Diffie-Hellman curve, the encryption scheme used to encrypt messages, and the hash function.

Each handshake pattern ensures some security properties for the handshake messages and the transport payloads. These properties are defined in the Noise documentation as a source property, related to authentication of the sender provided to the recipient, and a confidentiality property, related to the confidentiality provided to the sender. These properties, as defined in Noise, can be determined from the tokens involved. Each handshake pattern also ensures some level of identity hiding for the Initiator and Responder.

Noise handshake patterns have been formally analyzed [6], [7], [8], where more thorough analysis of the security properties ensured by Noise has been conducted. Protocols using a handshake *based* on the Noise handshake patterns have also been formally analyzed, such as the WireGuard protocol [9], [10], [11].

The NoiseIK pattern

$$
\begin{array}{ll}
\textbf{IK:} & \\
\leftarrow & s \\
\ldots & \\
\rightarrow & e,\ es,\ s,\ ss \\
\leftarrow & e,\ ee,\ se
\end{array}
$$

Figure 1: The NoiseIK pattern

In the NoiseIK pattern (Figure 1), both the Initiator and the Responder possess static `(S_A^pub, S_A^priv)` and ephemeral `(E_A^pub, E_A^priv)` key pairs. This pattern assumes that the Responder's static key has already been sent out-of-band to the Initiator, thus the pattern initiates with a pre-message through the `"s"` token. The first message from the Initiator to the Responder involves four tokens: `"e"`, `"es"`, `"s"`, and `"ss"`. Following this, the Responder sends a message involving three tokens: `"e"`, `"ee"`, and `"se"`. Subsequently, the parties can derive the session keys. A detailed description of this pattern is given in Figure 2 The lines in blue refer to the computation done by both parties (expressed through how the Initiator does it; the Responder does the same computation using the keys it has access to).

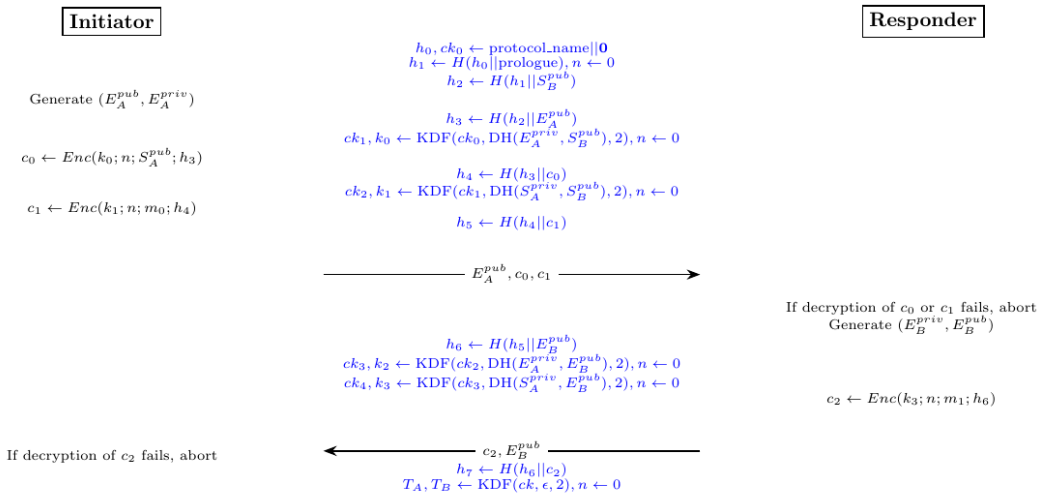

Figure 2: The NoiseIK description

The fields `protocol_name` and `prologue` are application-specific. `protocol_name` refers to the name of the NoiseIK instantiation, which is the concatenation of Noise_IK with the names of the Diffie-Hellman curve, the encryption scheme, and hash function chosen. The concatenations are separated with an underscore (see next section for an example).

The prologue is a field that can contain arbitrary data agreed upon by the Initiator and Responder, and `h_0` is either the concatenation of the `protocol_name` with zeros until the size reaches `HASHLEN` bytes (as described in the Figure 2), or `h_0` is the hash of `protocol_name`, if the size of this is larger than `HASHLEN` bytes.

Throughout the handshake, if data received from the other party fails to decrypt, the party failing to decrypt aborts the protocol. At the end of the handshake, the session keys derived are `T_A, T_B` (one key for encrypting/decrypting transport messages of the Initiator, and the other for encrypting/decrypting transport messages of the Responder).

For more details about the NoiseIK pattern, particularly its security guarantees, refer to [6], [7], [8].

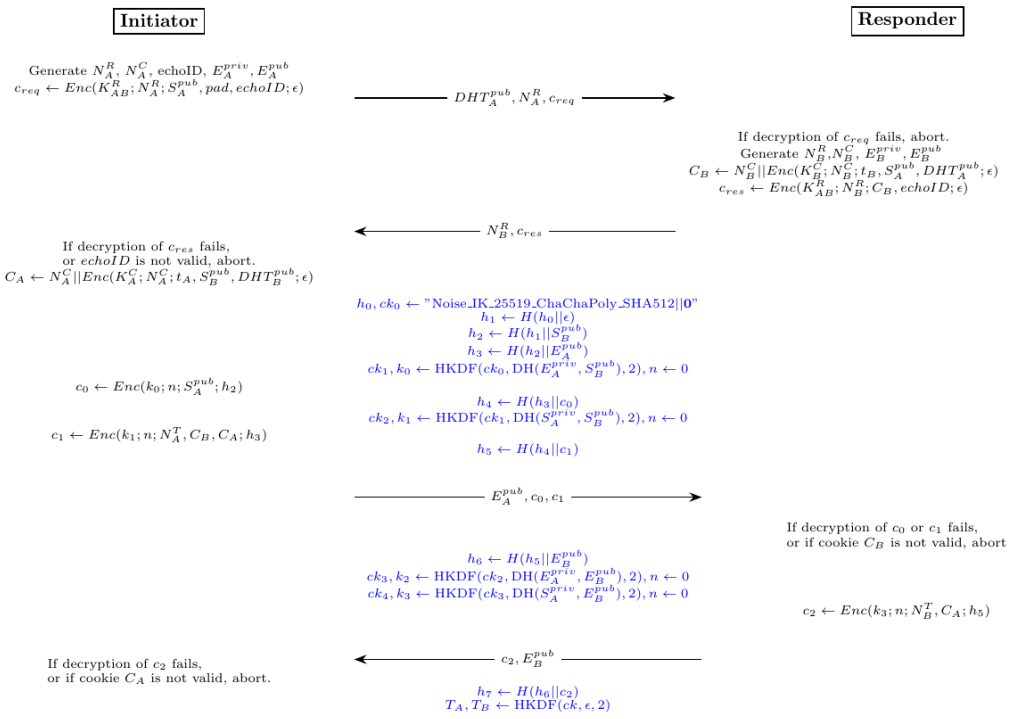

Figure 3: The NoiseIK based Tox handshake

2.4 Noise-based handshake for Tox

As mentioned earlier, the new handshake for Tox is based on NoiseIK. The full description of the handshake is given in Figure 3[2].

As to the cryptographic primitives chosen, the Diffie-Hellman curve chosen was Curve25519. Two encryption schemes were chosen, one for the handshake messages, ChaChaPoly, and one for the transport messages, XChaChaPoly. [3]. The hash function chosen was SHA512, which has a `HASHLEN` of 32 bytes.

The new handshake keeps the same cookie mechanism described in Section 2.2 of the old handshake, followed by the NoiseIK handshake. For transport messages, the Initiator (resp. the Responder) sets the nonce of a message to be the nonce of the previous message that they sent, incremented by one (resp. they sent), starting from a base nonce generated randomly by the Initiator (resp. the Responder)[4].

The prologue is fixed to be the empty string, and `h_0` is fixed as the protocol name, appended with zero bytes, until the size of `h_0` becomes that of `HASHLEN` bytes.

[2]Note that the protocol described in Figure 3 corresponds to commit `87f9c10942353bf108af483207c8b9cba60f90b`, which contained some modifications over the original commit on which we conducted the analysis; more details about these modifications are given in the report.

[3]Note that the usage of two encryption schemes happened as of commit `d87f9c10942353bf108af483207c8b9cba60f90b`, while the original commit contained the usage of only one encryption scheme.

[4]For details about transport messages in Tox, refer to the documentation [3]. Analyzing transport messages was out of scope of this report.

As for the payloads of the handshake messages, the one of the Initiator contains `C_B` created for it by the Responder, and the `C_A` it created for the Responder. The payload also contains the base nonce chosen by the Initiator for the encryption of the transport data (`N_A^T`). The payload of the Responder contains `C_A` that was created by the Initiator for the Responder, and the base nonce chosen by the Responder for the encryption of transport data (`N_B^T`).

Also note that the ephemeral keys are generated before the cookie request is sent. However, there is a case where the Responder generates the ephemeral keys after receiving the first packet of the Noise handshake from the Initiator. For details about this, refer to [3].

3 Security analysis

3.1 The Noise exchange

The protocol described in Figure 3 corresponds to a Noise handshake. That is, the steps from calculating `h_0` to the session keys `T_A` and `T_b`, corresponding to the IK pattern of the Noise handshake. However, the Noise-based handshake of Tox uses an additional encryption scheme for transport messages, XChaChaPoly, which has nonces of 192bits, in contrast to the Noise specifications of using one encryption scheme for both the handshake messages and transport data, with the size of nonces being 64bits.

The Noise documentation mentions some encryption schemes that can be used which have 96-bit nonces, but specifies that the first 32 bits of the nonces are encoded as zeros, and only the remaining 64 bits can differ between nonces.

In Tox's Noise-based handshake, the whole 192 bits is used to encode the nonces. This should not affect Noise's security guarantees. For more details about the design rationales behind fixing the size of nonce in Noise to be 64 bits, refer to section 15.1 in the Noise documentation.

Another departure from the Noise specifications is that the nonces of transport messages are incremented starting from the base nonces chosen by the Initiator and Responder, as opposed to the Noise specification where these nonces start from 0. This difference should also not affect Noise's security guarantees.

Discussion of the content of payload data in the handshake and the cookie mechanism, and how these may affect Noise's security guarantees appears in the following sections.

3.2 A Noise-based handshake

While the bulk of the new handshake is the NoiseIK as described in the previous section, the exchange related to the cookie mechanism should also be treated as part of the handshake. That is, the cookie mechanism cannot be treated separately from the handshake, as it is a mandatory part of the handshake. Specifically, the Initiator and the Responder cannot have a successful handshake if they do not include cookies that were generated in the cookie exchange in the handshake payloads that precede the Noise exchange.

This does have implications for the Noise handshake's guarantees, as we will see in the next sections, and in fact, any formal analysis of the handshake has to include the cookie exchange in the analysis, as well as all the keys that were introduced in the handshake due to Tox, namely the DHT keys, `K_AB^R`, `K_A^C`, and `K_B^C`. For instance, the new handshake sends the static key `S_A`, encrypted with the key `K_AB^R`, which is generated as a raw Diffie-Hellman key between the DHT keys of peers (it is worth mentioning that the DHT keys of a peer are renewed after every logout).

Some of the formal analysis conducted on the Noise handshake used automated tools, such as ProVerif [7] , CryptoVerif [11], and Tamarin [8], [9], where roughly speaking, the tool models the protocol to be analyzed as well as its security guarantees, then obtains results over the security guarantees. Now that the handshake consists of a modified flow of data, involving not only the Noise related keys, but also new keys that were added to the handshake, the modeling of the handshake has to be adjusted to accommodate these changes in order to analyze the new handshake formally.

This was the case, for instance, with WireGuard, which uses a Noise-based handshake. Specifically, WireGuard adds two Message Authentication Codes (MACs) and a cookie mechanism to the Noise handshake. Due to these changes in the Noise handshake, WireGuard has also been formally analyzed.

3.3 Clashes with NoiseIK due to the cookie mechanism

As explained earlier, the cookie mechanism adopted in the Noise-based handshake is the same as the one used in the old handshake, where the cookie exchange involves the Noise static keys. Besides that, these static keys are also included in the handshake's payloads.

This resulted in, for instance, that `S_A` is sent four times in total, encrypted (in some cases) with keys that are not related to the Noise handshake. That is, `S_A` is encrypted (1) with `K_AB^R` which is a key that is derived from the DHT keys, in the exchange that precedes the Noise handshake, (2) with the key `K_B^C` before the resulting encryption is encrypted with `K_AB^R`, in the exchange that precedes the Noise handshake, (3) with `K_B^C` when the resulting encryption is encrypted with `k_1` as payload data, (4) with `k_0`, which corresponds to the only case in the Noise handshake where the encryption of `S_A` is sent.

Similarly, `S_B` is sent twice, encrypted (1) with `K_A^C` then the resulting encryption is encrypted with `k_1` as payload data, and encrypted (2) with `K_A^C` then the resulting encryption is encrypted with `k_3` as payload data. In the original Noise handshake, it is not sent during the handshake (it is assumed to be sent as a pre-message before the handshake).

3.4 Implications of clashes in the security guarantees

While the initial purpose for adopting the Noise handshake for Tox has been achieved, i.e. resistance against a KCI attack, which is achieved once the Initiator and Responder reach the stage when they start sending transport messages (See 7.7 from the Noise documentation). Other Noise guarantees are affected by the cookie mechanism.

For instance, the Identity Hiding guarantee, which in the Noise documentation is limited (while being analyzed) to the leakage that might happen to the static public keys of the Initiator and Responder during the handshake. That is, Noise gives a ranking from 0 to 9, to reflect to which level the identities of the parties are hidden (see 7.8 from the Noise documentation for more details). For the case of NoiseIK, this ranking is 4 for the Initiator and 3 for the Responder. In the Noise-based handshake for Tox, these rankings are no longer valid because of the clashes with the cookie mechanism. Besides, the remaining keys have to go through a similar analysis, as the identity hiding property is related to them as well.

Besides this, the security guarantees that were defined in works that formally analyzed Noise or Noise-based handshakes might be affected as well due to the same clashes. For instance, in [8] the guarantee `RSNotSetTwice` reflects the fact that only one remote static key is being used by a party. This might be violated in the Noise-based handshake for Tox because the static public keys are sent more than once.

3.5 On supporting backward compatibility

In order to ensure backward compatibility, it is necessary for users that support the new handshake to be able to support the old handshake as well, which can induce the cookie mechanism clashes we mentioned earlier, including for instance having to send `S_B` during the handshake, or the redundancy in sending `S_A`.

We distinguish between two types of users: users that support the new handshake (to whom we refer as Noise users), and users that only support the old one (to whom we refer as non-Noise users). This distinction gives rise to three categories of communication between these types of users:

1. non-Noise user with non-Noise user
2. non-Noise user with Noise user
3. Noise user with Noise user

Backward compatibility means that a non-Noise user can still communicate with a Noise user via the protocol that the non-Noise user can follow. While this implies generating the cookie as described in Figure 3, one could edit the cookie so that it does not contain the static keys for the third category of communication (Noise user with Noise user).

In the old handshake, the static key of Peer A was communicated to Peer B through the cookie, but now the parties that support the new handshake can communicate this key by just following the NoiseIK specifications, so including this key in the cookie is no longer necessary for this type of user.

Therefore, supporting backward compatibility does not imply using the same cookie mechanism that was adopted for the old handshake for communication between two parties that both support the new handshake.

3.6 Code inspection

We performed a code inspection, on the branch where the new handshake is implemented [5][5] .

The specific files we inspected were `crypto_core.h`, `crypto_core.c`, `net_crypto.h`, and `net_crypto.c`, which are the files that contain the implementation of the new handshake.

3.6.1 Exchange of an unnecessary cookie

The code creates a cookie that is not necessary and not used. That is, this cookie was created by the Responder and included in the payload of `c_2`. This cookie was sent in a step after which the shared session key can be already established, and the parties will already start accepting messages from each other. This cookie was removed by Tobias[6].

[5] In commit `4f16bb2d0ace0bd890b4a7553b3986b742625297`.

[6] Removed in commit `190c0384a57a7369904a6a9e5621a88b6e37c095`

3.6.2 Variable being used instead of another one

In some lines of the code, the variable `CRYPTO_PUBLIC_KEY_SIZE`, is used instead of `CRYPTO_SECRET_KEY_SIZE`. These two variables are intended to set the size of the public key, and the private key respectively, taken from the curve being used for the encryption scheme.

By chance, this has no security implications as both keys are the same size, because the Curve25519 curve used in the handshake is able to use public keys of the same size as secret keys. The misplaced `CRYPTO_PUBLIC_KEY_SIZE` occurrences were corrected by Tobias[7].

3.6.3 Missing step from the Noise handshake.

During the analysis of Tox's Noise-based handshake Tobias discovered that the step of calculating `h_1 = H(h_0, prologue)` is missing from the implementation.

[7]Corrected in commit `2201b8c0384ff5dfa69b9577bf9e870f20b181ee`

4 Conclusion and recommendations

The Noise-based handshake for Tox is composed of the Noise handshake, plus a cookie a mechanism that involves exchanging packets before the Noise handshake, and sets some data that has to be sent as payload data in the handshake. While the KCI attack is prevented thanks to the Noise handshake deployment, other Noise security guarantees are affected by this cookie mechanism.

As such, we strongly recommend performing a formal analysis of the new handshake, including transport messages, and covering the cookie exchange and all the keys that are being used in the new handshake. This can be done using one of the tools that have been used in analyzing other Noise or Noise-based handshakes.

We also strongly recommend replacing the cookie mechanism, with another one that minimally impacts the guarantees of the Noise handshake, if the goal is to adopt the Noise handshake. The design choices that lead to the current cookie mechanism no longer hold, given the integration of the Noise handshake which takes care of how the parties can communicate their static keys to each other.

Finally, it's possible to add extra features to the handshake, as mentioned in [2], such as session rekeying and the use of a pre-shared symmetric key between the Initiator and Responder. The implementation of transport messages, beyond the scope of this report, can also be inspected, along with all the other parts of Tox.

5 Acknowledgement

We would like to thank Tobias for his help with understanding the protocol, both through his thesis and his prompt responses to our related questions.

6 Bibliography

[1] *KCI vulnerability in Tox*. https://github.com/TokTok/c-toxcore/issues/426. Accessed: 2024-03-07.

[2] *Tobias Buchberger Master thesis*. https://pub.fh-campuswien.ac.at/obvfcwhsacc/content/titleinfo/5430137. Accessed: 2024-03-07.

[3] *Tox Documentation*. https://toktok.ltd/spec. Accessed: 2024-03-07.

[4] *Noise Documentation*. http://www.noiseprotocol.org/noise.html. Accessed: 2024-03-07.

[5] *Git Repository for the Noise-based handshake for Tox*. https://github.com/goldroom/c-toxcore/tree/noiseIK_2023_PR. Accessed: 2024-03-07.

[6] *Benjamin Dowling, Paul Rösler, and Jörg Schwenk. Flexible Authenticated and Confidential Channel Establishment (fACCE): Analyzing the Noise Protocol Framework*. https://casa.rub.de/fileadmin/img/Publikationen_PDFs/2020_Flexible_Authenticated_and_Confidential_Channel_Establishment__fACCE__Analyzing_the_Noise_Pr Accessed: 2024-03-07.

[7] *Nadim Kobeissi, Georgio Nicolas. Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols*. https://eprint.iacr.org/2018/766.pdf. Accessed: 2024-03-07.

[8] *.A. Suter-Dörig.Formalizing and Verifying the Security Protocols from the Noise Framework* . https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/noise_suter-doerig.pdf. Accessed: 2024-03-07.

[9] *Jason A. Donenfeld, Kevin Milner. Formal Verification of the WireGuard Protocol*. https://www.wireguard.com/papers/wireguard-formal-verification.pdf. Accessed: 2024-03-07.

[10] *Benjamin Dowling, Kenneth G. Paterson. A Cryptographic Analysis of the WireGuard Protocol*. https://www.wireguard.com/papers/dowling-paterson-computational-2018.pdf. Accessed: 2024-03-07.

[11] *Benjamin Lipp, Bruno Blanchet, Karthikeyan Bhargavan. A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol*. https://inria.hal.science/hal-02396640/document. Accessed: 2024-03-07.

Appendix 1 Testing team

Younes Talibi Alaoui	Younes Talibi Alaoui received his PhD in cryptography from KU Leuven, with a focus on privacy-preserving technologies such as Multi-Party Computation. After his PhD, he started working as a cryptographer across companies.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.