Julia: Fast and Secure Key Agreement for IoT Devices

Frans Lundberg
Juraj Feljan
frans.lundberg@assaabloy.com
juraj.feljan@assaabloy.com
ASSA ABLOY AB
Sweden

ABSTRACT

Even the most resource-constrained IoT devices need to communicate securely. In order to establish a secure channel, key agreement between the communicating parties is used. Today's key agreement protocols require at least three scalar multiplications in the handshake to achieve mutual authentication, forward and backward secrecy, and protection against key-compromise impersonation. As this is a computationally heavy operation, resource-constrained devices benefit from a lower number of scalar multiplications. In this paper we present Julia Key Agreement (JKA), a protocol that satisfies the aforementioned security properties using two scalar multiplications, and thus saves both time and energy. In addition, we define an optimized JKA that only requires a single scalar multiplication for a particular use case.

CCS CONCEPTS

• **Security and privacy → Public key encryption**; • **Computer systems organization → System on a chip**.

KEYWORDS

IoT, scalar multiplication, asymmetric cryptography, secure channel, key agreement, handshake, Diffie-Hellman

ACM Reference Format:
Frans Lundberg and Juraj Feljan. 2021. Julia: Fast and Secure Key Agreement for IoT Devices. In *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21), June 28–July 2, 2021, Abu Dhabi, United Arab Emirates.* ACM, 10 pages. https://doi.org/10.1145/3448300.3468116

1 INTRODUCTION

One of the many challenges when it comes to security for the Internet of Things (IoT) is the fact that a typical IoT device is resource-constrained. The reason for the limited specification is two fold — to keep the price of systems on a chip (SoC) low and to limit their energy consumption. The former is necessary since IoT hardware is typically obtained in large batches, while the latter is dictated by the fact that many products do not have a constant source of power — they are either battery-powered or dependent on some form of energy harvesting (for example, the harvested energy can be as little as 10 mJ, with a budget for computations of only 2 mJ).

In order to communicate securely, communicating parties need to establish a secure channel over an insecure network. The traditional approach still used in many systems today is to utilize symmetric cryptography exclusively. The devices and servers are provisioned with pre-shared secrets — either a common secret for the whole system, or pairwise secrets for every two communicating parties. The main issue with using a common secret is that the system is left unprotected should any device become compromised leaking the secret. Pairwise secrets tackle this, but instead introduce the problem of scalability — the number of secrets grows exponentially with the number of nodes. On the other hand we have asymmetric cryptography where each device has a cryptographic identity represented by its public key, while also holding the corresponding secret key. If the secret key is leaked, only this single device is compromised, not the whole system. Asymmetric cryptography however requires algorithms that are more computationally intensive than the ones used for symmetric cryptography. If we combine asymmetric and symmetric cryptography, we can leverage their respective advantages — this is the approach used by modern secure channels. For each communication session, the communicating parties asymmetrically compute a shared secret. From the shared secret a symmetric key is derived, which is then used to encrypt and decrypt the communicated data. The shared secret is never sent over the network. Instead, both parties arrive at the same shared secret during a *key agreement*.

A key agreement protocol needs a certain amount of computational steps in order to establish a number of relevant security properties (we define these in Section 3). The main computational unit is a scalar multiplication, a resource-hungry operation and the main contributor to the duration and energy consumption of the key agreement. At ASSA ABLOY, we have experience from working with a wide range of embedded hardware, including the popular ARM Cortex M series of processors (for example the nRF52840 from Nordic Semiconductor or MAX32670 from Maxim Integrated), the MSP430 from Texas Instruments, and special processors embedded on NFC-powered smart cards. In all cases where we benchmarked key agreement protocols, scalar multiplications took over 80 percent of the processor time spent on computation. Typically on an embedded processor, the time spent on computation is proportional to the consumed energy. This means that we can reduce both the duration and the energy consumption of a key agreement with a lower number of scalar multiplications. However, this must not come at the cost of sacrificing the required security properties.

WiSec '21, June 28–July 2, 2021, Abu Dhabi, United Arab Emirates

ACM ISBN 978-1-4503-8349-3/21/06.
https://doi.org/10.1145/3448300.3468116

In this paper we present Julia Key Agreement (JKA), our solution for achieving mutual authentication, forward and backward secrecy, and protection against key-compromise impersonation with only two scalar multiplications in the handshake. In comparison, the key agreement protocols in use today require at least three scalar multiplications. The paper is organized as follows. In Section 2 we introduce the necessary mathematical prerequisites, before defining the problem in detail in Section 3. Section 4 describes the core contribution. In Section 5 we present related work, before discussing future work and concluding the paper in Section 6.

2 PREREQUISITES AND NOTATION

In this section we introduce the mathematical prerequisites necessary for understanding the contribution of the paper. We also define the notation used throughout the text.

In mathematics, a group is a set[1] equipped with a group operation. By definition, the group operation (denoted here by +) is binary, and when applied to any two group elements it produces a result that is also an element of the group. The set of the elements of a group is denoted with S. The following three axioms are valid for the group operation:

- Associativity. For any three group elements A, B, C in S, $(A + B) + C = A + (B + C)$.
- Identity. There exists an element 0 in S, such that for every A in S, $0 + A = A + 0 = A$. 0 is unique for S.
- Invertibility. For every A in S, there exists an element $-A$ in S, such that $(-A) + A = A + (-A) = 0$.

A scalar multiplication is a multiplication of a scalar (integer) with a group element. We denote the operation with $*$. Uppercase letters are used for group elements, while lowercase letters are used for scalars. In the equation below, C is a group element and the result of a scalar multiplication between an integer a and a group element B:

$$C = a * B \tag{1}$$

Since we differentiate between scalar and group elements, we also allow the reverse notation with the first operand being a group element, $C = B * a$. We can intuitively define a scalar multiplication as a repeated application of the group operation. The equations below illustrate this:

$$\begin{aligned} 0 * A &= 0 \\ 1 * A &= A \\ 2 * A &= A + A \\ 3 * A &= A + A + A \end{aligned} \tag{2}$$

Note that in the first equation above, the first zero is a scalar, while the zero to the right is a group element (the same symbol is used for both the scalar zero and the identity element of the group). Note also that the last equation can be written without parenthesis thanks to the associativity of the group, $(A + A) + A = A + (A + A)$.

Having defined mathematical groups in general, we can now focus on the specific kind of groups that are of interest in cryptography. For these groups, the computational discrete logarithm problem is hard (more specifically, it is assumed impossible to be solved in practice). They are finite, and their number of elements is typically in the order of 2^{256} or more. Groups that satisfy these criteria are used in elliptic-curve cryptography (ECC), for example Curve25519 [2] and NIST P-256 [8]. Given a randomly chosen integer a, we can compute:

$$A = a * G \tag{3}$$

G is a *generator*, a group element for which the series $0 * G$, $1 * G$, $2 * G$... will enumerate all the elements of the group. For ECC, computing Equation 3 takes $O(k)$ time for a key size of k. The inverse operation, to compute a given A, is much more difficult and requires $O(2^{k/2})$ time. This leads us to the following intuitive understanding of the computational discrete logarithm problem being hard — in reasonable time we can perform the computation in one way only — we can get A given a, but we cannot obtain a given A. In other words, given a secret key, we can compute the corresponding public key, but we cannot calculate the secret key from the public one. In Equation 3, a denotes the secret key, while A is the corresponding public key.

How to efficiently compute scalar multiplication for groups used in cryptography is a relevant topic that has attracted a substantial amount of research. See [7] for a survey of fast scalar multiplication suitable for IoT devices.

3 THE PROBLEM

In this section we define the scope of the problem that the contribution of the paper solves. We first discuss why we focus on the number of scalar multiplications. Then we state the requirements for a viable solution. Finally, in order to further demonstrate the problem, we present three key agreement protocols that do not represent a complete solution, and discuss in which aspects these fail.

3.1 Why scalar multiplications?

The premise of the paper is that we can optimize a key agreement protocol by reducing the number of necessary scalar multiplications. In this subsection we motivate this claim.

For a use case where a key communicates with a lock over UART, we performed measurements of an unlock session, which includes a key agreement, sending an unlock command with a credential, and finally verifying the credential, exchanging in total 906 bytes of data. The asymmetric cryptographic operations comprised 93 percent of the total duration of the cryptographic operations (including asymmetric, symmetric and hash operations). From the total duration of the unlock session, the computations took 64 percent, while the I/O operations took 19 percent (the rest was idle time). In a use case involving a smart card and a reader performing a key agreement over NFC, we measured that the asymmetric cryptographic operations took 96 percent of the total duration of the cryptographic operations. These two measurements illustrate that asymmetric operations represent the majority of the cost of a key agreement. Later (in Section 4.3) we, in turn, show that it is the scalar multiplications that comprise the bulk of the asymmetric operations. Combining these two facts backs our statement made in the introduction — that scalar multiplications compose the majority of the processor time and energy consumption for a key agreement protocol.

[1] A set is a collection of distinct elements.

Note that in addition to CPU-bound cases, where the computations are slower than the communication, we also have use cases that are I/O-bound — when we need to communicate securely over transports with a low bandwidth or a high round-trip time. For example, this can include communication over Bluetooth Low Energy (BLE), NB-IoT or wireless mesh networks. These cases are I/O-bound in the sense that the total key agreement time mostly comprises of I/O operations rather than computations. Despite not being able to significantly influence the key agreement duration for I/O-bound cases, it may still be possible to reduce the energy consumption through a reduction of the number of scalar multiplications. Typically, I/O-bound cases are optimized by reducing the number of round-trips, and our solution takes this into account — we are already at the theoretical minimum for a key agreement, namely a one round-trip overhead.

Not all scalar multiplications need to be implemented in the same way for a given protocol. In some cases, scalar multiplication is performed with an unknown group element, and it must be done in constant time (the execution time cannot depend on secret data) to prevent side-channel attacks. An example of such an operation is computing the Diffie-Hellman between two key pairs ($s_1 * S_2 = s_2 * S_1$). On the other hand, scalar multiplication is sometimes performed with a known group element. For example, when a key pair is generated, the group element used in the operation is always the group generator G ($S_1 = s_1 * G$). This allows for optimization that is not possible in the former case. Additionally, for some uses of scalar multiplication there may be no need for a constant-time implementation. For example, when a signature is verified, no secret data is used as input, and thus a variable-time implementation is allowed. Even though a certain degree of optimization can be achieved by using multiple implementations for scalar multiplication in a given protocol, we often opt to use a single implementation for a specific hardware target. This is to save development cost, to keep the code complexity as low as possible (which in itself is important for security), and to keep the code size small. What this means is that we in practice do not aim to minimize the computation costs by selectively optimizing particular scalar multiplications, but we can instead benefit from a reduction in the number of necessary scalar multiplications.

In order to precisely compare different key agreement protocols, optimized implementations on production target hardware are required. However, this is expensive. Optimizing the performance of cryptographic operations on embedded targets is time consuming and difficult, since it often involves writing constant-time assembly code. However, in our experience, the total number of scalar multiplications a party has to perform during a key agreement handshake gives an inexpensive and appropriate approximation of the practical performance that can be attained by an implementation. In Section 4.3 we present performance measurements that further support this claim.

3.2 Solution requirements

The goal of this work was to define a key agreement protocol that provides the required security properties (defined in the list below), but with less than three scalar multiplications in the handshake. The parties P_1 and P_2 want to communicate securely over a network. Each party P_i has a static secret key s_i and the corresponding public key S_i. The identity of party P_i is defined as ownership of the secret key s_i — anyone holding the secret key s_i can identify as party P_i. We assume a powerful attacker with full control over the network. The attacker can read all messages ever sent on the network, send arbitrary messages to any party, or block any message. The purpose of the key agreement protocol is for P_1 and P_2 to agree on a shared secret D. For a particular session, D must be known only by the two communicating parties. D must be unique for every communication session. The uniqueness applies over all sessions that ever took place on the network between any two parties.

A valid solution must compute a shared secret with less than three scalar multiplications per party, and meet the following requirements:

- It must provide *mutual authentication* of the parties. This means that each party proves ownership of the secret key corresponding to its public key.
- It must provide *forward secrecy*. Even with access to both s_1 and s_2, an attacker must not be able to calculate the shared secrets of previously recorded communication sessions.
- It must provide *backward secrecy*. Even with access to both s_1 and s_2, an attacker must not be able to calculate the shared secrets of future passively recorded communication sessions.
- It must provide protection against *key-compromise impersonation (KCI) attacks*. Even with access to s_1, an attacker must not be able to pretend to P_1 to be someone else. Likewise, if the attacker has access to s_2, she must not be able to impersonate someone else to P_2. In the general case, an attacker can impersonate to a party P if she can compute a shared secret D with P for a public key S for which she does not have the secret key s.

These are the typical security properties that the application layer requires from the secure channel layer in the solutions developed at ASSA ABLOY.

A valid question at this point is why we have not listed two fundamental security properties, namely confidentiality and integrity of the application data. This is because of the scope of the paper — we focus solely on the key agreement protocol, and do not define the subsequent symmetric operation of the established secure channel. It is the symmetric part of the secure channel that is responsible for confidentiality and integrity. A necessary requirement for being able to provide confidentiality and integrity is that no attacker is able to obtain D. An example of a symmetric algorithm that ensures confidentiality and integrity through authenticated encryption is Salsa20-Poly1305, as available from the NaCl cryptographic library ([4], [5]). Another aspect that is outside of the scope of the key agreement protocol for us in this paper, and that belongs to the symmetric operation of the secure channel, is how a shared key (or multiple shared keys) is derived from the calculated shared secret. Having the scope of the problem in mind, a viable solution terminates when P_1 and P_2 have attained a fresh shared secret D.

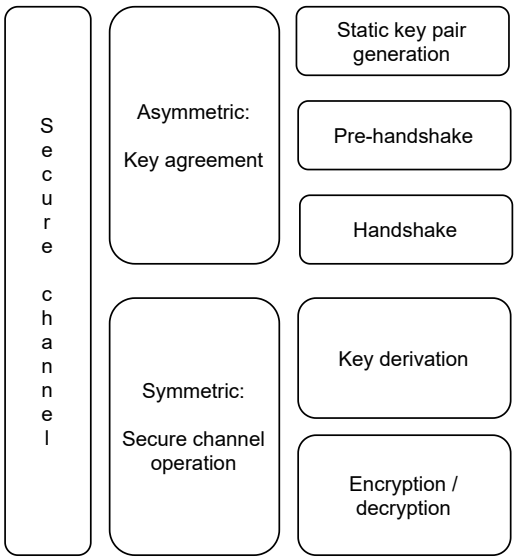

Figure 1: Secure channel terminology

When it comes to mutual authentication, all the protocols listed in the paper do provide mutual authentication — possession of the secret key is necessary in order to be able to calculate a shared secret. However, proof of possession of the secret key becomes available first on receiving an application message encrypted using a derived symmetric key.

In order to make sure that our usage of secure channel terminology is clear, we visualize the used terms in Figure 1. A secure channel has an asymmetric and a symmetric phase. In the former, a key agreement protocol is used to compute a shared secret. In the latter, the shared secret is used to derive a symmetric key (or multiple symmetric keys), which is then in turn used for encryption and decryption of application data. As mentioned above, only the key agreement is in the scope of the paper. We can divide a key agreement protocol into three phases. In the first phase, the long-term static key pairs are generated. Typically (but not always) this is only done once in the lifetime of an IoT device. The second phase (pre-handshake) includes the operations that need to be performed for each new session, but that can be computed prior to starting communication with another party, and thus stored on the device ready for the new session. Finally, we have the handshake phase in which the two parties communicate while computing a shared secret.

Our requirement on less than three scalar multiplications per party refers to the handshake phase. We allow pre-computation both for P_1 and P_2. Confidentiality of the public keys S_1 and S_2 is not required. Mutual authentication is based on raw public keys. The details of how these public keys are in turn authenticated as belonging to a particular entity is important, but out of scope. This can, for example, be achieved using certificates, out-of-band communication prior to the session, email signatures, or a public online directory.

3.3 Examples of incomplete solutions

In this subsection we take a look at three incomplete solutions, three protocols that do not satisfy all the listed requirements. The purpose of this is to provide an additional intuitive understanding of the problem, and to equip the reader with ideas on how to break a key agreement protocol, before we present our solution in the following section. The examples in this subsection also represent the current state of the art of key agreement protocols based on authentication using static Diffie-Hellman key pairs.

Protocol P01 is a simple Diffie-Hellman key agreement that requires only one scalar multiplication for each party.

$$
\begin{array}{ll}
P_1 & P_2 \\
\text{// Static keys} & \\
s_1 = secret() & s_2 = secret() \\
S_1 = s_1 * G & S_2 = s_2 * G \\
\xrightarrow{S_1} & \\
\xleftarrow{S_2} & \\
\text{// Handshake} & \\
c_1 = rand() & c_2 = rand() \\
\xrightarrow{c_1} & \\
\xleftarrow{c_2} & \\
D = s_1 * S_2 & D = S_1 * s_2
\end{array}
\tag{P01}
$$

Generation of the static key pairs for P_1 (s_1, S_1) and P_2 (s_2, S_2) and the exchange of the static public keys (S_1, S_2) is included above for illustration purposes, but from now on this phase will be omitted. The public keys (S_1, S_2) are exchanged prior to the handshake phase. When the handshake begins, each party already knows who it expects to communicate with. This is one scenario, but there are other possibilities for how a party can attain the public key of the counterpart. In some secure channel protocols, certificate chains are used, some use raw public keys, while others use short references that can be used to lookup the public key from some out-of-band source. The exact method of how P_1 attains the public key S_2, and likewise how P_2 attains the public key S_1 is not conceptually important for the key agreements presented in this paper. Therefore, this is omitted from the protocols that follow.

$secret()$ is a function that returns a new random secret key in the form of a scalar — an integer that can be used in a scalar multiplication. $rand()$ is a function that returns random data in the form of a byte array.

In the handshake, P_1 creates a random challenge c_1 and sends it to P_2. P_2 in turn sends a random challenge c_2 to P_1. Then they compute a shared secret D. D can then be used together with a hash of the session data (c_1 and c_2) to derive a symmetric key used to protect the subsequent session data. As discussed earlier, we consider this to be a part of the symmetric operation of the secure channel, and not of the key agreement protocol.

Just one scalar multiplication is needed by each party. However, the protocol lacks compromised key security properties. If an attacker gets hold of s_1, she can easily decrypt the session data by computing $D = s_1 * S_2$, and then computing the derived symmetric key. Similarly, if she gets hold of s_2, she can compute $D = s_2 * S_1$.

The attacker can also successfully do a KCI attack — if she has s_1, she can pretend to be P_2 to P_1. Actually, she can pretend to be anyone to P_1, because she can use any public key S when calculating the shared secret ($D = s_1 * S$).

Protocol P02 is based on ephemeral (temporary) key pairs that are only used for a single session, and then discarded. Note that the creation of the static keys is omitted from the protocol listing. Also the P_1/P_2 header is removed to keep the listing compact. The left-hand side is always party P_1 and P_2 is always to the right.

$$
\begin{array}{ll}
\text{// Pre-handshake} & \\
e_1 = secret() & e_2 = secret() \\
E_1 = e_1 * G & E_2 = e_2 * G \\
 & \\
\text{// Handshake} & \\
\xrightarrow{E_1} & \\
\xleftarrow{E_2} & \\
D_1 = s_1 * E_2 & D_1 = S_1 * e_2 \\
D_2 = e_1 * S_2 & D_2 = E_1 * s_2
\end{array}
\qquad \text{(P02)}
$$

Before the handshake, each party computes their ephemeral key pair for the next session. In the handshake phase, they exchange the public ephemeral keys. P_1 computes two scalar multiplications using her secret keys (s_1 and e_1), resulting in D_1 and D_2. The results of the scalar multiplications are then combined together with a running hash of the exchanged data to derive a symmetric key (not shown in the protocol). Similarly, P_2 can compute the same values D_1 and D_2 using her secret keys (s_2 and e_2), and then derive the symmetric key.

Let us consider a scenario where an attacker gets access to the secret key s_1. Since she has s_1 she can compute $D_1 = s_1 * E_2$. However, she cannot compute D_2 because she does not have access to s_2 nor e_1. So the protocol does protect against KCI attacks.

How about forward secrecy? If the attacker gets hold of the secret keys s_1 and s_2, she can decrypt the session data. She can compute both $D_1 = s_1 * E_2$ and $D_2 = E_1 * s_2$. So this protocol does not provide full forward secrecy. Note, however that she needs both secret keys to do so. If she only has one of the secret keys, she cannot compute both D_1 and D_2.

Protocol P03 is conceptually the same as the XX pattern of the Noise protocol framework [13].

$$
\begin{array}{ll}
\text{// Pre-handshake} & \\
e_1 = secret() & e_2 = secret() \\
E_1 = e_1 * G & E_2 = e_2 * G \\
 & \\
\text{// Handshake} & \\
\xrightarrow{E_1} & \\
\xleftarrow{E_2} & \\
D_1 = e_1 * E_2 & D_1 = E_1 * e_2 \\
D_2 = e_1 * S_2 & D_2 = E_1 * s_2 \\
D_3 = s_1 * E_2 & D_3 = S_1 * e_2
\end{array}
\qquad \text{(P03)}
$$

This protocol fulfills all the security properties that we require. However, it does not represent a viable solution since each party performs three scalar multiplications in the handshake.

4 JULIA KEY AGREEMENT

In this section we specify our Julia Key Agreement (JKA) protocol. We then reason about our design decisions, followed by a discussion about how the protocol satisfies the requirements listed in Section 3.2. After this we demonstrate the performance benefit of JKA. Then we show an optimization of JKA, before ending the section with a presentation of our implementation.

The Julia Key Agreement protocol is defined as follows:

$$
\begin{array}{ll}
\text{// Pre-handshake} & \\
e_1 = secret() & e_2 = secret() \\
E_1 = e_1 * G & E_2 = e_2 * G \\
t_1 = hash(\text{"}t1\text{"}|E_1) & t_2 = hash(\text{"}t2\text{"}|E_2) \\
h_1 = hash(\text{"}h1\text{"}|E_1) & \\
 & \\
\text{// Handshake} & \\
\xrightarrow{h_1} & \\
\xleftarrow{E_2} & \\
t_2 = hash(\text{"}t2\text{"}|E_2) & \\
t = t_1 + t_2 & \\
D = (ts_1 + e_1) * (t * S_2 + E_2) & \\
 & \\
\xrightarrow{E_1, app1} & \\
 & Verify: h_1 = hash(\text{"}h1\text{"}|E_1) \\
 & t_1 = hash(\text{"}t1\text{"}|E_1) \\
 & t = t_1 + t_2 \\
 & D = (t * S_1 + E_1) * (ts_2 + e_2)
\end{array}
\qquad \text{(P04)}
$$

Prior to the handshake, P_1 and P_2 create ephemeral key pairs, and compute integers t_1 and t_2, respectively[2]. For each of the parties P_i, t_i is calculated as a hash of a concatenation of the string "$t1$" or "$t2$" and the public ephemeral key E_i[3]. Additionally, P_1 calculates a hash of a concatenation of the string "$h1$" and its public ephemeral key E_1. This way P_1 *commits* to the value before sending it over the network. The commit-before-sending is essential so that an attacker cannot modify E_2 based on the value of E_1. In Section 4.1 we discuss how this could otherwise be exploited. The strings in the hashes ("$t1$" and "$h1$") are chosen arbitrarily, but it is important that they are different so that the values t_1 and h_1 are different, in order to prevent the same kind of attack.

When the handshake begins, P_1 sends h_1 to P_2, while P_2 responds with its public ephemeral key E_2. Having received E_2, P_1 can calculate t_2, which in turn enables it to compute the shared secret D. Having done this, it sends its public ephemeral key E_1 to P_2. In the same transaction it sends the first application message $app1$ encrypted with the symmetric session key (we do not show the key derivation step explicitly because it belongs to the symmetric operation of the secure channel). The reason for sending the first

[2] A secure hash function generally has an output in the form of a fixed-length byte array. How to compute an integer from that output is out of scope for this paper. However, our SageMath implementation includes an example of how this can be achieved.

[3] This operation requires that the public key (a group element) is serialized to an array of bytes. Exactly how this is achieved is out of scope for this high-level protocol description. See our implementation for an example.

application message already now is to keep the handshake overhead to one round trip. P_2 can now verify E_1 against h_1 received in the first step of the handshake. If the verification succeeds, P_2 calculates t_1, which enables it to calculate the shared secret D.

The expression for calculating the shared secret is the core of JKA. We call it the Julia expression. It is defined as:

$$D = (ts_1 + e_1)(ts_2 + e_2) * G \tag{4}$$

and is computed as $(ts_1 + e_1) * (t * S_2 + E_2)$ by P_1 and as $(t * S_1 + E_1) * (ts_2 + e_2)$ by P_2. Let's write these expressions together for clarity:

$$\begin{aligned} D &= (ts_1 + e_1)(ts_2 + e_2) * G \\ &= (ts_1 + e_1) * (t * S_2 + E_2) \\ &= (t * S_1 + E_1) * (ts_2 + e_2) \end{aligned} \tag{5}$$

We can see that P_1 and P_2 each compute only two scalar multiplications (denoted by $*$).

4.1 Design decisions behind JKA

Let us consider the aforementioned exploit that would be possible if P_1 did not commit to E_1 by sending h_1. This is how the protocol would be defined in that case:

// Pre-handshake	
$e_1 = secret()$	$e_2 = secret()$
$E_1 = e_1 * G$	$E_2 = e_2 * G$
$t_1 = hash("t1"\|E_1)$	$t_2 = hash("t2"\|E_2)$
// Handshake	
$\xrightarrow{E_1}$	
$\xleftarrow{E_2}$	
$t_2 = hash("t2"\|E_2)$	
$t = t_1 + t_2$	
$D = (ts_1 + e_1) * (t * S_2 + E_2)$	
	$t_1 = hash("t1"\|E_1)$
	$t = t_1 + t_2$
	$D = (t * S_1 + E_1) * (ts_2 + e_2)$

(P05)

An attacker, Mallory, can intercept the messages between P_1 and P_2. She can modify the first message from P_2 by replacing E_2 with $E_2' = M - t * S_2$ for $M = m * G$, where m is a scalar chosen by her. P_1 will then compute D as:

$$D = (ts_1 + e_1) * (t * S_2 + E_2') \tag{6}$$

Mallory will calculate D according to:

$$\begin{aligned} D &= (ts_1 + e_1) * (t * S_2 + (M - t * S_2)) \\ &= (ts_1 + e_1) * M \\ &= (t * S_1 + E_1) * m \end{aligned} \tag{7}$$

This means that Mallory has tricked P_1 into obtaining a shared secret with her instead of with P_2. This is possible because replacing E_2 with $E_2' = M - t * S_2$ "erases" $t * S_2$ from the expression $t * S_2 + E_2$.

With JKA this exploit is not possible, as Mallory does not have access to t_1, and thus cannot calculate t when intercepting E_2. In other words, she does not know what to use as E_2'. When P_1 sends E_1 to P_2, Mallory has access to both t_1 and t_2, but if she replaces E_1 with $E_1' = M - t * S_1$, P_2 will discover this when verifying h_1. JKA relies on each party committing to its E_i before the ephemeral public key of the other party is known. P_1 commits to E_1 by sending h_1 before knowing E_2. P_2 commits to E_2 by sending E_2 before knowing E_1.

4.2 Security properties of JKA

Let us discuss the security properties of JKA and motivate why JKA is conjectured to satisfy the requirements set in Section 3.2. Uniqueness of the shared secret across different sessions is achieved assuming a good source of random data, which in practice gives unique ephemeral keys for every new session. When it comes to confidentiality, integrity and mutual authentication — the observations stated in Section 3.2 are valid for JKA.

It is interesting to view the Julia expression in its expanded form:

$$\begin{aligned} D &= (ts_1 + e_1)(ts_2 + e_2) * G \\ &= t^2 s_1 s_2 * G + ts_1 e_2 * G + te_1 s_2 * G + e_1 e_2 * G \end{aligned} \tag{8}$$

Note that the three scalar multiplications of Noise XX (Protocol P03), $e_1 e_2 * G$, $e_1 s_2 * G$, $s_1 e_2 * G$, are included in Equation 8. Note also that an attacker cannot compute D even when she has access to both secret keys, s_1 and s_2. The first three terms can be computed by the attacker as $t^2 s_1 s_2 * G$, $ts_1 * E_2$, and $ts_2 * E_1$. However, the fourth term is impossible for the attacker to compute without knowing at least one of e_1 or e_2. Since D cannot be computed by the attacker even when both static secret keys are available to her, JKA achieves forward secrecy. The same argument is valid for backward secrecy.

We have found no way for an attacker to impersonate someone when communicating with P_i even when s_i is known to the attacker. Therefore, JKA is conjectured to protect against compromised-key impersonation.

4.3 Performance of JKA

In this subsection we present the measurements and calculations we conducted to test the performance of JKA. We compared JKA's performance to that of Protocol P03 (Noise XX). The purpose was two fold — to show that a scalar multiplication is an expensive operation (as stated in Section 3.1) and to demonstrate the performance benefit of JKA. The Julia expression is complex and it may not otherwise be obvious that it is faster than doing three scalar multiplications.

The measurements were performed using a Nordic Semiconductor nRF52840 development kit running the nRF5 SDK version 17.0.2. For the mathematical operations we used the finite field arithmetic low-level API provided by the Libsodium cryptographic library, version 1.0.18 [12]. The function calls that we measured are listed in Table 1. SMUL is a scalar multiplication, GADD is the group operation, while IMUL and IADD are integer operations modulus a large prime. The hash operation hashes 32 bytes of data.

Measuring the individual calls to the Libsodium functions gave us the results summarized in Table 2. Please note that the functions have a constant-time implementation. We can see that the scalar multiplication is by far the most costly operation.

Table 1: Libsodium functions

Operation	Function name
SMUL	`crypto_scalarmult_ed25519_noclamp()`
GADD	`crypto_core_ed25519_add()`
IMUL	`crypto_core_ed25519_scalar_mul()`
IADD	`crypto_core_ed25519_scalar_add()`
HASH	`crypto_generichash()`

Table 2: Measurement results

Operation	Cycles	Time (ms)
SMUL	4 068 990	63.58
GADD	463 232	7.24
IMUL	4 219	0.07
IADD	5 331	0.08
HASH	18 734	0.29

Table 3: Calculated results

Handshake	Cycles	Relative
Noise XX (P03) 3 x SMUL	12 206 970	1.000
JKA (P04) 2 x SMUL + GADD + IMUL + 2 x IADD + 2 x HASH	8 653 561	0.709

If we use the results from Table 2 and apply them to Noise XX and JKA, we get the calculated cost for the two handshakes shown in Table 3. We can see that JKA gives a 29 percent reduction in the number of CPU cycles, close to the theoretical maximum of 33 percent (two scalar multiplications instead of three). We can also calculate that the scalar multiplications comprise 94 percent of the total cycles for JKA.

We can extend the reasoning from the handshake phase to the complete key agreement, and include even the pre-handshake. Both Noise XX and JKA have one scalar multiplication in the pre-handshake. JKA has additional hash operations, two for P_1 and one for P_2, but as we have shown, this does not make a significant difference.

4.4 Julia Key Agreement with one scalar multiplication

In this subsection we focus on a flavor of JKA optimized for the case when a party knows beforehand whom it will perform a key agreement with. This allows us to save one scalar multiplication in the handshake.

The protocol is defined as follows:

$$
\begin{array}{ll}
\text{// Pre-handshake} & \\
e_1 = secret() & e_2 = secret() \\
E_1 = e_1 * G & E_2 = e_2 * G \\
t_1 = hash(\text{"}t1\text{"}|E_1) & t_2 = hash(\text{"}t2\text{"}|E_2) \\
h_1 = hash(\text{"}h1\text{"}|E_1) & \\
PC_1 = t_1 * S_2 & PC_2 = t_2 * S_1 \\
 & \\
\text{// Handshake} & \\
 & \xrightarrow{h_1} \\
 & \xleftarrow{E_2} \\
t_2 = hash(\text{"}t2\text{"}|E_2) & \\
D = (t_2 s_1 + e_1) * (PC_1 + E_2) & \\
 & \xrightarrow{E_1, app1} \\
 & Verify : h_1 = hash(\text{"}h1\text{"}|E_1) \\
 & t_1 = hash(\text{"}t1\text{"}|E_1) \\
 & D = (PC_2 + E_1) * (t_1 s_2 + e_2)
\end{array}
$$

(P06)

In this variant, P_1 knows beforehand that it will communicate with P_2. This enables it to compute $t_1 * S_2$ prior to the handshake. Note that the Julia expression D is slightly different than in Protocol P04 — in order to enable the pre-computation, S_2 is multiplied by t_1 instead of $t = t_1 + t_2$. This means that in the handshake phase, P_1 only needs to perform one scalar multiplication.

For clarity, let us write the complete expressions for D using JKA with one scalar multiplication:

$$
\begin{aligned}
D &= (t_2 s_1 + e_1)(t_1 s_2 + e_2) * G \\
&= (t_2 s_1 + e_1) * (t_1 * S_2 + E_2) \\
&= (t_2 * S_1 + E_1) * (t_1 s_2 + e_2)
\end{aligned}
\tag{9}
$$

We consider JKA with one scalar multiplication to be particularly suitable when P_1 is a resource-constrained IoT device and P_2 is its backend server, which is a typical use case for us at ASSA ABLOY. This tackles the limitation of the protocol, the fact that P_1 must know S_2 beforehand. Since P_2 communicates with multiple IoT devices, it does not know S_1 beforehand, and thus has to perform two scalar multiplications in the handshake phase. However, being a server, it has enough computational power so that an additional scalar multiplication does not make a relevant difference.

Please note that the protocol by no means limits P_2 from pre-computing $t_2 * S_1$ if it knows S_1 beforehand. This can be used when two resource-constrained IoT devices that have an ongoing session want to refresh the common secret, which is also a relevant IoT use case.

Pre-computation of a scalar multiplication by P_1 for a future session with a known P_2 is also possible for some other protocols. For example, $e_1 * S_2$ can be pre-computed for Noise XX (Protocol P03). Note that this optimization brings Noise XX from three to two scalar multiplications, while it brings JKA from two scalar multiplications to one, which is a percentually higher saving.

As far as we are aware, Protocol P06 could be used in the general case. This allows a single protocol to support both scenarios, when pre-computation is performed and when it is not. As discussed in Section 3.1, we need to keep code complexity low. Therefore, it is preferred to use a single protocol instead of multiple variants (which we currently have with Protocol P04 and Protocol P06). However, we keep Protocol P04 with $t = t_1 + t_2$ as the baseline version of JKA, and this is what we recommend for general use. The reason is that Protocol P04 has been more thoroughly analyzed than Protocol P06 so far. By using t with contribution from both parties, we make sure neither side can choose it freely.

With protocols P04 and P06, we present two different ways of computing t. Other ways might be feasible as well. For example, $t = t_1 = t_2$ could simply be a random number decided jointly by the parties independent of their ephemeral public keys. The use of $t_1 = hash("t1"|E_1)$ (and likewise for t_2) in Protocol P06 was chosen in favor of a random integer independent of the public ephemeral key. The intuitive reason is that the fact that t cannot be chosen freely gives an attacker less freedom. In other terms, the attack surface is smaller.

4.5 Implementation

In addition to the specification of Julia Key Agreement presented above, we have provided an implementation using SageMath [16]. The code is available in a SageMath worksheet at [10]. It is intended to illustrate the key agreement protocols described in the paper. It is not to be considered as a full implementation, but rather as a complement to the paper and as an invitation for experimentation and further scrutiny of the algorithms.

The code below is a sample from the implementation. It shows how the Julia Key Agreement (Protocol P04) is implemented. Note how a considerable part of the code corresponds well with the notation used in the paper. For example, the code for P_1 to compute the Julia expression is simply: `D1=(t*s1+e1)*(t*S2+E2)`, while the paper notation gives $D_1 = (ts_1 + e_1) * (t * S_2 + E_2)$. After the code has executed, the value of the shared secret computed by P_1 (`D1`) will equal the share secret computed by P_2 (`D2`). This can be tested in SageMath with the command `D1 == D2`.

Listing 1: SageMath implementation of JKA

```
==== Pre-handshake ====
# P1:
e1 = secret()
E1 = e1*G
t1 = hash2(to_bytes("t1") + to_bytes(E1))
h1 = hash1(to_bytes("h1") + to_bytes(E1))

# P2:
e2 = secret()
E2 = e2*G
t2 = hash2(to_bytes("t2") + to_bytes(E2))

# ==== Handshake ====
# P1:
# -> Send h1

# P2:
# <- Send E2

# P1:
t2 = hash2(to_bytes("t1") + to_bytes(E2))
t = t1+t2
D1 = (t*s1+e1)*(t*S2+E2)
# -> Send E1, app1

# P2:
# Verify h1 = hash1(to_bytes("h1") + to_bytes(E1))
t = t1+t2
D2 = (t*S1+E1)*(t*s2+e2)

D1 == D2
```

The code is independent of which mathematical group the operations are performed in. It contains some additional detail compared to the listing of Protocol P04. In particular, note the following:

- The conversion of strings and group elements to byte arrays is explicit with the `to_bytes()` function.
- Two hash functions are used. `hash1()` computes a secure hash of the input and returns the hash digest as a byte array. `hash2()` hashes the input and converts the hash digest to an integer that can be used for scalar multiplication. The protocol listing does not differentiate between these two cases.
- Note that `+` is used in multiple ways: to concatenate byte arrays, to add integers, and for the group operation.
- Note that `*` is used both for ordinary multiplication and for scalar multiplication.

The reader is invited to further explore the code with SageMath.

5 RELATED WORK

The most commonly used secure channel today is Transport Layer Security (TLS) [14]. It is a proposed Internet Engineering Task Force (IETF) standard and it defines multiple key agreement algorithms to choose from. The option most similar to JKA is ephemeral elliptic-curve Diffie–Hellman (TLS_ECDHE). TLS was not developed specifically for embedded systems, but does offer some optimizations for resource-constrained devices in its latest version (TLS 1.3), such as a reduced number of round trips in the handshake and allowing for raw public keys instead of certificates. However, it is still typically considered as a "heavy" protocol and as such it is not suitable for some resource-constrained devices.

Ephemeral Diffie-Hellman Over COSE (EDHOC) [15] is a key agreement protocol being developed specifically for use in IoT devices, and can be viewed as the IoT world's response to TLS being too resource-demanding. One of its design goals is to keep the message sizes small so they can fit into one data frame on radio technologies such as 6TiSCH or NB-IoT. The number of bytes in an EDHOC message can be six times lower compared to using TLS_ECDHE with raw public keys. From the options supported by EDHOC, authentication method 3 (see Figure 4 in [15]) is the most similar to JKA, as both parties use static Diffie-Hellman keys for authentication. EDHOC using authentication method 3 is conceptually the same as Protocol P02. Each party computes two scalar multiplications in the handshake: one between its own static secret

key and the other party's ephemeral public key and one between its own ephemeral secret key and the other party's static public key. Partial forward secrecy is attained. However, if both static secret keys are compromised, forward secrecy is broken and previously recorded session data can be decrypted.

Noise Protocol Framework [13] is a framework for cryptographic protocols based on Diffie-Hellman key agreement. It has attracted significant attention over the last years and has multiple open source implementations. For example, the well-known messaging application WhatsApp uses Noise. We already mentioned one of the patterns included in Noise, namely XX (Protocol P03 in the paper), that achieves the same security properties as JKA, but with three scalar multiplications in the handshake. Compared to EDHOC's authentication method 3 and Protocol P02, Noise XX adds one more scalar multiplication: the multiplication between a party's secret ephemeral key and the public ephemeral key of the other party. This additional operation is what enables full forward secrecy.

We have developed several secure channels at ASSA ABLOY. One of them is Salt Channel [11]. The specification is published openly and it has multiple open source implementations. Salt Channel is a SIGMA type of protocol [9], and it emphasizes efficiency, compactness and simplicity. It has been implemented in a number of programming languages and it runs over a wide set of underlying transport layers including TCP, UDP, Bluetooth Low Energy (BLE), NFC and UART. Its compact implementation comes partly from using only the cryptographic operations that are included in the NaCl cryptography library [4]. EDHOC is similar to Salt Channel in the sense that it provides a simpler and more efficient protocol than TLS. However, when Salt Channel was designed, EDHOC did not exist. Salt Channel does not provide options, to allow for simplicity and interoperability. Mutual authentication is attained using Ed25519 signatures [3]. In total, each party computes four scalar multiplications during the handshake: one between the party's secret ephemeral key and the public ephemeral key of the other party, one to create a signature, and two to verify a signature. This is the typical case of a SIGMA protocol using elliptic-curve cryptography.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have presented Julia Key Agreement (JKA), a protocol that achieves mutual authentication, forward and backward secrecy, and protection from key-compromise impersonation with only two scalar multiplications in the handshake, compared to three scalar multiplications required by key agreement protocols in use today. Since a scalar multiplication is a computationally heavy operation, JKA achieves a reduction both in the duration and the energy consumption of the key agreement, which are notable optimization goals for resource-constrained IoT devices. The main design features of JKA are the Julia expression for computing the shared secret, and the fact that the parties commit to their ephemeral keys before knowing the ephemeral key of the counterpart.

In addition to the baseline version of JKA (Protocol P04), we have also specified an optimization for the case where a communicating party knows beforehand whom it is going to perform a key agreement with, resulting in JKA with only one scalar multiplication in the handshake (Protocol P06). An additional benefit of P06 is that it enables a single protocol implementation both for the scenario where we know beforehand the public key of the counterpart (giving one scalar multiplication in the handshake), and for the scenario where the counterpart is unknown (requiring two scalar multiplications in the handshake). This is made possible by the fact that the scalars used to calculate the shared secret can be computed in the pre-handshake phase. On the other hand, this does somewhat reduce our confidence in the resistance of the protocol against exploits, and more analysis is necessary before we can proclaim P06 as the baseline specification for JKA.

It is notoriously difficult to prove the correctness of a security protocol. Rather, in our experience, after a certain degree of public scrutiny, a protocol is considered secure until an exploit is found to prove otherwise. Even though we at ASSA ABLOY have many proprietary solutions, we firmly believe in publicly specifying cryptographic protocols. The more use and probing a protocol has been exposed to, the bigger the confidence that it is secure. With the public specification of the two flavors of JKA and the accompanying implementation, we invite for further inspection of the protocols. We are continuing our own analysis and plan to verify the protocols using a tool for automatic reasoning with respect to security properties. State of the art verification tools include, for example, ProVerif [6] and Tamarin Prover [1]. Formal verification will also enable us to reason more precisely about the different forms of the Julia expression. As mentioned earlier, we have suitable industrial cases where we can use JKA after it has received formal verification and public inspection.

ACKNOWLEDGMENTS

This work was funded by ASSA ABLOY AB, through its central research funding. The work was done by the Pre-product innovation division within ASSA ABLOY's common research department called Global Technology Team (GTT). We would like to thank Jakub Kliwison for helping us with implementing the measurements to compare the performance of JKA and Noise XX.

REFERENCES

[1] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. 2017. Symbolically Analyzing Security Protocols Using Tamarin. *ACM SIGLOG News* 4, 4 (2017), 19–30. https://doi.org/10.1145/3157831.3157835

[2] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–228.

[3] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2011. High-Speed High-Security Signatures. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, Bart Preneel and Tsuyoshi Takagi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–142.

[4] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2011. NaCl: Networking and Cryptography library. Retrieved May 18, 2021 from https://nacl.cr.yp.to

[5] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. In *Progress in Cryptology – LATINCRYPT 2012*, Alejandro Hevia and Gregory Neven (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–176.

[6] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends in Privacy and Security* 1, 1-2 (2016), 1–135. https://doi.org/10.1561/3300000004

[7] Youssou Faye, Hervé Guyennet, and Ibrahima Niang. 2019. A Survey of Fast Scalar Multiplication on Elliptic Curve Cryptography for Lightweight Embedded Devices. In *Modern Cryptography*, Menachem Domb (Ed.). IntechOpen, Chapter 3.

[8] Cameron F. Kerry and Charles Romine. 2013. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS).

[9] Hugo Krawczyk. 2003. SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols. In *Advances in Cryptology -*

CRYPTO 2003, Dan Boneh (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 400–425.

[10] Frans Lundberg and Juraj Feljan. 2021. Julia Key Agreement implementation in SageMath. Retrieved May 12, 2021 from https://github.com/assaabloy-ppi/julia-in-sage

[11] Frans Lundberg and Simon Johansson. 2019. Salt Channel. Retrieved May 18, 2021 from https://github.com/assaabloy-ppi/salt-channel

[12] Open source community. 2013. Libsodium. Retrieved May 14, 2021 from https://github.com/jedisct1/libsodium

[13] Trevor Perrin. 2018. The Noise Protocol Framework, Revision 34. Retrieved May 18, 2021 from https://noiseprotocol.org/noise.html

[14] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. Retrieved May 18, 2021 from https://tools.ietf.org/html/rfc8446

[15] Göran Selander, John Preuß Mattsson, and Francesca Palombini. 2021. Ephemeral Diffie-Hellman Over COSE (EDHOC). Retrieved May 25, 2021 from https://datatracker.ietf.org/doc/html/draft-ietf-lake-edhoc-07

[16] The Sage Developers. 2021. *SageMath, the Sage Mathematics Software System (Version 9.2)*. Retrieved May 18, 2021 from https://www.sagemath.org