# IoTDisco: Strong yet Lightweight End-to-End Security for the Internet of Constrained Things

Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Peter Y. A. Ryan

DCS and SnT, University of Luxembourg,
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{hao.cheng,georgios.fotiadis,johann.groszschaedl,peter.ryan}@uni.lu

**Abstract.** Most widely-used protocols for end-to-end security, such as TLS and its datagram variant DTLS, are highly computation-intensive and introduce significant communication overheads, which makes them impractical for resource-restricted IoT devices. The recently-introduced Disco protocol framework provides a clean and well-documented basis for the design of strong end-to-end security with lower complexity than the (D)TLS protocol and no legacy baggage. Disco consists of two sub-protocols, namely Noise (known from e.g., WhatsApp) and Strobe, and is rather minimalist in terms of cryptography since it requires only an elliptic curve in Montgomery form and a cryptographic permutation as basic building blocks. In this paper, we present IoTDisco, an optimized implementation of the Disco protocol for 16-bit TI MSP430 microcontrollers. IoTDisco is based on David Wong's EmbeddedDisco software and contains hand-written Assembly code for the prime-field arithmetic of Curve25519. However, we decided to replace the Keccak permutation of EmbeddedDisco by Xoodoo to reduce both the binary code size and RAM footprint. The experiments we conducted on a Zolertia Z1 device (equipped with a MSP430F2617 microcontroller) show that IoTDisco is able to perform the computational part of a full Noise NK handshake in 26.2 million clock cycles, i.e., 1.64 seconds when the MSP430 is clocked at 16 MHz. IoTDisco's RAM footprint amounts to 1.4 kB, which is less than 17% of the overall RAM capacity (8 kB) of the Zolertia Z1.

**Keywords:** Internet of Things (IoT) · Security protocol · Elliptic curve cryptography · Cryptographic permutation · Efficient implementation

## 1 Introduction

The concept of *End-to-End (E2E) security* refers to the protection of data in transit, i.e., while being transmitted from a source to the intended destination (usually over an insecure network like the Internet) [29]. Strong E2E security is crucial for a wide spectrum of Internet applications and services, ranging from various kinds of e-commerce over personal communication to Web-based leisure and entertainment, e.g., social networks. In general, E2E security involves E2E authentication, E2E key agreement, and E2E encryption and integrity protection, all of which has to be implemented in a way that no third party, not even

the Internet Service Provider (ISP), can learn anything about the transmitted data (except of communication meta-data), which is far from trivial to achieve in practice [22]. The de-facto standard for E2E-secure communication over the Internet is the *Transport Layer Security (TLS)* protocol [26], previously known as *Secure Sockets Layer (SSL)* protocol. Originally developed in the 1990s, the TLS/SSL protocol has undergone various revisions since then to strengthen its security, e.g., by adding new ciphers to replace broken ones like RC4 [1]. TLS is a modular protocol comprising a few sub-protocols, of which the handshake protocol and the record protocol are particularly important [26]. The former is responsible for authentication and key establishment and uses classical public-key cryptosystems (e.g., RSA and Diffie-Hellman) or more recent elliptic-curve schemes (ECDSA, ECDH [15]). On the other hand, the record protocol ensures data encryption and integrity through symmetric algorithms, e.g., AES.

TLS is a highly complex protocol and challenging to implement for a couple of reasons. First, as stated above, achieving E2E security in general, and E2E authentication in particular, is a difficult problem for which no easy or simple solution exists (and likely never will exist [22]). In addition, also certain design decisions like *algorithm agility* have added to the complexity of TLS. Algorithm agility means that the TLS protocol supports various combinations of cryptographic algorithms, called *cipher suites*, and the client and server agree on one of them dynamically during the handshake phase. For example, in version 1.3 of the TLS protocol, a server can authenticate itself to a client via certificates based on three different signature algorithms (RSA, ECDSA, EdDSA [4]), and each of them comes with at least two levels of security. Another reason for the complexity of TLS implementations is that old(er) versions of the protocol, in particular TLS v1.2, are still widely used and have to be supported to ensure backwards compatibility. Furthermore, in the course of approximately 30 years of development (i.e., bug fixing) and evolution, the protocol became overloaded with niche features and extensions for very specific purposes with questionable relevance in real-world settings. The enormous complexity of TLS has been the root cause for a multitude of problems. First, it has been key enabler of various security flaws and vulnerabilities in the protocol itself (e.g., the re-negotiation attack analyzed in [12]), the underlying cryptography (e.g., the attacks known as BEAST and Lucky13 [13]), and different implementations of TLS (e.g., the "Heartbleed" bug found in OpenSSL [11]). Second, the high complexity of the protocol translates directly to high resource consumption, i.e., large code size and memory (RAM) footprint, which makes TLS relatively unattractive for the *Internet of Things (IoT)* and its billions of constrained devices.

The Internet Engineering Task Force (IETF) describes in RFC 7228 [7] three classes of constrained devices, depending on the code complexity (i.e., the size of ROM or flash) and the size of state/buffers (i.e., RAM) they can handle, see Table 1. Class 0 devices (C0 for short) are so limited in terms of memory and processing capabilities that they will "not have the resources required to communicate directly with the Internet in a secure manner." Class-1 (C1) devices are also constrained and can not easily communicate with other Internet nodes

**Table 1.** Classes of constrained devices according to RFC 7228 [7].

| Name | Data size (e.g., RAM) | Code size (e.g., flash) |
|---|---|---|
| Class 0, C0 | $\ll$ 10 kB | $\ll$ 100 kB |
| Class 1, C1 | $\sim$ 10 kB | $\sim$ 100 kB |
| Class 2, C2 | 50 kB | 250 kB |

through a complex protocol like TLS. However, they are capable enough to use protocols specifically designed for such devices and participate in conversations without the help of a gateway node. Indeed, a full-fledged TLS implementation like OpenSSL [31] contains over 500,000 lines of code and has a (binary) code size of several 100 kB, which is even beyond the resources of C2 devices. There exist a few optimized (D)TLS libraries for embedded systems, such as ARM's mbedTLS, but they nonetheless need a code space of around 50 kB as reported in [27]. Such code size is theoretically feasible for C1 devices, but not practical since it leaves only about half of the total ROM (resp., flash) capacity for the operating system and application(s). Two examples of E2E security protocols that have been specifically developed to be suitable for limited devices are the so-called Diet EXchange (DEX) variant [23] of IETF's Host Identity Protocol (HIP) and the Ephemeral Diffie-Hellman Over COSE (EDHOC) [28] protocol (which is still in development). While both protocols can be optimized to have a binary code size of less than 20 kB, they are still (unnecessarily) complex due to algorithm agility. Some EDHOC cipher suites are (unnecessarily) slow since they use cryptosystems that are nowadays considered dated and efficiency-wise not state-of-the-art anymore (e.g., ECDH with NIST curves or AES-GCM).

*Disco* is a protocol framework developed by David Wong [34] with the goal of simplifying the design and implementation of lightweight security protocols with high efficiency, but without making compromises regarding security. The Disco framework is specified as an extension of the well-known *Noise* protocol framework of Trevor Perrin [25]. Disco can be seen as a clean-slate approach to achieve end-to-end secure communication with much lower complexity and less legacy overheads than TLS. It has the potential to become a viable alternative to (D)TLS in the IoT, especially in settings where all communication endpoints are managed or controlled by one single entity. Disco supports a subset of the Noise handshakes for key exchange and (optionally) authentication. Noise has been widely adopted in the past couple of years; for example, it is used by the instant messenger WhatsApp [32] and the WireGuard VPN tunnel (part of the Linux kernel since version 5.6). WhatsApp is estimated to have more than two billion users worldwide, who send each other tens of billions of text messages per day [22]. Every Noise protocol starts with a handshake phase in which two parties exchange their Diffie-Hellman (DH) public keys and perform a sequence of DH operations, hashing the DH results into a shared secret key. The Noise framework comes with a collection of common "handshake patterns" with well-defined security properties (i.e., various kinds of confidentiality, integrity, and authenticity guarantees). Besides Noise, Disco also uses *Strobe*, a framework to

build symmetric cryptosystems and protocols [14]. Strobe goes beyond modes of operation (or modes of use) and supports the design of secure bi-directional communication channels. Disco is rather minimalist in terms of cryptosystems since the full protocol, including both Noise and Strobe, requires just two basic low-level components: Curve25519 [2] and the Keccak permutation [6].

*EmbeddedDisco* is a prototype implementation of Disco developed by David Wong that targets embedded environments [35]. The protocol itself (i.e., Noise and Strobe) consists of just about 1000 lines of C code, excluding the low-level cryptographic functions. EmbeddedDisco uses the Curve25519 implementation of TweetNaCl [5], a compact cryptographic library that supports variable-base scalar multiplication and fits into just 100 tweets. The Keccak implementation of EmbeddedDisco is also very compact (i.e., "tweetable") and fits in only nine tweets. Unfortunately, these tweetable implementations are extremely slow on microcontrollers, which makes EmbeddedDisco rather inefficient. We present in this paper *IoTDisco*, the first optimized implementation of the Disco protocol for C1 devices equipped with an ultra-low-power 16-bit MSP430(X) microcontroller [30]. IoTDisco replaces the Keccak permutation by *Xoodoo* [9], a modern permutation (designed by largely the same team as Keccak) with a state of 48 bytes instead of 200 bytes, thereby reducing the RAM consumption. Both the prime-field arithmetic for Curve25519 and the Xoodoo permutation are written in MSP430 Assembly to achieve high speed and small code size. The IoTDisco prototype we benchmarked performs a Noise NK handshake, which means the server gets authenticated to the client via a static public key that is known in advance (e.g., pre-deployed on the device), but the client is not authenticated to the server. Our experimental results show that Disco-based E2E security is not only feasible for C1-class IoT devices, but actually practical for real-world applications since the computational part of the handshake can be executed in just 1.64 seconds when our target device (an MSP430F2617 microcontroller) is clocked at 16 MHz. IoTDisco consumes 1.4 kB RAM and has a binary code size of 11.6 kB, which is less than 15% of the RAM/flash capacity of a C1 device.

## 2 Preliminaries

Wong's Disco specification [33] is written as an extension of Noise (and not as a self-contained protocol specification), which makes sense since the handshake phase of Disco is largely based on the Noise protocol framework. However, the subsequent transport phase, in which symmetric cryptosystems are utilized, is based on Hamburg's Strobe protocol framework. In this section, we first give an overview of both Noise and Strobe, and thereafter explain how Disco combines them into a single protocol framework.

### 2.1 Noise Protocol Framework

Noise, as specified in [25], is not a protocol but rather a framework to facilitate the creation of custom E2E security protocols that are tailored for certain use

cases. This protocol framework has shown to be especially beneficial in settings where a single entity controls/manages all communication endpoints, as is the case for a range of Internet applications, e.g., WhatsApp (the first widespread adoption of a Noise-based protocol) and also for many IoT applications. There are different reasons why designing a custom E2E security protocol can make sense; for example, the target application may need a special feature that none of the existing protocols (e.g., TLS, SSH, IKE/IPSec) offers and extending one of them turns out to be a non-trivial task. Likely more common is the situation where an application only needs a subset of the features of, e.g., TLS and the application developers would prefer a simpler and "lighter" solution. The Noise framework facilitates the design and security analysis of custom E2E protocols through the definition of a small set of basic elements called *tokens* (in essence DH keys or DH operations) along with well-documented rules for combing and processing them. A Noise-based protocol is composed of three layers: (i) a thin negotiation layer, (ii) a DH-based handshake layer (which, in fact, uses ECDH as underlying primitive), and (iii) a transport layer for the secure transmission of application data. Apart from ECDH, Noise-based protocols also employ two symmetric cryptosystems, namely an algorithm for Authenticated Encryption with Associated Data (AEAD) and a hash function, the latter of which serves to compute protocol transcripts and to derive AEAD keys.

One of the distinguishing features of Noise is a clear separation between the negotiation phase, in which initiator and responder agree on a common handshake pattern (including whether/how the parties are authenticated), and the actual execution of the handshake. This contrasts with TLS, where negotiation and handshake are "intertwined" and, therefore, the sequence of cryptographic operations performed by the client and server depends on negotiation decisions (e.g., protocol version, cipher suite, client authentication, etc) made *during* the handshake [26]. When using Noise, many of such run-time decisions become, in fact, design-time decisions within a framework, i.e., the protocol designer has to decide which handshake structure fits best for the target application; this includes decisions like who is authenticated to whom and which cryptosystems are employed. Remaining run-time decisions, if any, are separated out from the rest of the protocol and combined together, thereby enabling the handshake to become a straight (linear) sequence of messages and cryptographic operations without any branches apart from error handling. Such a linear execution profile reduces the run-time complexity of the handshake (compared to TLS) and also simplifies the implementation and testing of Noise-based protocols.

The core component of every Noise handshake is an *Authenticated Key Exchange (AKE)* protocol based on DH (in fact ECDH) that can be instantiated with two different elliptic curves: Curve25519 and Curve448. Depending on the concrete handshake pattern, either none, one, or both involved parties become authenticated. Each party has an ephemeral key pair, which is used to generate a fresh shared secret, and, optionally, a long-term (i.e., static) key pair for the purpose of authentication. Many classical AKE protocols in the literature have in common that the static keys are signature keys, i.e., authentication is done

through the generation and verification of digital signatures. Examples for this kind of AKE range from basic "Signed DH," where each party simply signs its own ephemeral DH key using the static private key, to advanced protocols like SIGMA [18], which requires each side to generate a signature over both public DH keys and compute a Message Authentication Code (MAC) of the signer's identity with a secret key derived from the shared DH secret. Alternatively, an AKE protocol can use DH for both key exchange and authentication (i.e., the static key-pairs are DH key-pairs); well-known examples are NAXOS [19] and MQV [20]. Also Noise follows this approach, which has two advantages: (i) an implementation only needs DH but no signature scheme[1] and (ii) the messages can be significantly shorter (see, e.g., Fig. 1 in [28]). The basic idea is to derive the shared secret not solely from the result of ephemeral-ephemeral DH, but to also include DH values combining ephemeral with static keys. For example, in order to authenticate the responder to the initiator, the latter has to perform a DH operation using its own ephemeral private key and the responder's static public key, whereas the responder uses its static private key and the initiator's ephemeral public key. The only responding party that is able to compute the correct shared secret is the party in possession of the static private key.

Another common feature of Noise-based handshakes is that the handshake messages are not limited to (public) DH keys but may also contain application data as "handshake payloads." These early payloads can be AEAD-encrypted as soon as at least one DH operation has been carried out; in certain cases it is even possible to encrypt the payload of the very first message of the handshake (e.g., if the responder's static public key was pre-distributed). The AEAD keys and nonces are derived from a so-called *chaining key*, which gets updated (and gradually evolves) with the output of each DH operation. Thus, the handshake payloads have normally weaker security guarantees than the transport payloads that follow after the handshake. Besides the chaining key, the two parties also maintain a *handshake hash* (essentially a transcript of handshake messages) to ensure they have a consistent view of the handshake.

### 2.2 Strobe Protocol Framework

The transport layer of Noise corresponds to the record layer of TLS; both protect the exchange of application data with the help of symmetric cryptographic

[1] Depending on the application, signatures (e.g., in the form of certificates) may still be necessary to confirm a cryptographically-secure binding between a static public key and the identity of an entity. However, in such case, a Noise-based protocol has to support only signature verification, but not the signing operation. Note that the provision of evidence for the binding of an identity to a static public key is outside the scope of the Noise specification. More concretely, [25, Sect. 14] states that "it is up to the application to determine whether the remote party's static public key is acceptable." Section 14 of [25] also outlines some methods to ensure a static public key is genuine and trustworthy: certificates (which may be passed in a handshake payload), pre-configured lists of public keys, or pinning/key-continuity approaches where parties remember the public keys they encounter.

algorithms. Transport payloads in Noise are secured through an AEAD scheme and a hash function; the latter is also the main component of a Key-Derivation Function (KDF). Strobe is based on the idea that all symmetric cryptographic operations needed for a secure transport protocol can be efficiently designed on top of a single low-level primitive, namely an un-keyed permutation. Similar to Noise, Strobe is not a protocol but a protocol framework; more concretely, it is framework for building secure transport protocols [14]. Strobe-based protocols operate on a "wrapper" around the *duplex construction* [9], which elevates the duplex into a stateful object, i.e., an object that maintains its state across an arbitrary sequence of absorb and squeeze phases. The specification [14] defines a simple API for a Strobe object to perform authenticated encryption, pseudo-random number generation, hashing, and key derivation. Strobe's main design principle is that the cryptographic output from any step shall not only depend on the directly-provided input (e.g., plaintext, nonce, and key if the operation is encryption), but also all preceding inputs processed in the session. The state of the permutation holds a "running hash" of the protocol transcript, which is the sequence of all operations and data as seen by the application layer. Strobe maintains such a running hash on both sides, making it easy to find out when one side diverged from the protocol, e.g., due to a corrupted or lost message.

Besides the messages that are sent back and forth, the running hash of the protocol transcript also includes metadata to ensure the sematic integrity of the cryptographic operations. The behavior of any operation is determined by five flags (one indicating metadata operations) for which Strobe reserves a byte in the rate-part of the permutation. A further rate-byte is used for tracking the start-position of an operation within the rate-part, i.e., the number of bytes in a Strobe block is always two bytes less than the rate of the permutation.

### 2.3 Disco Protocol Framework

Disco merges Noise and Strobe into one single protocol framework that aims to facilitate the design (and implementation) of custom security protocols. Disco improves Noise in two main aspects: it simplifies the symmetric cryptographic operations performed in the handshake phase and reduces the number of low-level primitives. A Noise handshake as described in [25] requires each party to maintain three objects, two of which contain hashes, keys, and nonces that are inputs or outputs of symmetric cryptographic operations. Disco replaces these two objects by a single *StrobeState* object, acting as an opaque cryptographic scheme based on the Keccak-$f$[1600] permutation. Using a permutation allows for simpler transcript hashing (because all the data from previous operations is naturally absorbed) and simpler encryption/decryption of handshake payloads (since no dedicated key derivation has to be carried out).

Disco also replaces the original transport layer of Noise by a Strobe-based transport mechanism. After completion of the handshake, the final StrobeState object is split up into two objects if full-duplex communication is desired, one for each direction. Each channel operates individually in half-duplex mode.

## 3 Implementation Details

Our target platform to assess the computational cost of IoTDisco is a Zolertia Z1 IoT device housing a low-power 16-bit MSP430F2617 microcontroller from Texas Instruments. The MSP430(X) architecture is based on the von-Neuman memory model, which means code and data share a unified address space, and there is a single address bus and a single data bus that connects the CPU core with RAM, ROM/flash memory, and peripheral modules. Its instruction set is rather minimalist, consisting of merely 27 core instructions, and supports seven addressing modes, including modes for direct memory-to-memory operations without an intermediate register holding (similar to CISC architectures). Some MSP430 models, such as the MSP430F2617, have a memory-mapped hardware multiplier capable to carry out $(16 \times 16)$-bit multiply and multiply-accumulate operations [30]. The MSP430F2617 is equipped with 8 kB SRAM and features 92 kB flash memory, i.e., it can be seen as a typical C1 device.

Our IoTDisco prototype is largely based on David Wong's EmbeddedDisco software, but we modified its Noise and Strobe component in order to improve efficiency on 16-bit MSP430(X) microcontrollers. First, we replaced the plain C implementation of Curve25519, which is based on TweetNaCl, by an optimized C implementation with hand-written Assembly code for the underlying prime-field arithmetic. Furthermore, we replaced the Keccak permutation by Xoodoo and also modified Strobe to become *Strobe Lite* as described in Appendix B.2 of [14]. In this section, we first describe our optimized Curve25519 and Xoodoo implementations, and then explain how a Noise-NK handshake is executed.

### 3.1 Curve25519

Our implementation of Curve25519 is a modified and improved version of the ECC software for MSP430(X) microcontrollers presented in [21]. This library is not purely optimized for speed but aims for a trade-off between execution time and binary code size. The elements of the underlying 255-bit prime field $\mathbb{F}_p$ are stored in arrays of unsigned 16-bit integers, i.e., arrays of type `uint16_t`. All low-level field-arithmetic functions are written in MSP430 Assembly language to reduce the execution time. Apart from inversion, the arithmetic functions do not execute operand-dependent conditional statements like jumps or branches (i.e., their execution time is constant), which contributes to achieve resistance against timing attacks. The $\mathbb{F}_p$-inversion is based on the Extended Euclidean Algorithm (EEA), but uses a "multiplicative masking" technique to randomize its execution time and thwart timing attacks (see [21] for details).

A scalar multiplication on Curve25519 can be implemented using either the Montgomery form or the birationally-equivalent *Twisted Edwards (TE)* model of the curve [3]. The former is beneficial for variable-base scalar multiplication (e.g., to derive a shared secret in ECDH key exchange) thanks to the so-called *Montgomery ladder* [8], which is not only fast but also provides some intrinsic resistance against timing attacks due to its highly regular execution profile. On the other hand, when a fixed-base scalar multiplication needs to be performed

(e.g., to generate an ephemeral key pair), our ECC software uses the TE form and takes advantage of Hişil et al's fast and complete addition formulae based on extended coordinates [16]. To be more concrete, the scalar multiplication is carried out via a so-called *fixed-base comb method* [15] with a radix-$2^4$ signed-digit representation of the scalar and uses a lookup table of eight pre-computed points. Similar to the Montgomery ladder, our fixed-base comb method is able to resist timing attacks. The pre-computed points are given in extended affine coordinates and occupy 768 bytes altogether in flash memory.

### 3.2 Xoodoo

*Xoodyak* [9] is a versatile cryptographic scheme that was developed by a team of cryptographers led by Joan Daemen, who is also one of the designers of the two NIST standards AES and SHA-3. At the heart of Xoodyak is Xoodoo, an extremely lightweight permutation that shares some similarities with Keccak's permutation and can potentially serve as "drop-in replacement." However, the state of Xoodoo is much smaller (384 versus 1600 bits), making it more IoT-friendly since it can be optimized to occupy less space in RAM and flash than Keccak. IoTDisco instantiates Xoodoo with a capacity of 256 bits (to achieve 128-bit security), which means the rate is 128 bits (16 bytes). As mentioned in Subsect. 2.2, Strobe dedicates two bytes in the rate-portion of the permutation to special purposes; one byte holds five flags and the other is used to track the beginning of a Strobe operation, see [14, Sect. 4.1] for further details. Since the operations for metadata are small in Strobe, it is normally not very efficient to execute the permutation for each operation, especially when the rate is large as in Keccak-$f$[1600]. To reduce overheads, Strobe packs multiple operations into one block when possible, which explains why keeping track of the start-position of an operation is necessary. However, when using a small permutation, such as Xoodoo, it makes sense to always begin a new block for every new operation (the resulting Strobe variant is called *Strobe Lite* in [14, Sect. B.2]). IoTDisco follows this approach and, therefore, a Strobe block is 15 bytes long (i.e., one byte less than the nominal rate of Xoodoo with a capacity of 256 bits).

We implemented Xoodoo from scratch in MSP430 Assembly language. One of the main challenges was to find a good register allocation strategy so as to reduce the number of load/store operations. Another challenge was to perform multi-bit shift and rotations of 32-bit words efficiently, which is important since MSP430 microcontrollers can only shift or rotate a register one bit at a time.

### 3.3 Noise NK Handshake

Noise-NK is one of 12 so-called *fundamental handshake patterns* for interactive protocols that are described in the Noise specification [25] and implemented in EmbeddedDisco and also IoTDisco. This handshake pattern authenticates the responder through a long-term, i.e., static, public DH key, which is known (and trusted) by the initiator. On the other hand, the initiator is not authenticated and does, therefore, not have a static public key. A real-world example for this

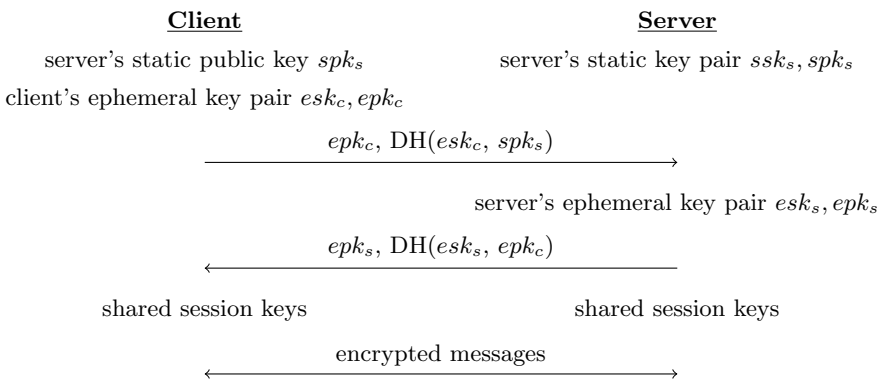


**Fig. 1.** Simplified Noise-NK handshake pattern.

kind of authentication scenario is a mobile-device application that connects to a webserver using certificate (i.e., public-key) pinning. An example with more relevance for the IoT is secure software update, i.e., a device regularly connects to a server to check whether software updates are available and, if this is the case, downloads and installs them. While it is obvious that authenticating the server to the client is an important security requirement, there is normally no need to authenticate the client because most software vendors provide patches for free. A minimalist implementation of such a secure software update service could pre-distribute the server's static public key, e.g., by "hard-coding" it in the IoTDisco software before the device gets deployed.

Figure 1 illustrates the main operations and messages when performing an NK handshake with IoTDisco/EmbeddedDisco. The handshake layer operates on a *HandshakeState* object that contains, among some other things, the local ephemeral and static key pair (if such keys exists) as well as the other party's ephemeral and static public key. At first, the client and the server initialize the HandshakeState by calling the function `Disco_Initialize()` with the server's static key and a token representing the NK pattern. Then, the client assembles its handshake message through the function `Disco_WriteMessage()`, using the HandshakeState object as input. This function generates the client's ephemeral key pair $esk_c, epk_c$, performs a DH computation on $esk_c$ and $spk_s$, and absorbs the results of all these operations into the client's StrobeState object, which is contained in the HandshakeState object. The client sends this first handshake message, consisting of the key $epk_c$ and the DH result of $esk_c$ and $spk_s$, to the server. Having received the message, the server uses `Disco_ReadMessage()` to (i) extract $epk_c$, (ii) compute the DH result of $ssk_s$ and $epk_c$, and (iii) update the StrobeState object in the server's HandshakeState. The server produces its handshake message by invoking `Disco_WriteMessage()` as well. In addition to generating the server's ephemeral key pair and computing the DH of $esk_s$ and $epk_c$, the `Disco_WriteMessage()` function also outputs shared session keys in two new StrobeState objects. The first object is named *s_write* and can be used

to encrypt all messages that the server sends to the client, whereas the second object, *s_read*, enables the server to decrypt messages of the client. Finally, the server sends the generated (second) handshake message to the client, who upon reception invokes `Disco_ReadMessage()` to retrieve the shared session keys in two new StrobeState objects, called *c_read* and *c_write*, which can be used in the same way as on the server side.

## 4 Experimental Results

We compiled the source code and evaluated the performance of both IoTDisco and EmbeddedDisco using version 7.21.1 of IAR Embedded Workbench for the MSP430 architecture. This development environment includes a cycle-accurate Instruction Set Simulator (ISS) and also provides some utilities for tracking the stack usage during debugging, enabling developers to easily measure execution time and RAM footprint. The binary code size of all modules of an application is reported in a map file after compilation of the source code. In this section, we present, analyze, and compare implementation results of different components of IoTDisco and EmbeddedDisco, which we determined with IAR Embedded Workbench using the TI MSP430F2617 [30] as target device. From a software-architectural point of view, the Disco framework can be (roughly) divided into three layers; the bottom layer includes the main functions of the cryptographic primitives, i.e., fixed/variable-base scalar multiplication on Curve25519 and the function to permute the state of Keccak or Xoodoo. The medium layer covers all Disco functions for the different handshakes (i.e., Noise) and for the secure transport of application data (i.e., Strobe). Finally, the top layer consists of the full handshakes supported by Disco, but we limit our attention to Noise-NK in this paper. We compare IoTDisco and EmbeddedDisco layer-wise from bottom to top and, thereafter, we also compare IoTDisco with implementations of two other E2E security protocols for the IoT.

**Table 2.** Execution time (in clock cycles on a MSP430F2617), throughput (in cycles per rate-byte), RAM usage, and code size of Keccak-$f$[1600] and Xoodoo.

| Permutation | Lang. | Exec. time (cycles) | Throughput (c/rb) | RAM (bytes) | Size (bytes) |
|---|---|---|---|---|---|
| Keccak (24 rounds) | C | 577,808 | 3481 | 536 | 2174 |
| Xoodoo (12 rounds) | Asm | 10,378 | 692 | 262 | 1312 |

Table 2 shows the results of the Keccak permutation used in the EmbeddedDisco software (written in C) and our Assembly implementation of the Xoodoo permutation. Since the permutations use different rates, it makes more sense to compare the throughputs (e.g., in cycles per rate-byte) than the raw execution times. As already mentioned in previous sections, the original Strobe protocol of EmbeddedDisco dedicates two rate-bytes for a special purpose, which means

**Table 3.** Execution time (in clock cycles on a MSP430F2617), RAM usage, and code size of some implementations of scalar multiplication on Curve25519.

| Implementation | Lang. | Exec. time (cycles) | RAM (bytes) | Size (bytes) |
|---|---|---|---|---|
| TweetNaCl [5] | C | 221,219,800 | 2014 | 2510 |
| Düll et al. [10] | Asm | 7,933,296 | 384 | 13,112 |
| This work (fixed-base) | Asm | 4,428,920 | 588 | 5260 |
| This work (variable-base) | Asm | 10,843,907 | 562 | 4717 |

the actual rate (i.e., the length of a Strobe block) is $(1600 - 256)/8 - 2 = 166$ bytes. On the other hand, IoTDisco uses *Strobe Lite* and, hence, only one byte in the rate-part is reserved, i.e., the rate is $(384 - 256)/8 - 1 = 15$ bytes. The throughput figures obtained on basis of these rate values indicate that Xoodoo outperforms Keccak by a factor of five. In addition, the RAM usage and code size of Xoodoo is much smaller.

Table 3 shows the results of some implementations of scalar multiplication on Curve25519 executed on a MSP430(X) microcontroller. Besides TweetNaCl (used by EmbeddedDisco) and our implementation, we also list the currently-fastest software of Curve25519 for MSP430(X), which was introduced by Düll et al. [10]. However, their field-arithmetic operations are aggressively optimized for speed (e.g., by fully unrolling inner loops), thereby inflating the binary code size. For example, the field-arithmetic library alone occupies some 10 kB of the flash memory, which is quite a lot for typical C1 devices. Our ECC software is optimized to achieve a trade-off between speed and code size instead of speed alone; therefore, we we did not unroll the inner loop(s) of performance-critical operations. This makes our implementation slower, but also much smaller than that of Düll et al. More concretely, when compared to Düll et al., our variable-base scalar multiplication is requires 2.9 million cycles more, but the fixed-base scalar multiplication 3.6 million cycles less than their software, which supports only variable-base scalar multiplication. However, when comparing binary code size, our implementation is around 2.6 times smaller. Note that our fixed-base and variable-base scalar multiplication share a lot of the low-level components (e.g., the field arithmetic); thus, the overall size of both is only 6650 bytes.

We summarize in Table 4 the execution time of some of the Disco functions needed for a Noise-NK handshake and for the secure transport of application data. The `disco_WriteMessage()` function performs a fixed-base scalar multiplication (to generate an ephemeral key-pair) and, thereafter, a variable-base scalar multiplication. On the other hand, the `disco_WriteMessage()` function includes just the latter. IotDisco outperforms EmbeddedDisco by more than an order of magnitude (up to a factor of almost 30), which is not surprising since TweetNaCl is not optimized at all for MSP430 and, therefore, quite slow. The `disco_EncryptInPlace()` function executes an authenticated encryption of 65 bytes of application data. Its execution time can be seen as benchmark for the efficiency of the Strobe implementation and its permutation.

**Table 4.** Execution time (in clock cycles on a MSP430F2617) of EmbeddedDisco and IoTDisco when executing the Disco functions for a Noise-NK handshake and for the secure transport of 65 bytes of application data using Strobe (resp., Strobe Lite).

| Disco function | EmbeddedDisco (C impl.) | IoTDisco (Asm impl.) |
|---|---|---|
| Initialization | | |
| `disco_Initialize()` | 583,052 | 55,699 |
| client → server handshake (Noise) | | |
| `disco_WriteMessage()` | 443,604,514 | 15,218,721 |
| `disco_ReadMessage()` | 222,379,982 | 10,825,192 |
| server → client handshake (Noise) | | |
| `disco_WriteMessage()` | 444,768,307 | 15,299,704 |
| `disco_ReadMessage()` | 223,543,472 | 10,903,793 |
| client ↔ server secure transport (Strobe) | | |
| `disco_EncryptInPlace()` | 1,158,706 | 75,774 |
| `disco_DecryptInPlace()` | 1,158,677 | 75,745 |

**Table 5.** Execution time (in clock cycles on a MSP430F2617) of EmbeddedDisco and IoTDisco when executing the a full Noise-NK handshake.

| Implementation | Lang. | Side | Exec. time (cycles) | RAM (bytes) | Size (bytes) |
|---|---|---|---|---|---|
| EmbeddedDisco | C | client | 667,731,038 | 3366 | 8911 |
| | | server | 667,731,341 | 3366 | 8911 |
| IoTDisco | Asm | client | 26,178,213 | 1382 | 11,602 |
| | | server | 26,180,595 | 1382 | 11,602 |

The running time, RAM consumption, and binary code size of a full Noise-NK handshake computation performed by EmbeddedDisco and IoTDisco are shown in Table 5. On each side (i.e., client and server), a Noise-NK handshake invokes the three functions `disco_Initialize()`, `disco_WriteMessage()`, and `disco_ReadMessage()` to obtain a shared secret. IoTDisco requires 26.2 million cycles for a full NK handshake on each the client and the server side, which is more than 25 times faster than EmbeddedDisco. Furthermore, IoTDisco is also much more efficient than EmbeddedDisco in terms of RAM footprint (1383 vs 3366 bytes). A part of this saving comes from the smaller state of the Xoodoo permutation. The RAM footprint given in Table 5 also includes two 128-byte buffers (for sending and receiving messages) on each side. Note that the 1.4 kB of RAM consumed by IoTDisco represents only about 14% of the overall RAM available on a typical C1 device, which leaves about 86% of the RAM for the operating system and the actual target application. The code size of IoTDisco amounts to 11.6 kB, which is 2.7 kB higher than that of EmbeddedDisco. This means IoTDisco occupies less than 12% of the total flash memory available on a typical C1 device.

**Table 6.** Handshake computation time (in clock cycles) of implementations of E2E security protocols for the IoT.

| Protocol | Sec. (bits) | Device | Side | Exec. time (cycles) | RAM (bytes) | Size (bytes) |
|---|---|---|---|---|---|---|
| HIP DEX [24] | 112 | 32-bit ARM9 @180 MHz | client | 192,960,000 | n/a | n/a |
| | | | server | 192,960,000 | n/a | n/a |
| $\mu$EDHOC [17] | 128 | 32-bit Cortex-M0 @16 MHz | client | 274,816,000 | 2381 | 18,950 |
| | | | server | 274,832,000 | 2624 | 18,950 |
| IoTDisco (This work) | 128 | 16-bit MSP430X @8 MHz | client | 26,178,213 | 1382 | 11,602 |
| | | | server | 26,180,595 | 1382 | 11,602 |

Finally, in Table 6 we compare IoTDisco with implementations of the two other E2E protocols mentioned in Sect. 1, i.e., the HIP DEX protocol by Nie et al. [24] and $\mu$EDHOC by Hristozov et al. [17]. Even though a 32-bit ARM9 microcontroller has more computing power than a 16-bit MSP430, IoTDisco is about 7.4 times faster than HIP DEX. It should also be noted that DEX was designed to offer only up to 112-bit security. When compared to $\mu$EDHOC on an ARM Cortex-M0, IoTDisco is more than an order of magnitude faster and also consumes 1.0 kB less RAM and 7.3 kB less flash memory.

## 5 Summary and Conclusion

Although E2E-secure communication is nowadays omnipresent in the classical Internet, it still represents a massive challenge for the IoT due to the resource constraints of the connected devices. We presented in this paper an optimized implementation and practical evaluation of Disco, a modern E2E security protocol combining Noise (a DH-based two-party handshake protocol) and Strobe (a permutation-based secure transport protocol). Disco is a "clean-slate" design and, therefore, unencumbered by most of the problems and issues that plague legacy protocols such as TLS, in particular backwards compatibility, algorithm agility, and/or inefficient cryptographic primitives. The IoTDisco prototype we introduced is optimized for MSP430-based C1 devices and contains carefully-tuned Assembly functions for the prime-field arithmetic of Curve25519 and the Xoodoo permutation, which serves as lightweight replacement for Keccak. Due to these optimizations, IoTDisco is capable to complete the full computational part of a Noise NK handshake in only 26.2 million cycles on our target device (a TI MSP430F2617 microcontroller), which compares very favorably with the implementations of other E2E protocols described in the literature. IoTDisco occupies only about 11.6 kB flash memory and roughly 1.4 kB RAM, which is less than 14% of the total flash capacity and less than 17% of the RAM of the MSP430F2617. All these results make IoTDisco an important milestone on the road towards strong E2E security in the Internet of constrained things, i.e., the Internet of C1 devices.

## References


1. AlFardan, N.J., Bernstein, D.J., Paterson, K.G., Poettering, B., Schuldt, J.C.: On the security of RC4 in TLS. In: King, S.T. (ed.) Proceedings of the 22th USENIX Security Symposium (USS 2013). pp. 305–320. USENIX Association (2013)
2. Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) Public Key Cryptography — PKC 2006. Lecture Notes in Computer Science, vol. 3958, pp. 207–228. Springer (2006)
3. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards curves. In: Vaudenay, S. (ed.) Progress in Cryptology — AFRICACRYPT 2008. Lecture Notes in Computer Science, vol. 5023, pp. 389–405. Springer (2008)
4. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering **2**(2), 77–89 (Sep 2012)
5. Bernstein, D.J., van Gastel, B., Janssen, W., Lange, T., Schwabe, P., Smetsers, S.: TweetNaCl: A crypto library in 100 tweets. In: Aranha, D.F., Menezes, A. (eds.) Progress in Cryptology — LATINCRYPT 2014. Lecture Notes in Computer Science, vol. 8895, pp. 320–337. Springer (2015)
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Keccak reference, version 3.0. Available for download at `http://keccak.team/files/Keccak-reference-3.0.pdf` (2011)
7. Bormann, C., Ersue, M., Keranen, A.: Terminology for Constrained-Node Networks. IETF, Light-Weight Implementation Guidance Working Group, RFC 7228 (May 2014)
8. Costello, C., Smith, B.: Montgomery curves and their arithmetic. Journal of Cryptographic Engineering **8**(3), 227–240 (Sep 2018)
9. Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., Van Keer, R.: Xoodyak, a lightweight cryptographic scheme. IACR Transactions on Symmetric Cryptology **2020**(S1), 60–87 (Jun 2020)
10. Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed Curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. Designs, Codes and Cryptography **77**(2–3), 493–514 (Dec 2015)
11. Durumeric, Z., Kasten, J., Adrian, D., Halderman, J.A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., Paxson, V.: The matter of Heartbleed. In: Williamson, C., Akella, A., Taft, N. (eds.) Proceedings of the 14th Internet Measurement Conference (IMC 2014). pp. 475–488. ACM (2014)
12. Giesen, F., Kohlar, F., Stebila, D.: On the security of TLS renegotiation. In: Sadeghi, A., Gligor, V.D., Yung, M. (eds.) Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS 2013). pp. 387–398. ACM (2013)
13. Guha Sarkar, P., Fitzgerald, S.: Attacks on SSL: A comprehensive study of BEAST, CRIME, TIME, BREACH, Lucky 13 & RC4 biases. Tech. rep., iSEC Partners Inc. (Part of NCC Group) (2013), available for download at `http://www.nccgroup.com/globalassets/our-research/us/whitepapers/ssl_attacks_survey.pdf`
14. Hamburg, M.: The STROBE protocol framework. Cryptology ePrint Archive, Report 2017/003 (2017), available for download at `http://eprint.iacr.org`
15. Hankerson, D.R., Menezes, A.J., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer (2004)
16. Hişil, H., Wong, K.K.H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: Pieprzyk, J. (ed.) Advances in Cryptology — ASIACRYPT 2008. Lecture Notes in Computer Science, vol. 5350, pp. 326–343. Springer (2008)

17. Hristozov, S., Huber, M., Xu, L., Fietz, J., Liess, M., Sigl, G.: The cost of OSCORE and EDHOC for constrained devices. In: Joshi, A., Carminati, B., Verma, R.M. (eds.) Proceedings of the 11th ACM Conference on Data and Application Security and Privacy (CODASPY 2021). pp. 245–250. ACM (2021)
18. Krawczyk, H.: SIGMA: The 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE-protocols. In: Boneh, D. (ed.) Advances in Cryptology — CRYPTO 2003. Lecture Notes in Computer Science, vol. 2729, pp. 400–425. Springer (2003)
19. LaMacchia, B.A., Lauter, K.E., Mityagin, A.: Stronger security of authenticated key exchange. In: Susilo, W., Liu, J.K., Mu, Y. (eds.) Provable Security — ProvSec 2007. Lecture Notes in Computer Science, vol. 4784, pp. 1–16. Springer (2007)
20. Law, L., Menezes, A., Qu, M., Solinas, J.A., Vanstone, S.A.: An efficient protocol for authenticated key agreement. Designs, Codes and Cryptography **28**(2), 119–134 (Mar 2003)
21. Liu, Z., Großschädl, J., Li, L., Xu, Q.: Energy-efficient elliptic curve cryptography for MSP430-based wireless sensor nodes. In: Liu, J.K., Steinfeld, R. (eds.) Information Security and Privacy — ACISP 2016. Lecture Notes in Computer Science, vol. 9722, pp. 94–112. Springer (2016)
22. Menezes, A.J., Stebila, D.: End-to-end security: When do we have it? IEEE Security & Privacy **19**(4), 60–64 (Jul 2021)
23. Moskowitz, R., Hummen, R., Komu, M.: HIP Diet EXchange (DEX). IETF, Internet draft draft-ietf-hip-dex-24 (Jan 2021)
24. Nie, P., Vähä-Herttua, J., Aura, T., Gurtov, A.V.: Performance analysis of HIP Diet Exchange for WSN security establishment. In: Chen, H., Ben-Othman, J., Cesana, M. (eds.) Proceedings of the 7th ACM Symposium on QoS and Security for Wireless and Mobile Networks (Q2SWinet 2011). pp. 51–56. ACM (2011)
25. Perrin, T.: The Noise protocol framework (revision 34). Specification, available for download at `http://noiseprotocol.org/noise.pdf` (2018)
26. Rescorla, E.K.: The Transport Layer Security (TLS) Protocol Version 1.3. IETF, Network Working Group, RFC 8446 (Aug 2018)
27. Restuccia, G., Tschofenig, H., Baccelli, E.: Low-power IoT communication security: On the performance of DTLS and TLS 1.3. In: Proceedings of the 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN 2020). pp. 1–6. IEEE (2020)
28. Selander, G., Preuß Mattsson, J., Palombini, F.: Ephemeral Diffie-Hellman Over COSE (EDHOC). IETF, Internet draft draft-ietf-lake-edhoc-22 (Aug 2023)
29. Stallings, W.: Cryptography and Network Security: Principles and Practice. Pearson, 7th edn. (2016)
30. Texas Instruments, Inc.: MSP430x2xx Family User's Guide (Rev. J). Manual, available for download at `http://www.ti.com/lit/ug/slau144j/slau144j.pdf` (2013)
31. The OpenSSL Project: OpenSSL: Cryptography and SSL/TLS Toolkit. Available online at `http://www.openssl.org` (2021)
32. WhatsApp LLC: WhatsApp encryption overview. Technical white paper, available for download at `http://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf` (2020)
33. Wong, D.: Noise extension: Disco (Revision 6). Specification, available for download at `http://www.discocrypto.com/disco.pdf` (2018)
34. Wong, D.: Disco: Modern session encryption. Cryptology ePrint Archive, Report 2019/180 (2019), available for download at `http://eprint.iacr.org`
35. Wong, D.: EmbeddedDisco (2020), available online at `http://embeddeddisco.com`