The STROBE protocol framework

Mike Hamburg*

Abstract

The "Internet of Things" (IoT) promises ubiquitous, cheap, connected devices. Unfortunately, most of these devices are hastily developed and will never receive code updates. Part of the IoT's security problem is cryptographic, but established cryptographic solutions seem too heavy or too inflexible to adapt to new use cases.

Here we describe STROBE, a new lightweight framework for building both cryptographic primitives and network protocols. STROBE is a sponge construction in the same family as Markku Saarinen's BLINKER framework.

The STROBE framework is simple and extensible. It is suitable for use as a hash, authenticated cipher, pseudorandom generator, and as the symmetric component of a network protocol engine. With an elliptic curve or other group primitive, it also provides a flexible Schnorr signature variant.

STROBE can be instantiated with different sponge functions for different purposes. We show how to instantiate STROBE as an instance of NIST's draft cSHAKE algorithm. We also show a lightweight implementation which is especially suitable for 16- and 32-bit microcontrollers, and also for small but high-speed hardware.

1 Introduction

It is famously difficult for software developers to design and deploy secure protocols, even if they have some familiarity with cryptography [21]. Expert analysis is needed to determine what information to encrypt, sign and hash, and with what keys. The usual guidance is that non-cryptographers should not design or implement protocols at all. If they choose to do it anyway, most existing protocols offer little guidance for secure protocol design.

The most common approach is simply to use TLS. But while TLS suffices for web traffic, some applications do not share its constraints. Limited devices might simply not have the resources to perform TLS. Parties may have the wrong sorts of authenticators, such as a symmetric key on one side and a certificate on the other. Designers may want to use

*Rambus Cryptography Research.

a mixture of classical and post-quantum cryptosystems, or they might want more or less repudiability than TLS offers. To accomplish these goals, developers can either build a custom protocol, or realize one from the academic literature. The literature route is easier, but not by much. Academic protocols are sometimes specified at a high level, and it is not obvious how to securely hash, e.g., an identity with a pair of group elements.

To assist the developers of these custom systems, several authors have recently published frameworks such as NaCl [3], Noise [16], and BLINKER [19]. Such frameworks aim to reduce the work for developers and limit their opportunities to make mistakes. Here we propose the STROBE protocol framework in the same vein as BLINKER. STROBE is intended to be more flexible and extensible than previous work, while still having only "one well-oiled joint" [15].

STROBE's design is based on both academic and industrial concerns. We hope that it will facilitate academic cryptography in addition to industrial protocols and implementations.

1.1 Related work

Cryptographic sponge constructions [8] aim to build many different symmetric cryptosystems from a single "random-looking" permutation or function. They have seen considerable attention due to the selection of the KECCAK sponge construction [4] for SHA-3 [1]. But despite the flexibility of sponges, most sponge constructions are aimed at one particular task.

A notable exception is Saarinen's BLINKER [19] framework, which provides support for many different cipher modes and protocols. In order to describe what operation is being done, BLINKER reserves a portion of the sponge's capacity for a "pad" word containing metadata. This word describes the meaning to the protocol of the operation being performed.

The main innovation in STROBE compared to BLINKER is that the cryptography of an operation depends primarily on that operation's data flow. Therefore STROBE's metadata is divided. At a low level, the padding is mainly an indication of data flow. This padding has a very strict format, and corresponds directly to how the operation is performed. A second level of metadata describes what a given operation means to the protocol. This metadata is free-form and has arbitrary length, and uses the same mechanisms as other messages. It can either be implicit, or sent to the transport as framing information. If it is used as framing, metadata may be sent either encrypted or in the clear. This technique gives greater flexibility to applications. Its performance is generally better than that of BLINKER, and the complexity is not much greater.

STROBE also improves BLINKER's ordering of metadata and data. BLINKER's pad word is input along with the data, so it only affects later operations and later blocks of the same operation. For example, if one party encrypts a zero message and the other party performs a MAC, the outputs will be the same, but future outputs will differ. STROBE's padding is entered before the data, so that a MAC is always different from an encryption.

1.2 Outline of this work

In Section 3, we give a high-level overview of the STROBE framework. In Section 4, we describe its internals. In Section 5, we describe how to use it to implement various protocols. In Section B, we recommend particular choices of sponge functions and parameters. In Section C, we describe a few extensions. In Section 6, we give implementation results.

2 Definitions and notation

A bit is an element of $\{0, 1\}$. A byte is an element of $\mathcal{B} := [0, 2^8)$. A string is an element of $\mathcal{B}^*$, meaning a sequence of bytes. All keys, messages, and other data in STROBE are strings.

Because STROBE is byte-oriented, endianness usually does not matter to us. When it does matter, we use a little-endian convention throughout this work, meaning that bit 0 is the least-significant bit of a multi-bit sequence, and likewise for bytes and larger types. More formally, let b and n be positive integers. When an element $[[X_i : i \in [0, n)]]$ of $[0, b)^n$ is converted to an element of $[0, b^n)$, its converted value $\sum_{i=0}^{n-1} b^i X_i$.

The notation $A||B$ means the concatenation of sequences A and B. The notation $a \oplus b$ means the exclusive-or of a and b.

"MAC" stands for "message authentication code". "AEAD" stands for "authenticated encryption with associated data". "PRF" stands for "pseudorandom function". "PRNG" stands for "pseudorandom number generator".

3 Strobe framework

3.1 Overview and limitations

STROBE is a framework for building cryptographic two-party protocols. It can also be used for symmetric cryptosystems such as hashing, AEAD, MACs, PRFs and PRNGs. It is also useful as the symmetric part of a Schnorr-style signature scheme. For brevity we will describe all of these as *protocols* in this work. Our protocols either have only one party, or they have two parties connected by a *transport* layer. The transport layer may be insecure: an adversary may be able to see and modify any data which is sent there. We will model a symmetric cryptosystem such as AEAD as a protocol in which one party encrypts and sends a message to a recipient using the transport, but the recipient never replies to the sender. Here the transport might be a storage medium rather than a network, but the security model is almost equivalent. We describe hashes, PRFs and PRNGs as one-party "protocols" with no transport.

STROBE's main design principle is that the cryptographic output from any step depends not only on any keys, but also on all preceding inputs. For example, when encrypting a message, the ciphertext depends on the key and plaintext. But it also depends on all preceding data, such as nonces and associated data. STROBE achieves this by keeping a running hash of the protocol transcript. This design choice optimizes for simplicity and security instead of speed.

Because of this running hash, STROBE is primarily suited to transports that reliably deliver messages in order. If a message is corrupted or lost, it will detect this and abort the protocol, but the session cannot be recovered. Furthermore, STROBE is suited to protocols that are *half-duplex*, meaning that the parties take turns sending messages instead of both sending at once. Many secure channel protocols, such as TLS, are half-duplex for the handshake phase of the protocol, and then become full-duplex once the channel is open. This is usually achieved by extracting two separate keys from the handshake: one for messages from Alice to Bob, and one from Bob to Alice. These individual channels then operate individually in half-duplex mode. We take the same strategy with STROBE: we focus our attention on half-duplex protocols, and discuss full-duplex ones in Appendix C.1.

Some protocols (e.g. DTLS) are additionally designed to tolerate message loss and out-of-order message delivery. STROBE is not a complete framework to build these protocols, but it can still be used in them as a symmetric encryption, hashing and MAC algorithm.

STROBE performs only the symmetric part of a protocol. Public-key encryption and signatures still need another primitive, such as RSA or elliptic curves. STROBE can make it easier to integrate these, as shown for Schnorr signatures in Section 5.3.

3.2 Application and transport

The STROBE framework is situated conceptually between two domains: the *application* and the *transport*. The application domain holds data which will be used directly: keys, plaintext messages, nonces and associated data. We want to protect the confidentiality and integrity of this data. The transport domain holds data to be transmitted between parties, or data to be stored and retrieved from untrusted memory. This domain includes ciphertexts, messages transmitted in cleartext, MACs and signatures. The application never accesses the transport directly. Instead, it reads and writes all data through STROBE operations such as encryption and decryption.

STROBE's operations move data to and/or from the application and/or transport. For example, an encryption operation takes some amount of data from the application, encrypts it, and sends the ciphertext to the transport.

In general, the threat model for a STROBE protocol assumes that the transport is entirely under adversarial control. The application will be largely under the control of an honest

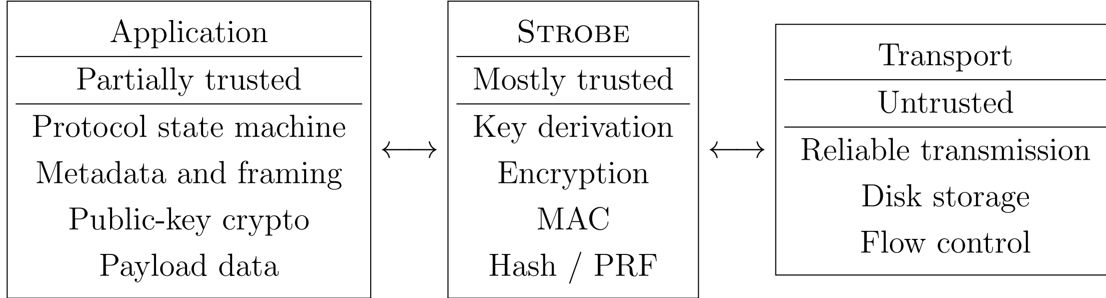

Figure 1: Strobe division of responsibilities

party. However, the adversary might be able to influence that party's behavior, or even compromise it entirely. In a typical threat model, the Strobe layer itself might have its state stolen by an application compromise, but the adversary cannot otherwise modify its behavior (e.g. through a fault attack).

3.3 Overview of supported operations

Strobe supports several classes of operations, summarized in Table 1 along with abbreviations. They are categorized by four important pieces of information, called *flags*:

- I: Does the operation move data in the "inbound" direction, i.e. transport $\rightarrow$ cipher $\rightarrow$ application, or in the opposite "outbound" direction?
- A: Does the operation send/receive data belonging to the application, or not? For example, sending an encrypted message uses application data, but a MAC does not.
- C: Does the operation set a key or use the cipher's output, or not? For example, encrypting a message interacts with the cipher, but sending it in cleartext does not. Operations with the C flag either output the cipher's data; or if they have no output, rekey the cipher.
- T: Does the operation send/receive data via the transport, or not? For example, a secret key does is not sent to the transport, but a cleartext message is.

A main innovation of Strobe is that the behavior of each operation follows in a straightforward manner from these four features.

Each operation is performed in an online, streaming manner. It can take data of unbounded size, if indeed it uses data at all. The size of the data need not be known beforehand. Any number of operations may be performed in any order. Implementations might

Abbr.	Operation	Flags	Application	STROBE	Transport
KEY	Secret key	AC	□ ——→	Ⓚ	
AD	Associated data	A	□ ——→	○	
PRF	Hash / PRF	IAC	□ ←——	⊕	—— 0
CLR	Send cleartext data	$A\ \ T$	□ ——	○	——→ □
recv-CLR	Receive cleartext data	$IA\ \ T$	□ ←——	○	—— □
ENC	Encrypt	ACT	□ ——	⊕	——→ □
recv-ENC	Decrypt	$IACT$	□ ←——	⊕	—— □
MAC	Compute MAC	CT	0 ——	⊕	——→ □
recv-ENC	Verify MAC	$I\ \ CT$	0 ←—	⊕	—— □
RATCHET	Rekey to prevent rollback	C	0 —→	Ⓚ	

Legend: □ Send/recv ○ Absorb into sponge ⊕ Xor with cipher Ⓚ Roll key

Table 1: STROBE operations and their data flow.

limit allowable orders for sanity-checking reasons, such as only allowing encryption operations after a key has been entered. Nevertheless, the operations' semantics are well-defined regardless of order.

The RATCHET operation doesn't have input or output. Instead, it is designed to prevent rollback attacks. Suppose an attacker recovers the sponge's state at the end of the protocol, perhaps through an application exploit. She can reverse the STROBE steps to learn earlier states and decrypt earlier messages. This is analogous to recovering the key of a symmetric cipher. The RATCHET operation prevents this by erasing part of the state. In the tiniest STROBE instances, the state is too small to erase part of it, and this operation must be done differently. RATCHET. This is useful to derive keys from passwords on devices which are too small to run a memory-hard function.

The behavior of an operation is determined by the 4 bits (I, A, C, T), but our implementation encodes these in a full byte. This leaves 4 bits for other uses. We will use one of those bits, labeled M, to distinguish metadata as described in Section 3.6. The M bit has no effect on how the operation is performed; it is merely hashed into the protocol transcript. We will use another bit, labeled K, for the DPA-resistant "key tree" extension, as described in Section C.2. Unlike M, the K bit does affect how the operation is performed. The final two bits are reserved for future use.

3.4 Protocol transcripts

STROBE maintains a running hash of the protocol transcript, which is the sequence of all operations and their data as seen by the application layer. This includes operations such as `KEY` and `AD` which do not transmit data to the transport. Formally, the transcript is a list of pairs of the form

$$(\text{AdjDir}(I_0;\ \text{operation}), \text{AppData})$$

The $\text{AdjDir}(I_0;\ \cdot)$ function adjusts the operations so that when Alice sends a message (with $I = 0$) and Bob receives it (with $I = 1$), the two protocol transcripts will match. It is implemented as

$$\text{AdjDir}(I_0,\ (I, A, C, T, M, K)) := (I \oplus (T \cdot I_0), A, C, T, M, K)$$

where I_0 is the value of the I flag in the first operation that has $T = 1$. That is, $I_0 = 0$ for the initiator and $I_0 = 1$ for the responder.

When $A = 1$, the AppData is the value sent from the application if $I = 0$, or returned to the application if $I = 1$. When $A = 0$, then $\text{AppData} := L \times [[0]]$ for an L-byte operation. This means that formally, a protocol transcript includes the length of MACs, but not their values. The protocol will abort when receiving an incorrect MACs.

3.5 Security of Strobe

A formal model of STROBE's security is outside the scope of this paper. Informally, the ith block of output (of a certain block size) from a STROBE operation $op \in \{\texttt{ENC}, \texttt{MAC}, \texttt{PRF}\}$ is function of op; the preceding protocol transcript Tr; that block of input; and the previous blocks of application data for that block. Specifically,

$$\text{Output}_i = \text{Input}_i \oplus G(\text{Tr},\ \text{AdjDir}(I_0, \text{operation}),\ (\text{AppData}_1 \ ||\ \ldots\ ||\ \text{AppData}_{i-1}))$$

In the random function model, this G is ϵ-indifferentiable from a random function after N total queries to the sponge primitive, where $\epsilon \leq (N/2^{c/2})^2$. This follows directly from the lemmas in [6] and the fact that STROBE's padding is sponge-compliant. Setting $c = 2\lambda$ is slightly stronger than a traditional cipher with a λ-bit key. An adversary must put in about 2^λ work to break either system. But the adversary is less likely to break STROBE by luck with much less than 2^λ work, and does not benefit as much from extra data.

This means that an adversary cannot distinguish STROBE's output from random with probability greater than ϵ (and cannot forge a t-bit `MAC` with probability greater than $2^{-t} + \epsilon$) unless one of the following holds:

- The adversary has guessed or calculated the transcript Tr, including all keys entered into it up to this point.

- Some other party has queried $G(\text{Tr}, \text{AdjDir}(I_0, \text{operation}), m')$ for some m' and given the result to the adversary. The adversary would then learn $m \oplus m'$.
- The adversary has compromised the state of the Strobe object through some kind of side channel.

3.6 Metadata

The security properties of many practical protocols are also stated in terms of protocol transcripts. This makes it straightforward to evaluate those properties for a protocol P built on Strobe. But to do this, we must make sure that Strobe's transcript can be parsed into a transcript for P.

To begin with, all Strobe protocols must begin with an `AD` operation containing a domain separation string. The string should be a URI that uniquely describes the protocol. This prevents most classes of cross-protocol attacks.

For a simple protocol, each operation might have only one possible meaning in context, and so the transcript would be parseable. For example, an AEAD encryption scheme might just use the domain separator and then `KEY`, `AD`, `ENC` and `MAC` in that order. The meaning of each of these operations is clear, so long as the length of the MAC is fixed.

A more complex protocol might have many different meanings for the same operation. A natural pattern to ensure parseability is to precede each operation with a comment in the transcript that disambiguates it. Such information is usually provided anyway though protocol framing. That is, when Alice sends a message within (say) a TCP stream, Bob must be able to determine the kind of message and the length. In order to provide this, usual Alice will send a *tag* to indicate the message type, followed by a *length* field, and finally a *value*. This is called tag-length-value framing.

We support this pattern through *metadata* operations, which are no different from ordinary operations except that they have the M flag set. The M flag has no effect on the operation, except that it is included in the transcript. We describe such an operation as `meta-`*op*. Most protocols send framing data in the clear. For this we would use a `meta-CLR` transaction, which exactly like `CLR` except that it has the M flag set in the transcript. If a `meta-ENC` operation is used instead, then the framing information will be encrypted. This may be useful for steganography or length padding. If a protocol does not transport enough framing information to be completely parseable, it can supply this information through a `meta-AD` operation.

We recommend that each non-metadata operation should be preceded by a metadata operation that describes it. Let `OP` be some operation, and `info` be a tag that defines the meaning of the message to the protocol. The `info` also includes the message's length, unless

this is either unknown or fixed by the protocol. We define a *tagged operation* $\texttt{OP}[\texttt{info}](msg)$ as follows. First the protocol processes the info with a `meta-CLR`, `meta-AD` or `meta-CLR` operation as appropriate for framing, and then it processes $\texttt{OP}(msg)$.

For example, $\texttt{ENC}[\texttt{app-ciphertext}]$("hello") might mean the two operations

$$\texttt{meta-CLR}([[\texttt{0x03}, \texttt{0x05}, \texttt{0x00}]]); \quad \texttt{ENC}(\text{"hello"})$$

This frames the message with a tag `0x03` representing `app-ciphertext` and a length [[`0x05`, `0x00`]] representing five bytes. The format of the tag and length is protocol-specific. So is how the framing is transmitted: in the clear (`meta-CLR`), encrypted (`meta-ENC`) or not at all (`meta-AD`).

The length of a Strobe operation does not need to be known ahead of time. This is especially useful when receiving framing information, since different message types may have different amounts of extra information. Of course, if the message is to be preceded by its length, then the length does need to be known first.

4 Implementation

Strobe is based on the duplexed sponge construction [6]. Its implementation is therefore primarily a matter how the data is padded, and in what order it is absorbed into or squeezed out of the sponge.

4.1 Beginning an operation

The most interesting part of Strobe's padding is how it begins an operation. Because Strobe uses small operations for metadata, it is inefficient to run F for each operation unless the rate of the sponge is small. Instead Strobe tracks the position of the beginning of the operation in an extra state variable p_{begin}. Before running F, it absorbs p_{begin} into the sponge and then resets $p_{\text{begin}} \leftarrow 0$. This ensures that p_{begin} need only be tracked within a block.

To begin an operation with operation (I, A, C, T, M, K), we perform these steps:

- If this is the first send or receive operation – that is, if $(T, I_0) = (1, \perp)$ – then set this party's role $I_0 \leftarrow I$.
- Absorb p_{begin}. This increments the position p. Then update $p_{\text{begin}} \leftarrow p$.
- Absorb $\text{AdjDir}(I_0, \text{operation})$ to disambiguate the operation being performed.

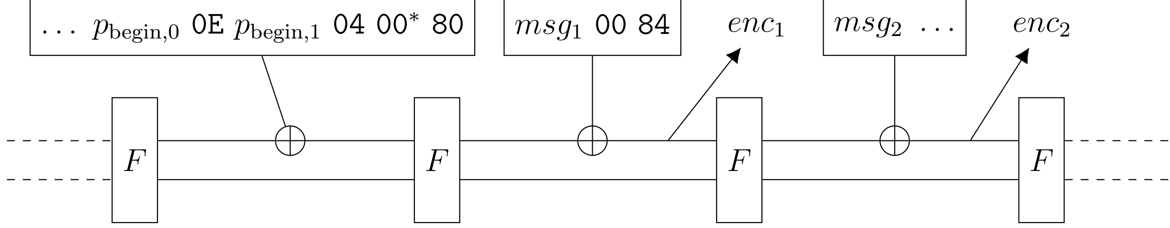

Figure 2: STROBE padding for a 2-block ENC operation sent by the initiator, encrypting $msg_1||msg_2$ into $enc_1||enc_2$. The numeric constants are written in hexadecimal notation.

- If the C or K flag is set, end the block by padding it and running F. This ensures all previous input will affect the next block. The padding uses the new p_{begin}.[1]

4.2 Implementation of operations

The operation itself is implemented using a duplex construction [6]. The full details are in Appendix A.

An `ENC` operation is illustrated in Figure 2. The operation absorbs first p_{begin} and the operation byte AdjDir(I_0; `ENC`), which is `0E` if it is being sent by the initiator or `0F` if it is sent by the responder. It updates p_{begin} to the position of the operation byte. Then the block is ended with the STROBE padding – the new p_{begin} – and the cSHAKE padding `04 00`* `80`. Next comes the first block of the message. Once the rate is reached, the sponge absorbs p_{begin} = `00` and the cSHAKE padding `84`.

After second block of the message is encrypted, the sponge function F isn't yet run. Instead, the next operation will continue in that block.

5 Applications

5.1 Recommended metadata

Here we present some example protocols. These protocols are defined more completely at `https://strobe.sourceforge.io/examples/`. These protocols can be composed. The signature protocol, for example, can be used as part of a handshake instead of as a standalone signature.

[1] Lightweight sponges such as Ketje [5] may have non-uniform round counts. Ending the block this way should use the higher round count, and ending it by overrunning the rate should use the lower one.

5.2 Authenticated encryption

In an AEAD scheme, Alice shares a symmetric key K with Bob. She wants to send him an encrypted message M. She may also want to send some unencrypted data D. She wants to authenticate this message with a MAC, perhaps along with some associated data A that will not be transmitted. For semantic security, the parties must use a nonce N, which must be unique among all AEAD messages with the same key.

With STROBE, it is easy to construct an AEAD scheme. An appropriate order is:

$$\texttt{KEY}[\texttt{sym-key}](K);\;\; \texttt{AD}[\texttt{nonce}](N);\;\; \texttt{AD}[\texttt{auth-data}](A);\;\; \texttt{CLR}[\texttt{auth-data}](D);$$
$$\texttt{ENC}[\texttt{app-ciphertext}](M);\;\; \texttt{MAC}[\texttt{MAC}](128 \text{ bits});$$

The operation `AD[nonce]`(N) can be replaced with `CLR[nonce]`(N) to transmit the nonce.

This construction relies on a unique combination of the key, nonce and authenticated data. In particular, its security may fail if the nonce is not unique. To achieve nonce-misuse resistance, a two-pass scheme such as SIV [17] must be used.

The AEAD construction can be used with any number nonces, authenticated data elements, and ciphertexts. It can be used with no ciphertext for just the MAC. It can be used with no authenticated data by omitting the `auth-data` item. This is arguably more natural than encrypting or authenticating the empty string.

5.3 Schnorr signatures

Schnorr's signature scheme [22] is a simple way to build signatures based on the discrete logarithm problem in a group $\mathbb{G}$. We will assume that $\mathbb{G}$ has an efficient encoding to and from fixed-length sequences of $b_{\mathbb{G}}$ bytes.

The Schnorr scheme can be applied to sign an entire STROBE protocol transcript inline with the session. As in a typical Schnorr signature, fix a base point $g \in \mathbb{G}$ of prime order p. Let b be the number of bytes required to sample $(1/\sqrt{g})$-uniformly from $\mathbb{Z}/p\mathbb{Z}$. Alice has a private key k, from which she derives[2] a secret exponent a by running

$$\texttt{KEY}[\texttt{sym-key}](k);\; a \leftarrow \texttt{PRF}[\texttt{derive-key}](b \text{ bytes})$$

Her public key is then $A := g^a$. When Alice needs to sign a STROBE context, she first begins the signature using a `AD[sig-scheme]`$(name)$ operation. She then needs a pseudorandom value r. She calculates this determinsitically[3] by copying the context and running

$$\texttt{KEY}[\texttt{sym-key}](k);\; r \leftarrow \texttt{PRF}[\texttt{sig-determ}](b \text{ bytes})$$

[2] It is just as valid for a to be the private key, but this may impede a standard-model analysis.

[3] Choosing r randomly also works, but it results in catastrophic failure if the random generator is biased.

Operation	Code size	Global mem	Stack mem
Keccak-f[800]	248 B	0 B	36 B
Strobe tagged operations	792 B	0 B	132 B
Strobe as a PRNG	132 B	120 B	148 B
Curve25519 ECDH	1006 B	0 B	392 B
Schnorr signing with Curve25519	438 B	0 B	480 B
Schnorr verification with Curve25519	248 B	0 B	472 B
Total (incl. padding)	2889 B	120 B	600 B +

Table 2: Strobe implementation results. The total stack memory column includes Strobe functions and one Strobe object, but not the caller's message buffers or the OS networking data.

in the copy. She calculates $R := g^r$ and runs

$$\texttt{AD}[\texttt{pubkey}](A);\ \texttt{CLR}[\texttt{sig-eph}](R);\ c \leftarrow \texttt{PRF}[\texttt{sig-chal}](b \text{ bytes});\\ \texttt{ENC}[\texttt{sig-resp}](r + ac \bmod p)$$

To verify the signature, Bob runs

$$\texttt{AD}[\texttt{pubkey}](A);\ R \leftarrow \texttt{CLR}[\texttt{sig-eph}](b_{\mathbb{G}} \text{ bytes});\ c \leftarrow \texttt{PRF}[\texttt{sig-chal}](b \text{ bytes});\\ s \leftarrow\ \texttt{recv-ENC}[\texttt{sig-resp}](\lceil \log_{256}(p) \rceil \text{ bytes})$$

Bob then checks that $R = g^s/A^c$. This holds because $g^s = g^{r+ac} = R \cdot A^c$.

The usual argument [22] proves that this is secure in the random oracle model.

6 Implementation results

We implemented Strobe-128/800 (a 128-bit security level and the Keccak-F[800] sponge) and Curve25519 [2] for the ARM Cortex-M4 in a mixture of C and assembly. We did not include support for the key-tree bit K. We used assembly for the Keccak round function, for the Curve25519 arithmetic functions, and for a few intrinsics. The library uses callbacks for transport I/O, so that many forms of I/O can be supported. It includes callbacks for memory buffers, but not for sockets because embedded systems may have very different socket environments compared to Linux. The C code is moderately optimized for size. We compiled with `gcc-4.9.3 -Os -DNDEBUG` and estimated stack usage with `-fstack-usage`. Each Strobe context occupies 120 bytes of stack space: 100 B for the sponge data, and 20 B for auxiliary data and callbacks. The results are shown in Table 2.

This demonstrates that an IoT device running Strobe can support pseudorandom generation, symmetric and asymmetric encryption and decryption, signing, verification with only

4kB of memory. This figure also includes framing code, tagged operations and callback-based input/output.

We have not yet tested this with STROBE lite and a lighter-weight cipher. This would save about 10% of the implementation budget, because the block function and duplex mode are simpler and the cipher state is smaller.

6.1 Open-source implementation

An open-source version of the STROBE framework is available at `https://sf.net/p/strobe`. As of this writing, the ARM intrinsics and the DPA-resistant key tree are not enabled in the open-source version.

7 Future work

There is much work yet to be done in the area of lightweight protocol frameworks, and for STROBE in particular. The first order of business is a formal security model, which is likely to be a much longer paper. A related question is how to incorporate and improve recent results on reduced sponge sizes [11]. This may be tricky because not all STROBE protocols will be immediately keyed, and it might require changing the sponge's rate during operation. It may also be helpful to change the encryption formula so that decryption doesn't overwrite the state. For example, we could change $c = m \oplus s$ to $c = m \oplus \alpha s$ for some $\alpha \notin \{0, 1\}$.

We would also like to improve support for protocols where messages may be delivered out of order. A different direction is to improve support for handling protocol state machines, especially in systems which are too small to use threads. It also seems likely that STROBE's padding can be made simpler or more efficient. Finally, we are working to implement STROBE in lightweight hardware, both with KECCAK-f and with a lighter-weight sponge.

Some standards bodies may not be satisfied with an algorithm that instantiates cSHAKE. It should not be difficult to implement STROBE's operations in terms of a hash and a block cipher or PRF, but we will need to formally specify the security model first.

7.1 Acknowledgements

Thanks to Mark Marson and Lydia Hamburg for proofreading this paper.

References

[1] SHA-3 standard: Permutation-based hash and extendable-output functions. 2015. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`.

[2] Daniel Bernstein. Curve25519: new Diffie-Hellman speed records, 2006. https://cr.yp.to/ecdh/curve25519-20060209.pdf.

[3] Daniel Bernstein, Tanja Lange, and Peter Schwabe. NaCl: Networking and cryptography library. https://nacl.cr.yp.to/.

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The KECCAK SHA-3 submission. 2011. http://keccak.noekeon.org/Keccak-submission-3.pdf.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. CAESAR submission: KETJE v2. 2016. https://competitions.cr.yp.to/round3/ketjev2.pdf.

[6] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *International Workshop on Selected Areas in Cryptography*, pages 320–337. Springer, 2011. http://sponge.noekeon.org/SpongeDuplex.pdf.

[7] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the Association for Computing Machinery*, 33:792–807, 1986. http://groups.csail.mit.edu/cis/pubs/shafi/1986-jacm.pdf.

[8] Michaël Peeters Guido Bertoni, Joan Daemen and Gilles Van Assche. Cryptographic sponge functions. http://sponge.noekeon.org/.

[9] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. *Advances in Cryptology–CRYPTO 2011*, pages 222–239, 2011. https://eprint.iacr.org/2011/609.pdf.

[10] Xu Guo, Sinan Huang, Leyla Nazhandali, and Patrick Schaumont. Fair and comprehensive performance evaluation of 14 second round SHA-3 ASIC implementations. In *The Second SHA-3 Candidate Conference*. Citeseer, 2010.

[11] Philipp Jovanovic, Atul Luykx, and Bart Mennink. Beyond $2^{c/2}$ security in sponge-based authenticated encryption modes. Cryptology ePrint Archive, Report 2014/373, 2014. http://eprint.iacr.org/2014/373.

[12] Elif Bilge Kavun and Tolga Yalcin. A lightweight implementation of Keccak hash function for radio-frequency identification applications. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 258–269. Springer, 2010.

[13] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 derived functions. 2016. http://csrc.nist.gov/publications/drafts/800-185/sp800_185_draft.pdf.

[14] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[15] Adam Langley. Cryptographic agility. `https://www.imperialviolet.org/2016/05/16/agility.html`.

[16] Trevor Perrin. Noise protocol framework. `https://noiseprotocol.org/`.

[17] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In *Advances in Cryptology - EUROCRYPT 2006*. Springer, 2006. `https://www.iacr.org/archive/eurocrypt2006/40040377/40040377.pdf`.

[18] Pankaj Rohatgi. Fight side-channel attacks with leakage-resistant protocols, 2011. `http://www.embedded.com/print/4230467`.

[19] Markku-Juhani O. Saarinen. Beyond modes: Building a secure record protocol from a cryptographic sponge permutation. *CT-RSA*, 2014. `http://eprint.iacr.org/2013/772`.

[20] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with s-box optimization. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 239–254. Springer, 2001.

[21] Bruce Schneier. Amateurs produce amateur cryptography. `https://www.schneier.com/blog/archives/2015/05/amateurs_produc.html`.

[22] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

A Implementation details

Here we give a detailed description of Strobe's implementation.

A.1 State of a Strobe object

A Strobe object is described completely by certain fixed parameters and by its state variables. The parameters are:

- The sponge's rate r and capacity c. We require r and c to be multiples of one byte (8 bits) with r at least 3 bytes and at most 256 bytes. In Appendix C.2 we describe how

a certain power analysis countermeasure is very similar to reducing the rate to only a few bits. Let $\hat{r} = r/8 - 2$. This is the portion of the rate which is used for user data, measured in bytes.

- The sponge function $F : \mathcal{B}^{(r+c)/8} \to \mathcal{B}^{(r+c)/8}$. Our recommendations use Keccak-f instances.

- The initial state $S_0 \in \mathcal{B}^{(r+c)/8}$. This may be nonzero for domain separation from other sponges with the same function F. Our recommendations use the draft cSHAKE [13] domain separation with strings of the form "STROBE v$x.y$".

- A nonzero constant byte `DDATA`. This is absorbed into the sponge at the end of every block to delimit how much data was in the block. Our recommendations use `0x04` for compatibility with cSHAKE.

- A constant byte `DRATE`, which has no security effect and may be zero. Our recommendations use `0x80` for compatibility with cSHAKE.

The state variables are:

- The sponge state $S \in \mathcal{B}^{(r+c)/8}$.

- The position within the sponge state $p \in [0, \hat{r}]$, measured in bytes. This is the index of the next byte of the sponge state to be used. The last two bytes are reserved for padding.

- A second position $p_{\text{begin}} \in [0, \hat{r} + 1]$, also measured in bytes. This is set to p when the `beginOp` function is called.

- A value $I_0 \in \{0, 1, \perp\}$, to indicate whether this party is an initiator or responder. $I_0 = 0$ for a initiator, which is a party that sent a message to the transport before receiving one. It is 1 for a responder, which is a party that received before sending. It is $\perp$ if this party has neither sent nor received a message.

STROBE has three subroutines: `runF`, `beginOp` and `duplex`.

A.2 runF

The subroutine `runF`() records the current position p and operation beginning p_{begin}, and then runs the protocol's F function.

Function `runF` **is**

> Update $S[p] \leftarrow S[p] \oplus p_{\text{begin}}$;
> Update $S[p+1] \leftarrow S[p+1] \oplus \texttt{DDATA}$;
> Update $S[\hat{r}+1] \leftarrow S[\hat{r}+1] \oplus \texttt{DRATE}$;
> Update $S \leftarrow F(S)$;
> Reset $p \leftarrow 0$ and $p_{\text{begin}} \leftarrow 0$;

end

A.3 beginOp

The `beginOp` subroutine is run at the beginning of a new operation. It absorbs a byte into the sponge state which describes what the operation is. It also writes p_{begin} into the sponge state, and then changes p_{begin} to delimit the beginning of the operation. Finally, if the operation uses the cipher state, it runs the F function so that all prior changes affect the sponge's output. Note that the K flag is an extension and might not be implemented.

Function `beginOp` (I, A, C, T, M, K) **is**

> **input** : operation (I, A, C, T); metadata indicator M; keytree indicator K
> Update $S[p] \leftarrow S[p] \oplus p_{\text{begin}}$;
> $p \leftarrow p + 1$; $p_{\text{begin}} \leftarrow p$;
> **if** $p \geq \hat{r}$ **then** `runF` ();
> **if** $I_0 = \bot$ **and** $T = 1$ **then** $I_0 \leftarrow I$;
> opByte $\leftarrow [[I \oplus (T \cdot I_0), A, C, T, M, K, 0, 0]] \in \mathcal{B}$;
> Update $S[p] \leftarrow S[p] \oplus$ opByte;
> $p \leftarrow p + 1$;
> **if** $p \geq \hat{r}$ **or** $C = 1$ **or** $K = 1$ **then** `runF` ();

end

A.4 duplex

We are now ready to describe the main STROBE duplex mode. It is worth noting that while this duplex mode is expressed in terms of a length L or message m known before calling `duplex`, this is not actually required. The algorithm can be run in a streaming fashion, one byte at a time.

```
Function duplex ((I, A, C, T, M, K);  L or m) is
    input  : Operation (I, A, C, T, M, K)
    input  : If A = 1 and I = 0, an L-byte message m from the application.
    input  : If T = 1 and I = 1, an L-byte message m from the transport.
    input  : Otherwise, a message length L in bytes.

    output: If A = 1 and I = 1, an L-byte message for the application.
    output: If T = 1 and I = 0, an L-byte message for the transport.
    output: Otherwise, success or failure.

    // If no input, then input is zeros
    if (I = 0 and A = 0) or (I = 1 and T = 0) then m ← L × [[0]];

    // Mark our position
    beginOp(I, A, C, T, M, K);

    // Duplex operation
    if K = 1 then
        // DPA-resistant key tree as specified in Section C.2
        ...;
    else
        for each byte b in m do
            if C = 1 then append b ⊕ s[p] to output;
            else append b to output;
            if C = 1 and (I = 1 or T = 0) then s[p] ← b;
            else s[p] ← b ⊕ s[p];
            p ← p + 1;
            if p ≥ r̂ then runF ();
        end
    end

    // Write output
    if I = 0 and T = 1 then send the output to the transport;
    else if I = 1 and A = 1 then return the output to the application;
    else if (A, T, I) = (0, 1, 1) and output ≠ L × [[0]] then fail: invalid MAC;
    else return success;
end
```

Algorithm 1: STROBE duplex operation

B Instantiation

Our main recommendations are STROBE-128, STROBE-256 and STROBE-128/800 as described below.

B.1 Strobe

The protocol framework STROBE-f-λ/b is a STROBE instance with a target security level of λ bits. It has $c = 2\lambda$, $r + c = b$ and $F = f[b]$. If f is Keccak-f, it is omitted from the name. If b is the default b for that f (1600 in the case of Keccak-f) it is likewise omitted. For all these instances, `DDATA` and `DRATE` are `0x04` and `0x80`, respectively.

Let V be the 11-byte string "STROBEv$x.y.z$", for single-digit major version x, minor version y, and patch version z, initially 1.0.0. Let the initial state be

$$\begin{aligned} S_0 &:= F(\text{bytepad}(\text{encode_string}(V)||\text{encode_string}(\text{``''}), r/8)) \\ &= F([[1, r/8, 1, 12]] \mid\mid V \mid\mid [[0]] \mid\mid (r/8 - 17) \times [[0]]) \end{aligned}$$

Then the data absorbed into the sponge before calling F is always of the form

$$\text{bytepad}(\text{encode_string}(S)||\text{encode_string}(\text{``''}), r/8) \mid\mid X \mid\mid \mathtt{0010^*1}$$

for some X. The padding is written `0010*1` as a little-endian bit string; as a byte string it is either `0x84` or `0x04 0x00* 0x80`. For STROBE-128, any data squeezed from the sponge will then be draft cSHAKE128(X). Likewise, for STROBE-256, any data squeezed from the sponge will be of the form draft cSHAKE256(X).

We recommend using 256-bit keys even with STROBE-128, if this can be afforded. Because of the 256-bit capacity, this construction is designed to take on the order of 2^{128} operations to break even with 2^{128} data. It is therefore stronger than the traditional notion of 128-bit security. With only 128-bit keys, an adversary who gets n encryptions with the same nonce and n different keys would need only $2^{128}/n$ operations to break one key, as in a traditional 128-bit cipher mode. This same degradation applies to STROBE-256 if 256-bit keys are used.

STROBE-128/800 is ideal for efficiency on 32-bit microcontrollers, because Keccak-f[800] uses 32-bit logical operations. Also, the smaller 100-byte state is easier to squeeze into resource-constrained environments. As with STROBE-128, better security will result from using 256-bit keys when resources permit.

STROBE-128/400 should offer a competitive combination of light weight and high speed in hardware. A cycle-per-round Keccak-f[400] implementation costs less than 11 kilogate equivalents (kGE) [12]. After adjusting for STROBE's reduced rate $\hat{r}$, this would consume 1.25 cycles per byte. An AES-128 engine with comparable throughput costs 8.8 kGE [20], but this includes no hashing or authentication. A comparable SHA-256 engine would cost almost 20 kGE [10].

B.2 Strobe lite

Since Strobe is simple and adaptable, it is attractive to use it in lightweight hardware. Suitable sponges include Keccak-f[200], as well as lightweight sponges such as Photon [9]. However, with a rate of only a few bytes, it makes no sense to track or encode the beginOp position p_{begin}. Instead, we simply end the block, using a different `DRATE` to indicate that a new operation is beginning. This enables us to use $\hat{r} = r/8 - 1$ instead of $r/8 - 2$. Since we certainly cannot be compatible with cSHAKE, we don't need a `DRATE`. We also simplify by setting `DDATA` = `0x03` when beginning an operation and `0x02` otherwise. We then modify `runF` and `beginOp` as follows:

Function `runF_lite` **is**
    Update $S[p] \leftarrow S[p] \oplus$ `0x02`;
    Update $S \leftarrow F(S)$;
    Reset $p \leftarrow 0$;
end

Function `beginOp_lite` (I, A, C, T, M, K) **is**
    input : operation (I, A, C, T, M, K)
    if $I_0 = \perp$ **and** $T = 1$ **then** $I_0 \leftarrow I$;
    opByte $\leftarrow [[I \oplus (T \cdot I_0), A, C, T, M, K, 0, 0]] \in \mathcal{B}$;
    Update $S[p] \leftarrow S[p] \oplus$ opByte;
    Update $S[p+1] \leftarrow S[p+1] \oplus$ `0x03`;
    Update $S \leftarrow F(S)$;
    Reset $p \leftarrow 0$;
end

The `RATCHET` operation is not effective at preventing rollback when $\hat{r} < \lambda/8$. Instead one must instead read $\lambda/8$ bytes with `PRF`, then writing them back with `KEY`.

B.3 Non-sponge

Strobe's operations can also be implemented with a hash and a PRF (eg, a block cipher) instead of a sponge function. This forgoes the code size and memory advantages of a sponge function. But it may be preferable for standards-compliance reasons, or when AES and SHA accelerators are available.

We leave the exact implementation to future work. It should look very similar to the symmetric operations in Noise [16].

C Extensions

C.1 Full-duplex and multiple-channel protocols

Practical protocols often have a half-duplex key agreement phase, but once the key has been determined, the protocol becomes full-duplex. This use case is easy to implement with Strobe. One can treat a full-duplex protocol as two half-duplex protocols, one in each direction. For each half-duplex instance, one copies the Strobe object's state. To make the sponge states different and to ensure forward security, it is sufficient to run on each state a `meta-AD` and a `RATCHET` operation with a different AD for each direction. This also works for protocols which instantiate more than two channels, but at the cost of a separate Strobe object for each channel.

In some cases, it is sufficient just to copy the Strobe state without performing any other operations. This will be enough if three conditions are met:

- Only two half-duplex channels are required, one in each direction.
- Forward secrecy is not required within the session. Otherwise an attacker who compromises the state for one direction might learn the initial state of the other direction.
- At least one message has been sent to the transport already, so that $I_0 \neq \perp$. This ensures that the opByte is different between the two directions.

C.2 DPA countermeasures

Differential power analysis (DPA [14]) is a side-channel attack which extracts a key by observing its use with many different inputs. This is especially a problem with conventional block cipher modes, because they do not change the key between blocks. Strobe's secret data changes every block to a new pseudorandom value, which naturally mitigates DPA attacks. Furthermore, Keccak-f itself is suited to primitive-level DPA countermeasures due to its relative paucity of nonlinear operations. But the area and performance cost of primitive-level countermeasures may deter implementers, so protocol countermeasures are desirable.

Even though the secret data changes every block, there is the potential for attack when the same key is used in several Strobe instances, e.g. as a long-term pre-shared symmetric key. This can be mitigated by using a nonce for each connection. Each party injects the key using a `KEY` operation immediately after the session is initialized, before it contains any per-session data. This operation resists DPA because the key is mixed with a constant state. Then one nonce must be contributed by each party that might be subject to a DPA attack, to ensure uniqueness. After the nonces are mixed, the parties `RATCHET` to prevent rollback.

The resulting session will have a unique shared key which is a pseudorandom function of the long-term key. Since this diversified key is never reused, the inherent resistance of stream cipher modes should be enough to protect against DPA.

There is still the question of how to mix in the nonce in a DPA-resistant manner. To do this, we use a variant of the "key tree" operation [18], which is essentially the GGM construction of a pseudorandom function [7]. This is almost exactly equivalent to reducing the rate to r_{kt} bits for some small r_{kt} dividing 8.[4] Then each intermediate capacity value can only ever be used with at most $2^{r_{\mathrm{kt}}}+1$ possible inputs.

Here we describe the key tree duplex operation, to be slotted into the ellipsis in `duplex`. In practice the "**for each r_{kt}-bit chunk**" loop will be a nested loop which iterates over each byte and modifies each byte several times.

```
// Key tree duplex operation.  Note that p = p_begin = 0 because beginOp
   runs F
for each r_kt-bit chunk b in m do
    if C = 1 then append b ⊕ (first r_kt bits of s[0]) to output;
    else append b to output;
    if C = 1 and (I = 1 or T = 0) then s[0] ← s[0] & (2^8 − 2^r_kt);
    s[0] ← b ⊕ s[0];

    // Implement runF but with p_begin = 0 and p = (r_kt bits).
    Update S[1] ← S[1] ⊕ (DDATA ≪ r_kt);
    Update S[2] ← S[2] ⊕ (DDATA ≫ (8 − r_kt));
    Update S[r̂ + 1] ← S[r̂ + 1] ⊕ DRATE;
    Update S ← F(S);
end
```

When this algorithm is employed, each key is only passed to F in a few ways:

- The initial state is marked and then passed to F. The number of different states passed to F this way is the number of possible $\mathrm{AdjDir}(I, A, C, T, M)$ operations which can be called in that state, which depends on the protocol but is likely to be very small.
- Each intermediate state is either used in the loop body, with $2^{r_{\mathrm{kt}}}$ possible values of b before running F again, or exits the loop at the bottom. To prevent rollback, we recommend that the next call be `duplex` (`RATCHET`, $c/8$ bytes), which also calls F in exactly one way.

[4] It is also possible to do this with $r_{\mathrm{kt}} > 8$ or r_{kt} not dividing 8, but it adds extra unenlightening code, so we omit this.

The key tree algorithm can be used for DPA-resistant authenticated encryption, by using a key tree as a DPA-resistant MAC algorithm [18].