Disco: Modern Session Encryption

David Wong

NCC Group, August 2018

Abstract

At Real World Crypto 2017, Joan Daemen won the Levchin Prize and announced that he believed permutation-based crypto was the future of symmetric cryptography. At the same conference Mike Hamburg introduced Strobe, a symmetric protocol framework capable of protecting sessions as well as building symmetric cryptographic primitives for the single cost of Joan Daemen's permutation Keccak. The next year, at Real World Crypto 2018 Trevor Perrin came to talk about the Noise protocol framework, a modern TLS-like protocol with similar traits but with a focus on flexibility, offering many handshake patterns to choose from in order to authenticate peers of a connection in different ways. Disco is the natural merge of the two projects, creating a new protocol based solely on two unique primitives: Curve25519 and the Keccak permutation (or more correctly its wrapper Strobe). Experimental results show that a library based on Disco can be implemented on top of these two cryptographic primitives with only a thousand lines of code. This, while offering both a flexible way to encryption sessions and a complete cryptographic library for all of an application's needs.

Keywords: Session Encryption, Embedded Devices, SHA-3, Keccak, Duplex construction, Permutation-Based Cryptography, Strobe, Noise, Protocol Framework, Disco, SSL, TLS

1 Introduction

The SHA-3 competition[12] started more than 10 years ago and came to an end in 2012 with the nomination of the Keccak[13] algorithm built from a sponge construction hosting a permutation at its core. Later, a more generic construction was invented called the duplex construction[2], which led to the invention of the Strobe protocol framework[7], a wrapper around the construction that can be used to construct symmetric protocols. On the other side of the field, the Noise protocol framework[14] is a project aiming at making the creation of secure protocols (like SSL/TLS) more flexible, easier to analyze and easier to implement. This paper attempts to merge the two frameworks Noise and Strobe into what we call Disco, a complete protocol framework aiming at simplifying secure protocols and minimizing the need to rely on several cryptographic primitives. The result of this experiment is a cryptographic library that holds in 1000 lines of code. The library can be used to setup (flexible) secure communications between endpoints and as a typical cryptographic library for operations like hashing, deriving keys, signing, encrypting, authenticating, etc.

Section 2 introduces the algorithm behind SHA-3 (a sponge and a permutation), while section 3 presents a similar construction (the duplex construction) and one of its real world applications (Strobe). Section 4 summarizes the efforts behind the Noise protocol framework. Finally, section 5 merges the effort from the previous two sections into one: Disco. Section 6 concludes with experimental results.

2 SHA-3 and Keccak

In 2007, the NIST organized the SHA-3 competition[12]. Unlike the SHA-1 and SHA-2 algorithms that had previously been designed by the NSA, SHA-3 would be open to anyone willing to publicly share their design. 64 teams from all around the world entered the competition in hopes of becoming the new Secure Hashing Algorithm. Five years later, in 2012, Keccak (invented by a Belgian/Italian team including Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche) was designated as the winner[13]. The other candidates who made it to the last round were BLAKE, Grøstl, JH, Skein.

While the NIST had many reason to choose Keccak, an interesting one was its departure in design from previously seen hash algorithms. Indeed, Keccak is built from a sponge, a construction that was invented during the course of the competition.

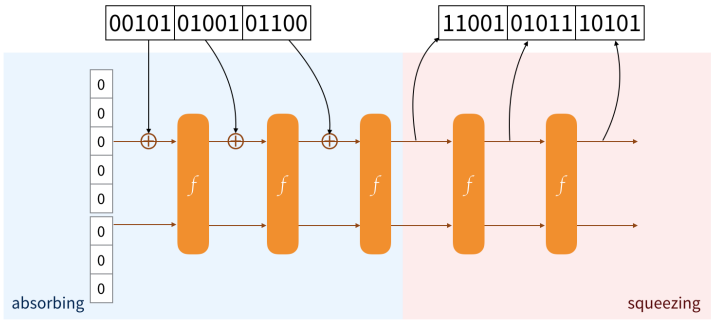

Figure 1: A simplified representation of the sponge construction.

As can be seen in Figure 1, a sponge is a very simple construction. At its core, a permutation f takes an input state of size b (8 bits in the above example) and randomizes it. f is used repeatedly to take on a larger initial input by way of XOR'ing. Recall that our construction is a sponge, so we naturally call this phase "absorbing". To obtain an output, the permutation is used again until enough bits have been obtained, we naturally call this phase "squeezing". The security of the construction comes from the segregation applied to input and output states of the permutation: their first part (top, of 5 bits in our example) is called the **rate**, their second part (bottom, of 3 bits in our example) is called the **capacity**. As can be observed, the capacity is never touched and only the rate is mixed with the original input and read to produce the final output. More details (like padding, size of the capacity, etc.) are involved but for the sake of simplicity we omit them here.

A full security proof[5] exists over the security of the sponge, provided that the permutation used in the construction has no structural distinguishers. Unfortunately, we have no known ways of mathematically proving this property on the permutation and as with the study of block ciphers, cryptanalysts must be invoked and thrown at the algorithm for years in order to obtain an estimation of its security posture as well as to tweak its parameters. As the inventors of Keccak had experience with block ciphers –Joan Daemen being the co-inventor of AES (with Vincent Rijmen)– the design of **Keccak-f** (Keccak's permutation) follows similar patterns.

3 The duplex construction and Strobe

One can wonder if there must only be one absorbing phase and one squeezing phase. And indeed, a construction introduced as the sister of the sponge, called the duplex, breaks this boundary:

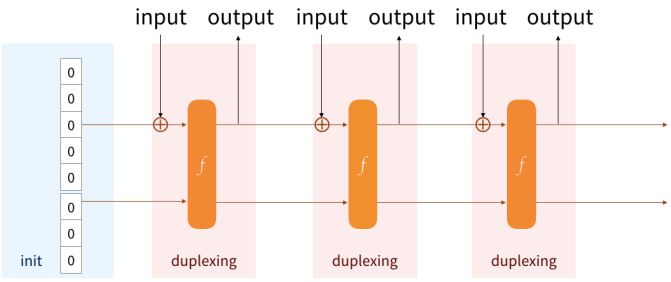

Figure 2: A simplified representation of the duplex construction.

While a set of formal proofs shows security equivalence to the sponge, the duplex construction is quite different in practice. It allows for the construction of more varied symmetric cryptographic primitives like re-seedable pseudorandom number generator, authenticated encryption schemes and the idea of sessions. The latter one encouraged the idea that a secure protocol could be formed around the construction, and thus the **Strobe protocol framework** was born. At its core is a set of public commands permitting access to the duplex functionalities by continuously mutating an internal state.

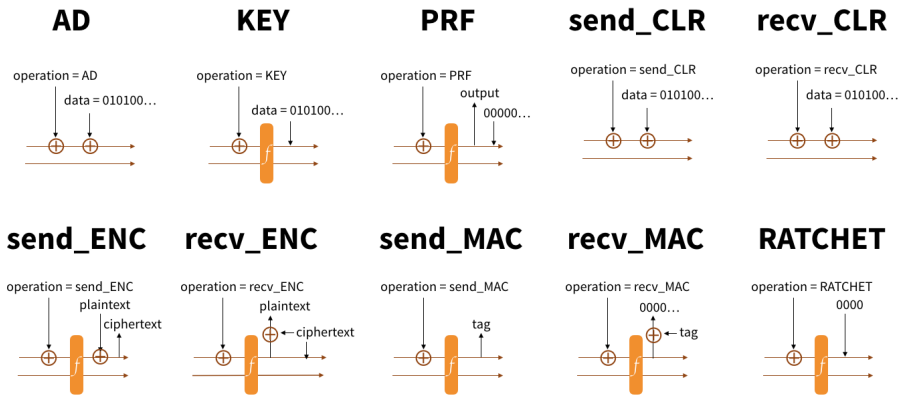

Figure 3: All of the functions publicly exported by the API of Strobe.

These commands can be combined (see figure 3) to produce symmetric cryptographic primitives as well as to build more involved protocols. The Strobe protocol framework is first and foremost a specification around a standardized and common usage of the duplex construction. Once implemented, it typically takes around **a thousand lines of code**, and that includes the permutation Keccak-f as well. These two qualities make Strobe an excellent choice for embedded devices that have memory limitations as well as high-trust systems that do not want to rely on too many cryptographic primitives. In addition, implementations are easy to audit and to use as developers can consider Strobe as a

black box.

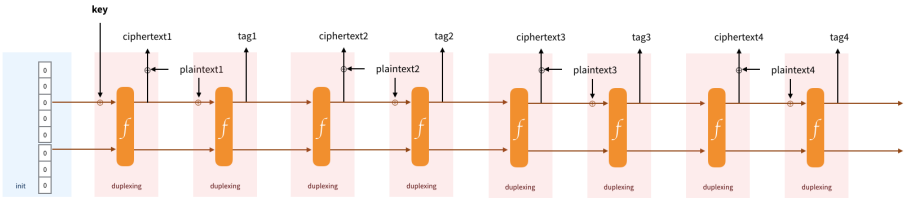

Figure 4: A session built from the duplex construction. Each tag not only authenticate the previous encryption, but any operation that has happened prior to it.

The next section forgets about SHA-3, Keccak-f, the sponge and the duplex construction for a bit, but they will make a return (with Strobe) in section 5.

4 The Noise Protocol Framework

The Noise protocol framework is a specification[14] to build TLS-like protocols. One can implement only parts of it to form a secure communication protocol. While there are limited options for cryptographic primitives (X25519 or X448? Chacha20-Poly1305 or AES-GCM? SHA-256 or BLAKE2s?), the document specifies dozens of **handshake patterns** to chose from. These different patterns allow peers to authenticate each other in different ways, and will consequently provide different security properties at different point of the handshake. Once set on a handshake and a set of cryptographic primitives, the protocol is locked-in and can be implemented in a very linear way. All of this is described in the Noise specification as well as formally verified (with Tamarin[6, 16], ProVerif[8] and CryptoVerif[9]) by different community efforts.

Handshake patterns are a series of directional messages containing tokens that must be digested one by one (on both sides of the connection) by a linear state machine. These patterns do not enforce the use of a public key infrastructure (PKI) and/or x.509 certificates and can be used to authenticate peers with plain keys, pre-shared secrets and even session fingerprints.

```
NX:
  -> e
  <- e, ee, s, es
```

Figure 5: The NX handshake pattern: both sides use ephemeral keys (`e`), but only the server authenticates itself by sending its long-term public key (`s`).

Implementers have to choose what pattern to use and implement the relevant subset of the specification. Once peers decide to eventually use the protocol to secure a communication, the handshake is started and each token is processed

by the peers, dictating to them what needs to be done. For example, `e` means that the sender (resp. receiver) needs to generate an ephemeral keypair and send the public part (resp. receive a public key) whereas `es` means that a peer has to compute the Elliptic Curve Diffie-Hellman (ECDH) key exchange between the client's ephemeral key and the server's static key. Internally, a **running hash** h continuously absorbs all messages that are being sent or received by the protocol, and is authenticated at the end of each turn, effectively providing transcript consistency to the session.

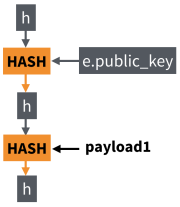

Figure 6: The running hash absorbs every message that transits through the network.

At the same time, a **chaining key** ck is used continuously with outputs of the different key exchanges (that happen during the handshake) to derive new keys. These different keys are used to encrypt messages during and after the handshake.

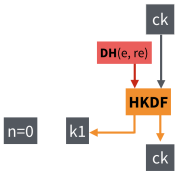

Figure 7: The chaining key is given as input to HKDF along with the key exchange output, the result is used to derive an encryption key as well as the next chaining key.

The following dense diagram is a graphical simplified representation of the *NX* handshake pattern, chosen to illustrate our work as it is closely resembling the typical TLS key exchange between a web browser and a web server.

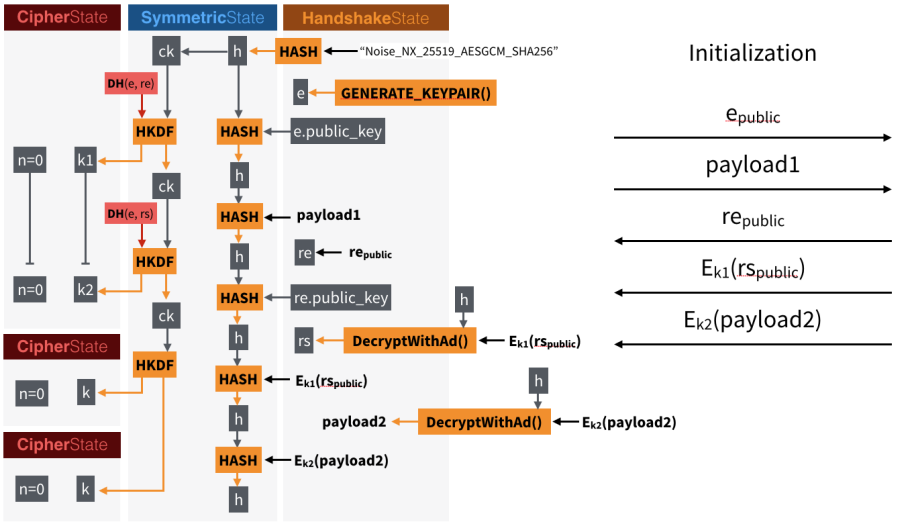

Figure 8: A simplified representation of the *NX* handshake pattern of the Noise protocol framework. As can be seen, three different states are maintained by a peer, one used to store asymmetric keys, one used to store the running hash and the chaining key, one used to store derived keys used for encryption.

As can be seen, while the standard is quite simple compared to something like TLS, and while the state machine is truly linear, neither analyzing nor implementing the protocol appear to be straightforward tasks. The next section will attempt to use Strobe as a black box primitive in order to replace all of the symmetric primitives of Noise (HKDF, HMAC, SHA-2, BLAKE2, AES-GCM, Chacha20-Poly1305) with the Keccak-f permutation.

5 Noise + Strobe = Disco

In the description of the running hash h, we purposefully used the word "absorbing" to point out that a sponge or a duplex construction could be used. Furthermore, the duplex construction also supports the derivation of keys, reseeding and authenticated encryption. This leads us to believe that Strobe could be used to replace all of the complex machinery we've seen in figure 8.

Indeed, this is what we propose with our contribution, that we naturally call **Disco**.

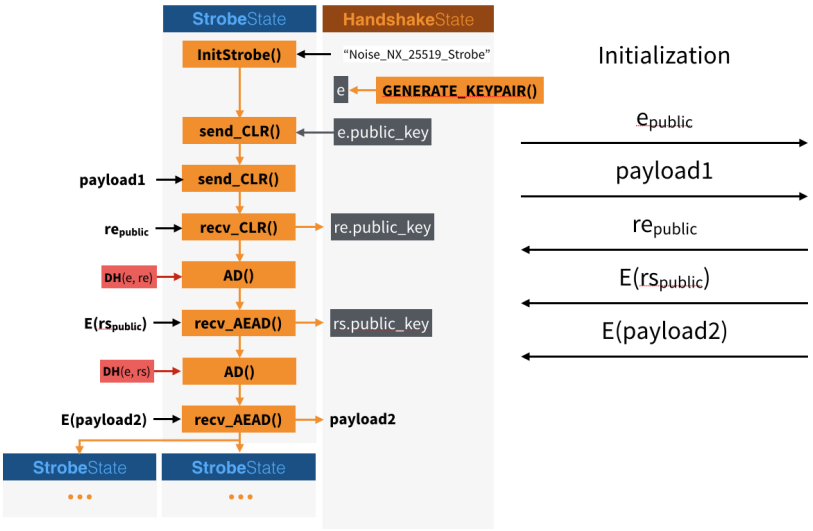

Figure 9: The NX pattern within Disco. Both the CipherState and SymmetricState have been removed in favor of a unique and simple StrobeState.

The `send_CLR` (resp. `recv_CLR`) operation can be used to absorb messages being sent (resp. received) while the similar `AD` operation can be used to absorb key exchange outputs. `send_AEAD` and `recv_AEAD` are used to encrypt and decrypt messages. These functions do not exist in Strobe and are used as shorthands to represent a `send_ENC` (resp. `recv_ENC`) operation followed by a `send_MAC` (resp. `recv_MAC`) operation.

The duplex construction naturally absorbs every operation. No more key derivation is needed as encryption/decryption is influenced by everything that was absorbed previously. A stronger session consistency is obtained in the same way. Finally, we note that from an implementer's perspective, the protocol now seems drastically simpler. From a trust perspective, we now only rely on one single symmetric primitive: Keccak-f.
The construction is so graceful that we can "zoom-in" and discern the internals of the duplex construction during the same handshake.

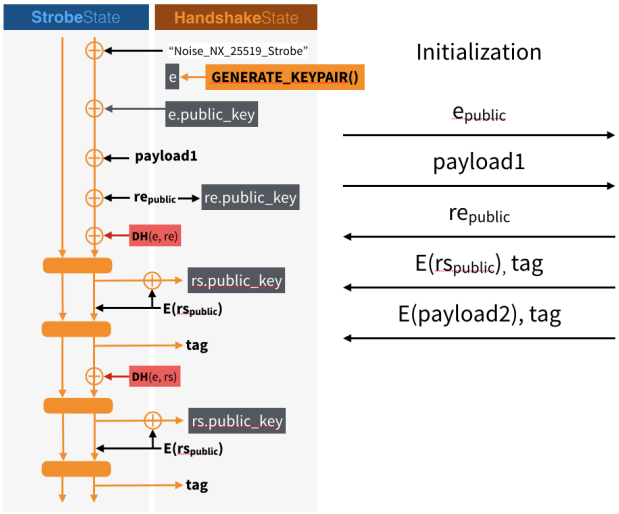

Figure 10: The Disco NX handshake pattern view from the inside. Absorption has been simplified: if the input is larger than the rate, the state needs to be permuted in between smaller absorptions.

Disco is currently being proposed as an extension to Noise[18]. It defines a new `SymmetricState` and a set of functions that acts on a stored Strobe state:

`InitializeSymmetric(protocol_name)` calls `InitializeStrobe(protocol_name)` on the Strobe state.

`MixKey(input_key_material)` calls `AD(input_key_material)` on the Strobe state.

`MixHash(data)` calls `AD(data)` on the Strobe state.

`MixKeyAndHash(input_key_material)` calls `AD(input_key_material)` on the Strobe state.

`GetHandshakeHash()` calls `PRF(32)`. This function should only be called at the end of a handshake, i.e. after the Split() function has been called. This function is used for channel binding[14, Section 11.2].

`EncryptAndHash(plaintext)` returns a ready to be sent payload to the caller by using the `send_ENC()` and `send_MAC()` functions on the Strobe state.

`DecryptAndHash(ciphertext)` returns the received payload by using the `recv_ENC()` and `recv_MAC()` functions on the Strobe state.

`Split()` clones the strobe state and differentiates each one of them via the rec-

ommendations given in [7, Appendix C.1.]. It then returns the pair of Strobe states for encrypting transport messages.

In practice, this `SymmetricState` transparently replaces Noise's `SymmetricState` and `CipherState`. More details are available on the specification of Disco itself[18].

6 Experimental Results

To obtain a better understanding of our contribution, we have implemented Disco in Golang[20] and in C[19]. The protocol was independently implemented in C#[10], Python[11] and Rust[15] as well. The process is pretty straightforward, first implement or re-use a Strobe library, then implement the Noise protocol framework (or modify an existing library) using the Disco specification. The C implementation of Disco is around **1000 lines of code** which includes everything it needs: an X25519 implementation and Strobe (which contains an implementation of Keccak-f and a variant of Schnorr's signature algorithm with X25519). On the other hand, OpenSSL[1] is around 700,000 lines of code and already lists 165 CVEs[2] on its page.

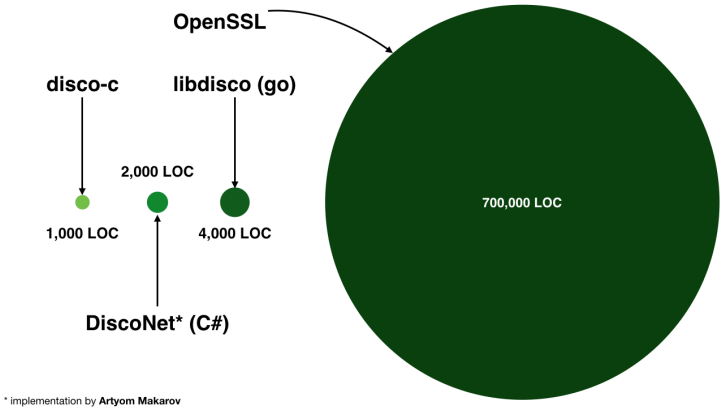

Figure 11: A scaled representation of the lines-of-code count of different codebases.

The Golang implementation was released as a more complete cryptographic library, offering a plug-and-play experience to connect two peers (with a choice of different handshake patterns) and a set of stand-alone cryptographic primitives for all of an application's need. Its implementation is 1000 lines of code. Add Strobe (which contains an implementation of the Keccak-f permutation) and that's a total of 2000 lines of code. Add an implementation of X25519 and we end up with a total of 4000 lines of code. On the other hand, Golang's standard

[1] https://www.openssl.org/
[2] https://www.openssl.org/news/vulnerabilities.html

crypto/tls[3] library is 5000 lines of code not including certificate parsing and the numerous cryptographic primitives it needs to operate.

In order to test the impact of the protocol on embedded devices, some tests for the NX pattern have been executed on STM32 MCUs, specifically an STM32L073RZT6 (Cortex-M0+, running at 32MHz) and an STM32F429ZIT6 (Cortex-M4, running at 180MHz), both compiled with Atollic TrueSTUDIO 9.2.0 (arm-gcc) and optimized with -Os option.

Performance for a client sending 1024 byte payloads are shown in Table 1: for each type of CPU, performance are provided both optimized for space and for time. In the former case, we used the default C code for Keccak and Curve25519 that is provided by [19], while in the latter case we took ASM code for Keccak (eXtended Keccak Code Package[4]) and Curve25519 (two different versions for the Cortex-M0+[5] and for the Cortex-M4[6]) from other public domain repositories, assuring state-of-the-art performance.
A note regarding RAM footprint: those values have been taken with 1B (and not 1024B) messages, in order to benchmark only the protocol minimum RAM usage. The size of the messages will depend, instead, on the application.

CPU	**ASM opt.**	**Footprint (B)**		**Clock Cycles**	
		Flash	**RAM**	**Handshake**	**Payload**
M0+	–	6,192	5,169	398,918,083	517,536
	✓	23,892	3,465	17,807,320	247,515
M4	–	6,046	5,097	89,824,263	347,560
	✓	15,894	3,401	2,077,886	129,143

Table 1: Results on some Cortex-M MCUs

7 Formal Verification

While the focus of this paper is not on formal verification, we include some preliminary results in this section. We modeled the IK handshake of Disco with Tamarin Prover[1], a security protocol verification tool, and have open sourced the models[17]. While the results are not yet conclusive (and might be published later in an update), formal verification is an helpful process in order to make sure that the security properties of Disco retain the security properties of the Noise protocol framework.

Modeling Strobe. Thanks to built-ins for diffie-hellman and the XOR operation, most of the operations of Disco except for Strobe can easily be modeled.

[3] https://golang.org/pkg/crypto/tls/
[4] https://github.com/XKCP/XKCP
[5] http://munacl.cryptojedi.org/curve25519-cortexm0.shtml
[6] https://github.com/Emill/X25519-Cortex-M4

In order to model Strobe calls, we need to model the underlying duplex construction. We note that the security of the duplex construction is equivalent to the one of the sponge construction (as seen in [3, lemma 3]), and that a keyed sponge construction can be modeled as a random oracle[4]. For this reason, we model each calls to Strobe as a single call to a random oracle function in Tamarin Prover via `functions: sponge\1`. For example, if we previously absorbed `input1` we model a call to `send_ENC()` followed by a call to `send_MAC()` as:

```
1 input2 = <'ENC', input1>
2 ciphertext = sponge(input2) ⊕ plaintext
3 input3 = <plaintext, input2>
4 input4 = <'MAC', input3>
5 tag = sponge(input4)
6 input5 = <'0000000000000000', input4>
```

Figure 12: Tamarin code for successive calls to `send_ENC()` and `send_MAC()`

The sponge call on line 5 is equivalent to:

```
sponge(<'MAC', plaintext, 'ENC', input1>)
```

Note that to circumvent the pair notation of Tamarin Prover we are forced to absorb inputs in the reverse order, which does not matter as this is an abstraction of the sponge calls. For the same reason, we use `'MAC'` as a marker of the operation, abstracting Strobe's padding and ignoring the destination (`'recv_MAC'` or `'send_MAC'`) as this is an implementation-specific concern.

Strobe forces permutations for operations involving the network, this is to make sure that previous operations are absorbed prior to involving new sensitive operations. To model this in Tamarin Prover we call the `sponge()` function at this point in time. This can be seen on line 2 and line 5 of figure 12.

Formally proving the Noise security properties. In order to prove the security properties of the IK handshake, we wrote four type of lemmas:

1. Sanity checks. To obtain confidence in the translation to Tamarin, we wrote a few lemmas that checks that the protocol is correctly modeled and can properly run.
2. Standard security properties. To ensure that session keys cannot be leaked from the protocol, and that long-term keys remain secret to active adversaries, generic protocol security lemmas were written.
3. Payload security properties. [14, Section 7.7 of the Noise Protocol Framework] claims specific security properties like forward-secrecy, KCI-resistance and replay-resistance on each of the handshake pattern messages, as well as the post-handshake messages.

4. Identity hiding properties. [14, Section 7.8 of the Noise Protocol Framework] claims specific properties about an adversary's knowledge of the peers' long term static public keys.

We currently have proved all lemmas except for the payload security ones (3). This is due mostly to a lack of time as the use of the XOR built-in in Tamarin Prover make advances dramatically slow. For comparison, NoiseExplorer documents that the analysis of IKpsk2 with proverif took 21 days to complete[7]. Due to these limitations, the work needs to be completed at a later point in time. This is nonetheless encouraging results, and shows that modeling a Disco handshake in formal verification tools is quite natural.

8 Conclusion and Future Work

In this document We have introduced Disco, a simple protocol to encrypt sessions based on both Noise and Strobe. Disco can be implemented in only a thousand lines of code and its trust can be traced back directly to two solid cryptographic primitives: Curve25519 and Keccak-f. At the same time, thanks to the use of Strobe, an entire cryptographic library results from implementations of Disco in different programming languages. Future work includes proving all useful Disco handshakes formally, developing and testing new implementations of Disco, deciding on a signature algorithm (Ed25519, qDSA, Strobe's schnorr-variant, etc.) and standardizing the protocol as an RFC.

Acknowledgements

Thanks to Matteo Bocchi and Ruggero Susella who contributed to the experimental results on embedded devices.
Thanks to Artyom Makarov and Michael Rosenberg for implementing Disco in other programming languages and providing feedback on the process.
Many thanks to Mike Hamburg and Trevor Perrin who are the two legs of this project.
Thanks to Katriel Cohn-Gordon and the Tamarin Prover team for answering all my questions about formal verification.
Thanks to Thomas Pornin for discussions and help with the C implementation.

[7] https://moderncrypto.org/mail-archive/noise/2019/002001.html

References

[1] David Basin et al. *Tamarin Prover*. https://tamarin-prover.github.io. 2012.

[2] Guido Bertoni et al. *Duplexing the sponge: single-pass authenticated encryption and other applications*. Cryptology ePrint Archive, Report 2011/499. https://eprint.iacr.org/2011/499. 2011.

[3] Guido Bertoni et al. *Duplexing the sponge: single-pass authenticated encryption and other applications*. https://keccak.team/files/SpongeDuplex.pdf. 2011.

[4] Guido Bertoni et al. *On the security of the keyed sponge construction*. 2011.

[5] G. Bertoni et al. *Cryptographic sponge functions*. SHA-3 competition (round 3). https://keccak.team/files/CSF-0.1.pdf. 2011.

[6] Jason A. Donenfeld and Kevin Milner. *Formal Verification of the WireGuard Protocol*. https://www.wireguard.com/papers/wireguard-formal-verification.pdf. 2017.

[7] Mike Hamburg. *The STROBE protocol framework*. Cryptology ePrint Archive, Report 2017/003. https://eprint.iacr.org/2017/003. 2017.

[8] Nadim Kobeissi. *Noise Explorer*. https://noiseexplorer.com. 2018.

[9] Benjamin Lipp. *A Mechanised Computational Analysis of the WireGuard Virtual Private Network Protocol*. https://benjaminlipp.de/master-thesis/master-thesis.pdf. 2018.

[10] Artyom Makarov. *.Net core implementation of libdisco*. https://github.com/Fasjeit/DiscoNet. 2018.

[11] Artyom Makarov. *PyDisco: Python 3 implementation of libdisco*. https://github.com/Fasjeit/PyDisco. 2018.

[12] NIST. *SHA-3 Project*. https://csrc.nist.gov/projects/hash-functions/sha-3-project. 2007.

[13] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. https://csrc.nist.gov/publications/detail/fips/202/final. 2015.

[14] Trevor Perrin. *Noise protocol framework*. https://noiseprotocol.org/. Oct. 2017.

[15] Michael Rosenberg. *Disco-rs: A Rust implementation of the Disco protocol framework*. https://github.com/rozbb/disco-rs. 2018.

[16] A. Suter-Dorig. *Formalizing and Verifying the Security Protocols from the Noise Framework*. https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/noise_suter-doerig.pdf. 2018.

[17] David Wong. *Disco Formal Verification*. https://github.com/mimoo/disco-formal-verification. 2019.

[18] David Wong. *Disco protocol framework*. http://discocrypto.com/disco.html. 2017.

[19] David Wong. *EmbeddedDisco: A tiny C cryptographic library to encrypt sessions, authenticate messages, sign, hash, etc. based only on SHA-3 and Curve25519*. https://github.com/mimoo/disco-c. 2018.

[20] David Wong. *libdisco*. https://www.discocrypto.com. 2017.