ASAP: Algorithm Substitution Attacks on Cryptographic Protocols

Sebastian Berndt*, Jan Wichelmann†, Claudius Pott‡, Tim-Henrik Traving§, and Thomas Eisenbarth¶

Universität zu Lübeck

The security of digital communication relies on few cryptographic protocols that are used to protect internet traffic, from web sessions to instant messaging. These protocols and the cryptographic primitives they rely on have been extensively studied and are considered secure.Yet, sophisticated attackers are often able to bypass rather than break security mechanisms. Kleptography or algorithm substitution attacks (ASA) describe techniques to place backdoors right into cryptographic primitives. While highly relevant as a building block, we show that the real danger of ASAs is their use in cryptographic protocols. In fact, we show that a highly desirable security property of these protocols—forward secrecy—implies the applicability of ASAs. We then analyze the application of ASAs in three widely used protocols: TLS, WireGuard, and Signal. We show that these protocols can be easily subverted by carefully placing ASAs. Our analysis shows that careful design of ASAs makes detection unlikely while leaking long-term secrets within a few messages in the case of TLS and WireGuard, allowing impersonation attacks. In contrast, Signal's double-ratchet protocol shows high immunity to ASAs, as the leakage requires much more messages. But once Signal's long-term key is leaked, the security of the Signal messenger is completely subverted by our attack due to unfortunate choices in the implementation of Signal's multi-device support.

*s.berndt@uni-luebeck.de
†j.wichelmann@uni-luebeck.de
‡c.pott@uni-luebeck.de
§timhenrik.traving@student.uni-luebeck.de
¶thomas.eisenbarth@uni-luebeck.de

1. Introduction

In the past few years, the widespread use of cryptography to protect digital communication has become the norm. More than 90% of web traffic are now end-to-end encrypted via protocols such as TLS for synchronous web sessions or the popular Signal protocol for asynchronous instant messaging. Besides increased awareness and new privacy laws requiring better protection, leaks like the Snowden revelations showed that government agencies are heavily invested in eavesdropping and intercepting web traffic. To overcome cryptographic protection, agencies do not only apply cryptanalytic techniques, but also circumvent cryptosystems. More recently, such knowledge has become available to less well-funded governments and police forces, which have been reported to even bypass highly protected messenger communications of Telegram [7] and EncroChat [15]. While the latter examples rely on exploitation of implementation bugs and phishing, Snowden's documents also revealed efforts to achieve longer term access. One class of attacks tries to manipulate the algorithms used in implementations. The main idea behind this manipulation is the injection of a *backdoor* into otherwise secure implementations, e.g. via the compiler, as proposed by Ken Thompson and later observed as XCodeGhost [48]. Even cryptographically secure algorithms can be subverted, e.g. through the manipulation of standardization processes (like the issues surrounding the Dual_EC_DRBG number generator [10, 41, 42]).

A formal treatment of these manipulations was first given by Young and Yung under the name *kleptography* [49, 50]. The recent developments started by Snowden's publication reignited the interest in this kind of attacks, starting with the work of Bellare, Paterson, and Rogaway [6], that studied the attacks under the name of *algorithm substitution attacks* (ASA).

1.1. Our Contributions

Instead of focusing on a single algorithm, we extend the study of ASA to cryptographic protocols. We first formally define an appropriate notion of such attacks and then prove that an important, widespread property of modern protocols — *forward secrecy* — directly implies the vulnerability against such attacks. We then focus on three concrete protocols — TLS, WireGuard, and Signal — and show that all of these protocols have multiple vulnerabilities against ASA. In two of these protocols, TLS and WireGuard, these vulnerabilities can be used to leak the long-term secret key with at most four messages, which allows us to perform Man-in-the-Middle attacks.

Leaking the long-term secret key from Signal takes more messages, but we show that the leak of this key has dramatic consequences: We are able to completely circumvent the forward secrecy of the protocol, by using Signal's multi-device option to register a new device that is also able to access messages of an already established communication. While the theoretical vulnerability to such an attack on Signal was already proposed in 2016 by Cohn-Gordon, Cremers, and Garratt [13], we show that this attack is indeed feasible: The Signal messenger implements the "subtle details" mentioned by Cohn-Gordon et al. [12] in such a way that an attacker knowing the long-term key is able to

easily manipulate the protocol to completely invalidate Signal's end-to-end encryption. Our finding highlights that even protocols that could have a high resistance to ASA and are widely considered secure, fail due to the complexity of their implementation, making ASA a practical threat.

In order to show that these vulnerabilities are not only theoretical, we modified the implementation of these protocols in OpenSSL (TLS), the Linux kernel (WireGuard), and the Signal desktop client. We experimentally verified that these modified implementations are able to leak the long-term keys with minimal computational overhead and only a few changed lines of code, making the attacks hardly detectable.

1.2. Related Work

As described above, the concept of algorithm substitution attacks was first formalized by Young and Yung under the name *kleptography* [49, 50]. The current name of algorithm substitution attacks was proposed by Bellare, Paterson, and Rogaway, who also presented several attacks on certain symmetric encryption schemes [6]. Degabriele, Farshim, and Poettering criticised this model as it relied on the assumption that all ciphertexts produced by the subverted algorithm must be valid [16]. The model of Bellare, Paterson, and Rogaway was extended to signature schemes by Ateniese, Magri, and Venturi [3]. Bellare, Jaeger, and Kane strengthened the result of Bellare, Paterson, and Rogaway by showing the proposed attacks can be made stateless [4]. Berndt and Liśkiewicz showed that algorithm substitution attacks can be interpreted as steganographic systems, which allowed them to generalize the above results and give upper bounds for the number of information embeddable in a single message via black-box attacks [8]. Just recently, Chen et al. showed that this upper bound can indeed be beaten via non-black-box attacks against certain key encapsulation mechanisms [11]. As the authors only focus on algorithms and aim to replace the encapsulation algorithm, they can only embed the (often short-lived) session key in their algorithms, as this is the only sensitive information that the encapsulation algorithm can access. Furthermore, they also introduced asymmetric algorithm substitution attacks that use asymmetric keys.

There is a wide literature about countermeasures against ASA: the split-program methodology that immunizes the randomness generation [39]; the use of purely deterministic primitives [6]; cryptographic reverse firewalls, that re-randomize all outgoing communications [30]; self-guarading mechanisms, which contain an untamperable initial first phase [22]; backdoored pseudorandom generators that add a salt to the pseudorandom generator [19].

2. Preliminaries

In the following, we fix the notations used in this work, introduce the notion of algorithm substitution attacks against protocols, and show that a widely used property of cryptographic protocols — *forward secrecy* — always leads to vulnerabilities against such substitution attacks. We often consider randomized algorithms R and for fixed ran-

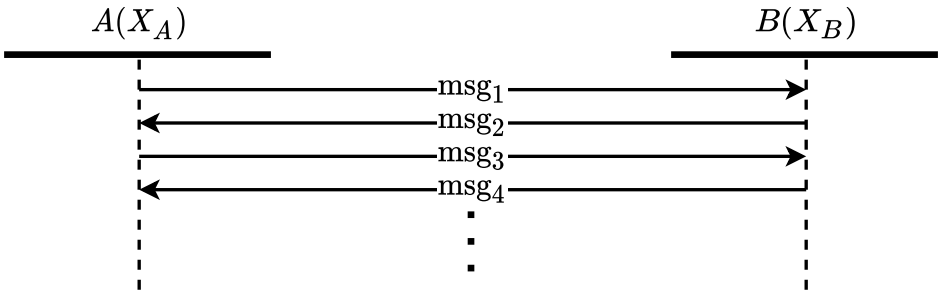

Figure 1: A run of $\Pi_{A,B}$ on X_A and X_B.

domness r, we denote their deterministic output on input x as $R(x;r)$. We also define $\log(x) := \lceil \log_2(x) \rceil$.

2.1. Cryptographic Protocols

Let A and B be two randomized algorithms, called *parties*. In a *protocol* $\Pi_{A,B}$, both of these algorithms exchange messages back and forth. More formally, such a protocol is described by a sequence of messages $\mathsf{msg}_1, \mathsf{msg}_2, \ldots$, where messages with odd indices are sent from A to B and messages with even indices are sent from B to A. The *run* of a protocol $\Pi_{A,B}$ on inputs X_A and X_B is described by such a sequence of messages, with $\mathsf{msg}_i = P(X_P, \mathsf{msg}_1, \ldots, \mathsf{msg}_{i-1})$, where $P = A$ for odd i and $P = B$ for even i (see Fig. 1). We denote this sequence by $\Pi_{A,B}(X_A, X_B)$. Usually such protocols are used to transfer information between A and B in a secure manner. For example, in a *zero-knowledge protocol*, A wants to convince B that $f(X_A) = 1$ for some public function f without actually revealing the input X_A. Protocols play an essential part in cryptography and are widely used. Nowadays, nearly 90% of the internet traffic is encrypted via such protocols [28].

To distinguish between symmetric keys and asymmetric keys, we denote symmetric keys with lower case letters and asymmetric keys with upper case letters. Furthermore, for an asymmetric key-pair K, we denote the secret key by $\mathrm{sec}(K)$ and the public key by $\mathrm{pub}(K)$.

Adversaries trying to attack such a cryptographic protocol are typically characterized by their abilities. A *passive adversary* or *eavesdroppper* can only listen to the communication between the parties. In contrast, an *active attacker* can directly interact with the parties. For example, an active attacker can impersonate A and convince B to share sensitive information.

2.2. Algorithm Substitution Attacks

In this work, we consider algorithm substitution attacks against cryptographic protocols. As noted above, all previous work concentrated on the replacement of a single algorithm from a cryptographic primitive. In the work of Chen et al., this led to the problem that

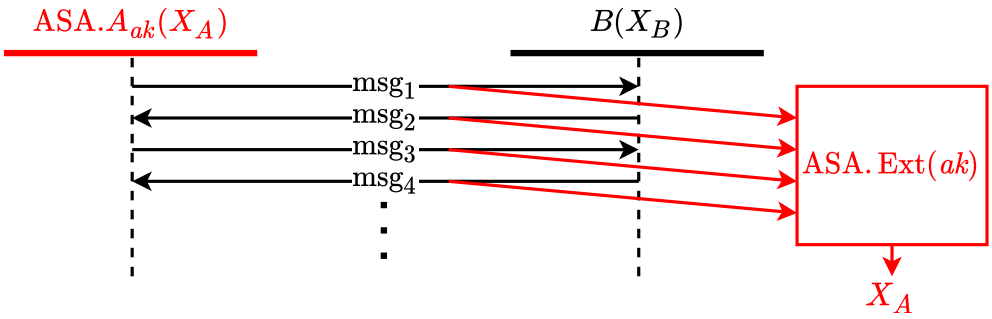

Figure 2: A run of the ASA

they were only able to leak session keys in this model [11]. But primitives are usually embedded in protocols and usually, *both* parties of a key encapsulation mechanism have some secret. Hence, in this scenario, an attack against the protocol would allow a subverted encapsulation algorithm to leak the long-term secret key of a party instead of only the short-term session key.

Usually, at least one of the inputs X_A or X_B of a cryptographic protocol contains some value that is to be kept secret. The goal of an algorithm substitution attack against $\Pi_{A,B}$ is to manipulate one of the parties such that the run of the protocol leaks this secret to an observer. For everyone but this observer, the run of the manipulated protocol should be indistinguishable from the original protocol. For the sake of simplicity, we only give a formal definition of an algorithm substitution attack against party A, but the adaption against party B is straightforward. An *algorithm substitution attack* ASA against $\Pi_{A,B}$ is a tuple of randomized algorithms $(\mathsf{ASA}.A, \mathsf{ASA.Ext})$. In addition to the inputs given to A, the algorithm $\mathsf{ASA}.A$ is also given an *attacker key* ak. To separate the input in the protocol from this key, we write $\mathsf{ASA}.A_{ak}$ to denote that $\mathsf{ASA}.A$ has knowledge about ak. The *extraction algorithm* $\mathsf{ASA.Ext}$ is also given this attacker key and the sequence of messages $\mathsf{msg}_1, \mathsf{msg}_2, \ldots$ and tries to extract X_A from these messages.

We say that ASA is *undetectable*, if no probabilistic polynomial-time algorithm Dist is able to distinguish $A(z_1), A(z_2), \ldots$ from $\mathsf{ASA}.A_{ak}(z_1), \mathsf{ASA}.A_{ak}(z_2), \ldots$ with non-negligible probability upon random choice of ak, even if Dist is allowed to choose the inputs z_i. Furthermore, ASA is *reliable*, if the probability that $\mathsf{ASA.Ext}(ak, \mathsf{msg}_1, \mathsf{msg}_2, \ldots) \neq X_A$ is negligible, where $\Pi_{\mathsf{ASA}.A_{ak},B}(X_A, X_B) = \mathsf{msg}_1, \mathsf{msg}_2, \ldots$ and the probability is taken over the random choice of ak and the internal randomness of the algorithms. Throughout this work, κ denotes the key length of the attacker key ak.

Bellare et al. showed the existence of a black-box (or universal) algorithm substitution attack, i. e. an algorithm substitution attack, based on rejection sampling, that works against every randomized symmetric encryption scheme [5]. Berndt and Liśkiewicz were able to show that this substitution attack can be used against every randomized algorithm with sufficient high min-entropy [8].

Theorem 1 (Theorem 7.1 in [8])**.** *Let R be any randomized algorithm that has a secret input s. If pseudorandom functions exist, there exists a generic algorithm substitution*

attack ASA *against* R *such that* ASA *is secure and reliable as long as the secret input* s *to* R *is sufficiently long and the min-entropy of* R *is sufficiently high.*

Universal ASA The general idea behind this universal ASA is the following: Suppose that we want to embed λ bits of the secret s in the output of the randomized algorithm R on input x. Let s_i be the i-th block of s of length λ, i. e. $s = s_1 \mid\mid s_2 \mid\mid \ldots s_L$ with $L = |s|/\lambda$. We assume that we have access to a pseudorandom function F that — equipped with key ak — on some input y outputs a pair (b, i), where $|b| = \lambda$ and $|i| = \log(L)$. The universal ASA now samples random bits r until it finds $r^\star$ with $F_{ak}(R(x; r^\star)) = (s_i, i)$, i. e. until the pseudorandom function outputs the i-th block of s and the index i for some i. A common target for our attack in this work is the Diffie-Hellman key exchange and we thus illustrate the attack here for clarity. Given a finite cyclic group $\mathbb{G}$ of order n and some generator $g \in \mathbb{G}$, a non-subverted party generates $1 < a < n$, keeps a secret and sends g^a to the other party. Our goal is now to embed information about some secret s into g^a. Hence, to embed a single bit ($\lambda = 1$) in the universal attack, we sample $a \leftarrow \{2, \ldots, n-1\}$ until we obtain $a^\star$ such that $F_{ak}(g^{a^\star}) = (s[i], i)$, where $s[i]$ is the i-th bit of the secret s.

It can easily be seen that the probability that we do not find a suitable candidate after $2^\lambda \cdot \kappa$ samples of random bits is negligible in κ. Using the analysis of the *coupon collector problem*, it is easy to see that $O(L \cdot \log(L))$ received ciphertexts are sufficient to reconstruct s with high probability. In fact, for sufficiently large values of L, the probability that more than $L\ln(L) + \beta L$ samples are needed is at most $1 - \exp(-\exp(-\beta))$ [31, Theorem 5.13]. For example, the probability that the number of samples is within $[L\ln(L) - 4L, L\ln(L) + 4L]$ is at least 0.98.

Optimizing the universal ASA: As the probability that no fitting block (s_i, i) is found is negligible in κ, there is no need to deal with embedding errors on the side of the extractor. By only trying at most $2^\lambda \cdot \gamma$ samples, an embedding error occurs with probability $2^{-\gamma}$. Adding error correction to the extractor would thus allow us to reduce the number of samples. For example, setting $\gamma = 3$ yields an error probability of at most $1/8$. The extractor could now use a simple majority rule whenever they encounter multiple blocks $\{(s^{(j)}, i)\}_j$, where not all $s^{(j)}$ are identical. Another optimization that can be used to save bandwidth is the fact that the secrets s that we want to leak are often highly structured and contain some redudancy. For example, if we aim to leak long-term RSA keys, it is sufficient to leak only half of the bits of one of the prime factors p or q, as the remaining bits can be reconstructed via the lattice algorithm of Coppersmith [14, 23].

Modified IV-replacement ASA Another important non-universal algorithm substitution attack can be applied, whenever a uniformly random string is transferred. This is often the case, if both parties communicate via a symmetric encryption scheme that requires a random nonce or IV. The modified IV-replacement attack is a simple adaption of the *IV-replacement* attack described by Bellare et al. in [6]. For the sake of simplicity, we assume in the following that the complete message msg_i is a uniformly random string (otherwise, we only apply the technique on the nonce-part). Again, if we want

to embed λ bits of the secret s, split into $s_1 \,||\, s_2 \,||\, \ldots \,||\, s_L$, we first choose a string r of length $|\,\mathsf{msg}_i\,| - \lambda$ completely random. This string r encodes some index j consisting of $\log(L)$ bits (e. g. via its most significant bits or its least significant bits). Then, we compute $\mathsf{msg}'_i = r \,||\, (F_{ak}(r) \oplus s_j)$ via a pseudorandom function F and output msg'_i. The extractor, knowing λ and ak, can easily compute both j and s_j from msg'_i. As described above, about $O(L \cdot \log(L))$ samples are needed to reconstruct the secret s. The main advantage of this attack is that no repeated sampling is necessary, i. e. the running time of the protocol is only increased by a single call to the pseudorandom function F. The optimal value for λ here depends on the length of msg_i, but must be small enough that the length of r is sufficiently high to avoid detectability.

Passive and Active Attacks While the deployment of the modified algorithm (i. e. exchanging A with $\mathsf{ASA}.A$) is an active attack, the extraction of the information via $\mathsf{ASA.Ext}$ is done in a purely passive way. In this work, we do not consider how to deploy the modified algorithm $\mathsf{ASA}.A$, but focus on the possible places where such a substitution attack might be used in *cryptographic protocols*. Furthermore, we also study how the long-term keys can be used after extraction, both by a passive attacker and an active attacker.

2.3. ASA and Forward Secrecy

An important feature of modern cryptographic protocols is called *forward secrecy*, which concerns encrypted communication. Informally, this means that a breach of the long-term keys is not sufficient for the decryption of the messages encrypted before the breach. In our setting introduced above, some of the messages msg_i are encryptions, i. e. $\mathsf{msg}_i = \mathsf{Enc}(k_i, p_i)$ for some plaintext p_i, some key k_i and a symmetric encryption scheme Enc. Forward secrecy now means that an attacker that knows the secret inputs X_A and X_B of the parties and all of the msg_i (but neither k_i nor p_i) cannot distinguish the key k_i from a random key [24].[1]

A common way to enable this forward secrecy is to only use *ephemeral keys* for some part of the communication. For example, in TLS, a handshake starting a session consists of a key exchange of such an ephemeral key. At the end of the session these ephemeral keys are completely discarded and the next communication between the parties starts a new session with new ephemeral keys.

Lemma 1. *If $\Pi_{A,B}$ is a cryptographic protocol with forward secrecy, then there is an ASA against A or B.*

Proof. Consider the earliest message $\mathsf{msg}_i = \mathsf{Enc}(k_i, p_i)$, where some attacker cannot distinguish k_i from a random key. As the attacker knows both X_A and X_B, the key k_i is not part of these secrets. Hence, the communication between the parties must include some messages $\mathsf{msg}_j, \mathsf{msg}_{j+1}, \ldots$ where this key k_i is exchanged. If all of the messages $\mathsf{msg}_1, \mathsf{msg}_2, \ldots, \mathsf{msg}_i$ are chosen deterministically, the attacker can simulate this with

[1] Note that this notion is called *weak forward secrecy* in [24], as it does not prevent active attacks.

their knowledge of X_A and X_B and can thus also reconstruct the key k_i. Therefore, there is a message $\mathsf{msg}_{i^\star}$ which is chosen randomly. Furthermore, as the protocol has forward secrecy, the attacker is not able to brute-force all possible choices of randomness for the construction of $\mathsf{msg}_{i^\star}$. Hence, the requirements for Theorem 1 are fulfilled and we can embed information about X_A in $\mathsf{msg}_{i^\star}$. □

3. Substitution Attacks against Protocols

In light of Lemma 1, we looked at multiple widely used protocols that support forward secrecy and analyzed their vulnerability with regard to algorithm substitution attacks. In addition to the attacks on the forward secrecy parts of the protocols, we also discovered that TLS 1.2 and WireGuard are vulnerable to IV-replacement attacks.

3.1. TLS

The Transport Layer Security (TLS) protocol is arguably one of the most important protocols for secure communication, providing encryption, integrity protection and authenticity confirmation. TLS is located between the application-layer and transport-layer of the Internet Protocol (IP) Stack. It provides a transparent way of secure communication to the application-layer. In general, TLS supports a large number of different algorithms used for key exchange, signatures, and encryption.

The protocol is split into two (main) layers: The Handshake layer (composed by the Handshake-, Change Cipher Spec-, and Alert Protocols), which initiates a connection between the parties, and the Record layer (formed by the Record Protocol), which is used to send application data. Usually, TLS is used for communication between a server S and a client C. The protocol distinguishes between *sessions*, which are an association between the parties with some state that specifies the used algorithms, and *connections*, which are secure streams within a session. The most current version of TLS is TLS 1.3, but it is currently only supported by 40% of servers. Its predecessor, TLS 1.2, is supported by nearly 99% of the servers [35].

We first introduce the two protocol layers of TLS 1.3, highlight the differences to TLS 1.2, and discuss possible targets for ASA afterwards.

Handshake layer [36] To initiate a session, a handshake between the parties is performed. The server S has a certificate $CERT_S$ to authenticate itself. The client C sends a "Hello" message to the server which includes a list of preferred algorithms and a random string r_C of length 32 bytes. It also guesses, which key exchange algorithm will be chosen by the server, generates an appropriate ephemeral key EK_C and send its public component $\mathrm{pub}(EK_C)$ to the server. If C tries to resume a session, it also send a session ID. The server S chooses the algorithms to be used from the client's list, a random string r_S of length 32 bytes, and an ephemeral key EK_S and sends all of this as "Hello" message to the client. If the client wanted to resume a session, S can also send a session ID. Both parties then derive a handshake secret hs from the ephemeral keys EK_C and

EK_S and from a hash of both "Hello" messages via HKDF [25]. From *hs*, both parties also derive the handshake traffic key *htk* via HKDF. Then, the server computes a hash of the current communication and signs it using its certificate. The server S encrypts this signature and the public key of the certificate via *htk* and sends it to the client. The client decrypts these values and verifies the signature. Both parties then derive several other symmetric secrets/keys from *hs* via HKDF: the finished key *fk*, the master secret *ms*, the traffic secret *ts*, and the traffic key *tk*. The client now computes a MAC of the current transcript via *fk* and encrypts this with *htk* and sends this "Finish" message to the server. The server then also computes such a MAC in the same way and sends this encrypted with *htk* as "Finish" message to the client. A more detailed description of the handshake can be found in the work of Diemert and Jager [17].

In TLS 1.2 [18], the client does not guess the key exchange algorithm, causing the need for another roundtrip between the parties. Furthermore, the handshake messages are not encrypted, i. e. *htk* is not present in the protocol.

Record layer [36] In the record layer, the application data is encrypted via the symmetric traffic key *tk*. The encryption is performed via an authenticated encryption with associated data (AEAD) encryption scheme [37, 38] chosen during the handshake, which is either AES GCM, AES CCM, or ChaCha20-Poly1305 [27, 34]. The nonce needed for this AEAD is produced by XOR-ing the sequence number (describing how many messages were already sent) and an initialization vector derived from the master secret *ms*.

In contrast, TLS 1.2 allows a much wider range of encryption schemes, not only AEADs. Also, the nonce used for the schemes is constructed differently. It consists of an initialization vector derived from the master secret *ms* concatenated with an *explicit initialization vector (eIV)*, which is chosen randomly and transmitted. In earlier versions of TLS, no eIV was used, which led to vulnerabilities [32].

3.1.1. Security Analysis w.r.t. Substitution Attacks

In the following, we identify possible attack vectors against both versions of TLS. The first thing to consider is which key-material we want to leak. By leaking the short-term keys (either $\mathrm{sec}(EK_S)$, $\mathrm{sec}(EK_C)$, or *hs* and the derived keys), an attacker is able to decrypt the complete communication within a single connection. Usually, the only long-term key that exists is the certificate of the server, allowing an attacker to impersonate the server and perform Man-in-the-Middle attacks. Note that the short-term keys are only derived from later messages in the Handshake layer. Hence, attacks on the early messages in the Handshake layer can only leak long-term keys.

Leaking in the Handshake Layer The TLS handshake consists of different messages that may or may not be sent when initiating a connection. These messages may or may not be encrypted. These factors vary between the specific setup, as well as TLS versions. But one message always contains a nonce, is always unencrypted, and will always be sent: The "Hello" message. This nonce can be chosen in a way that reveals information without compromising the subsequent calculations. As the "Hello" messages

are unencrypted, they allow an IV-replacement ASA, resulting in a setup independent attack across all versions of TLS.

Furthermore, the key shares pub(EK_C) and pub(EK_S) are also transmitted in the clear. As both of these shares are public keys, we cannot simply replace them by random strings (as obtaining the corresponding private keys would be very expensive), but the universal ASA can be used to repeatedly sample EK until the public key contains some information.

Leaking in the Record layer In TLS 1.3, all communication deterministically depends on the master secret *ms*. As deterministic algorithms cannot be used for algorithm substitution attacks [8], no attack vector seems to exist against the Record layer of TLS 1.3.

In contrast, the usage of the explicit IV in the Record layer of TLS 1.2 introduces randomness, which we can use for an attack. As described above, the nonce used in the encryption scheme is split in two halves: the *implicit IV* or *static IV* (sIV) and the *explicit IV* (eIV). The sIV is usually a session number, which is known to both client and server. As both parties know the sIV, there is no need to transmit this part. The eIV is randomly chosen and transmitted with the ciphertext. The other party does not know this random number and has no way to derive it from previously shared knowledge. Therefore it is necessary to transmit the eIV with the ciphertext, but the eIV is not necessarily transmitted in the clear (see below).

3.2. WireGuard

WireGuard is a Virtual Private Network (VPN) solution that has recently been added to the Linux kernel [20]. It is becoming increasingly popular due its simple design and implementation, especially compared to other widely used protocols, and it uses generally acknowledged and fast algorithms. For example, all public-key operations are performed on Curve25519, all hashes are computed via blake2s, all keys are derived via HKDF, and the symmetric authenticated encryption schemes used are either ChaCha20Poly1305 or XChaCha20Poly1305 [40, 26, 25, 33]. As usual for VPNs, WireGuard allows peers to communicate with each other in a secure manner. A central part in the design and one reason for the simplicity of the protocol is that each peer is identified only by its static asymmetric key pair. Before they can create a connection, both peers have to manually share their static public key via a secure channel, such that they can prove that each is communicating with the correct party.

Again we give an overview of the protocol and afterwards discuss potential targets for ASA.

Handshake The WireGuard protocol does not distinguish between clients and servers, however, to distinguish between the two parties one is called the *initiator* I and the other one the *responder* R. Both parties have an asymmetric static key SK_I, resp. SK_R. To communicate, I also needs to know pub(SK_R) and R needs pub(SK_I). In addition, I needs an IP address of R in order to send the initial message. As a first step, I generates

an ephemeral key EK_I and computes a symmetric handshake key hsk by performing a Diffie-Hellman key exchange on pub(SK_R) and sec(EK_I) and a second symmetric handshake key hsk' by performing a Diffie-Hellman key exchange on pub(SK_R) and sec(SK_I). Then, it initiates the connection via the handshake initiation message. This message contains among other things the public key pub(EK_I), an encryption of pub(SK_I) using hsk, a random string r_I consisting of 4 bytes used for the session ID, and a timestamp encrypted with hsk'. All encryptions are AEADs and the authenticated data are hashes of the currently computed values that will be given to R. The responder uses pub(EK_I) and sec(SK_R) to also derive the symmetric handshake key hsk and another key exchange on pub(SK_I) and sec(SK_R) to derive hsk'. It then decrypts the encrypted messages and verifies them. Afterwards, R generates an ephemeral key EK_R and derives another symmetric handshake key hsk'' from key exchanges on the pairs (pub(EK_I), sec(EK_R)) and (pub(SK_I), sec(EK_R)). They then send a message containing pub(EK_R), a random string r_R of 4 bytes used for the session ID, the string r_I, and an encryption of the the empty string with hsk''. When these messages have been exchanged, both parties derive their transport data keys tdk_I and tdk_R (one for sending and one for receiving) from the ephemeral keys EK_I and EK_R and the handshake is complete.

Transport Data In the following, every message sent between I and R contains a counter used as a nonce and an encryption of the application data either with tdk_I (if I sends a message to R) or tdk_R (if R sends a message to I).

Denial-of-Service Protection To avoid that a malicious party performs a Denial-of-Service (DoS) attack by abusing the CPU-intensive asymmetric cryptographic operations of a handshake, WireGuard introduces a special *cookie* mechanism: If a peer (initiator or responder) P receives a handshake packet with a valid MAC, but currently cannot perform the necessary elliptic curve computations due to being under load, it returns a *cookie message*. This message contains a randomly generated nonce rn of 24 bytes, which is used alongside the peer's public key pub(SK_P) to encrypt a secret number s, that is randomly generated by P every two minutes. The other peer P' decrypts the cookie, and waits until the initiator's internal rekey timeout has passed. During the ensuing restarted handshake, P' sends their handshake message along with an additional MAC, using the secret number s as the MAC key. The peer P then checks whether that MAC is valid, and if it is, continues or completes the handshake. Note that the random nonce rn is transported in the clear.

3.2.1. Security Analysis w.r.t. Substitution Attacks

As discussed above, the identity of a peer P is given by an asymmetric key pair SK_P. The 256-bit private key sec(SK_P) is therefore the most valuable target for attackers, as it allows stealing the identity of the victim: An attacker who has obtained sec(SK_P) can perform Man-in-the-Middle attacks or impersonate the victim, thus gaining access to a formerly secure VPN. The public keys of other peers are designed to be kept secret within a VPN, and are only accessible when one is able to decrypt a handshake initiation packet;

if a peer I sends a handshake initiation packet to the attacker, the attacker can use $\text{sec}(SK_R)$ to decrypt the packet and obtain $\text{pub}(SK_I)$, possibly enabling other attacks. Since only the handshake initiation message contains an encryption of the public key of the initiator I, leaking the secret key of a responder R allows an attacker to also collect public keys. If a victim on the other hand only acts as initiator, no public keys of other peers in the VPN can be decrypted by an attacker. In this case one could additionally leak the public key of the responder, thus enabling an attacker to connect to the VPN as I.

Another possible target are the symmetric short-term transport keys: If, for example, the attacker obtains tdk_I, they may decrypt all messages sent from I to R, until a new handshake occurs. In order to be able to decrypt the entire communication between I and R over multiple sessions, the attacker needs a way to leak both transport keys tdk_I and tdk_R within one session. Alternatively this could be achieved by leaking the private static key $\text{pub}(SK_P)$ of the victim once and then leaking the private ephemeral key within each session, thus enabling the attacker to compute both transport keys.

WireGuard presents three opportunities for embedding data into randomly generated values: The ephemeral keys $\text{pub}(EK_P)$ exchanged during the handshake, the random session IDs r_P, and the nonce rn of the cookie message.

Leaking via Handshake Messages WireGuard's handshake messages have two sources of randomness: The session IDs r_I or r_R, and the public ephemeral keys EK_I or EK_R. The handshake messages are suitable to leak long-term secrets such as the static private key of the transmitting party. Unfortunately, the short-term secrets are refreshed every few minutes via the handshakes and their leakage in these handshake messages is thus not feasible.

The session IDs of both I and R have a length of 4 bytes and are chosen uniformly at random for each handshake, thus the attacker could perform a modified IV-replacement ASA. However, this attack may be easily detectable due to the short length of the session IDs: If an attacker decides to embed one byte of secret data, the number of possible session IDs decreases to 2^{24}. Since handshakes are designed to be executed every few minutes, one can thus expect a collision within a few days, as opposed to several months.

The public ephemeral keys can be used to conduct an universal ASA, by sampling random private keys until the resulting public key contains the desired secret. Since this approach requires repeated elliptic curve computations, it is quite expensive to embed more than a few bits per handshake. Long delays may get noticed by the user, due to slow connection establishment or frequent connection losses.

Leaking via Cookie Messages The nonce rn used in the cookie messages is chosen uniformly at random and transmitted in plaintext. Since its length of 24 bytes is quite high, the nonce is well-suited for a modified IV-replacement attack. By generating the first 8 bytes (64 bits) randomly, detecting the attack becomes practically infeasible. This leaves 16 bytes (128 bits) for payload, which means that one half of a 256-bit key can be leaked in a single message.

This approach allows leaking long-term secrets as well as previous short-term secrets, since cookie messages are only sent prior to completing a handshake. Also, cookie messages are only sent when a peer is under load.

3.3. Signal

The *Signal* (formerly *Axolotl*) protocol [43] provides end-to-end encryption for text messages and multimedia files. It is widely used in different communication applications such as WhatsApp [47], Skype [29] and the Signal messenger itself. The protocol is based on the Double Ratchet algorithm and uses a triple Elliptic-curve Diffie–Hellman handshake (X3DH) to initiate new conversations. The Sesame protocol is used to enable multi-device support. Signal uses a number of cryptographic primitives including

- Elliptic Curve Diffie-Hellman functions (implemented by X25519 or X448 [26]);
- a signature scheme called XEdDSA producing EdDSA-compatible signatures from X25519 or X448 using the hash function SHA-512 [43];
- a hash function (implemented by SHA-256 / -512);
- a key derivation function KDF based on the HKDF algorithm [25];
- an authenticated encryption (AEAD) scheme [37, 38]. Concretely, KDF is used to produce an encryption key, an authentication key, and an initialization vector (IV). The plaintext is then encrypted with AES-256 in CBC mode.Finally, HMAC with the hash function and the authentication key is used on the authenticated data.

We explain the three protocol parts and discuss possible targets for ASA attacks afterwards.

X3DH [44] Every user in the Signal protocol has an *identity key IK*. These keys are long-term keys and are needed to setup the initial communication between two parties. All of the public keys are stored on the central server. Furthermore, in order to enable an initial communication even if one of the parties is offline, all parties store a signed prekey pub(SPK) along with its signature and a set of one-time prekeys pub($OPK^{(1)}$), pub($OPK^{(2)}$), ... on the server. If party A now wants to initialize communication with party B, they obtain the following information from the server: The public identity key pub(IK_B), the signed prekey pub(SPK_B) along with its signature, and a one-time prekey pub(OPK_B) (if available). Now, A produces an ephemeral key EK_A and verifies the signature of pub(SPK_B). Afterwards, three Diffie-Hellman key agreements are performed: Between sec(IK_A) and pub(SPK_B), between sec(EK_A) and pub(IK_B), and between sec(EK_A) and pub(SPK_B). A symmetric key sk is then derived from these agreements. If a one-time prekey pub(OPK_B) was available, a fourth key agreement between sec(EK_A) and pub(OPK_B) is performed and also taken into account in the computation of sk. Finally, A sends an initial message to B that contains pub(IK_A), pub(EK_A), the index of pub(OPK_B) (if available), and an initial ciphertext encrypted with key sk. After B got this initial message, B performs the symmetric computations of A to obtain the symmetric key sk and then decrypts the initial ciphertext to verify sk.

Double Ratchet [45] The Double Ratchet protocol (Figure 3 in Appendix A) tracks the cryptographic state of communication between two parties A and B. It is designed to provide forward and backward secrecy even when several of the involved keys are leaked.

The protocol state consists of four *chains*, which are stored by each party: The asymmetric *Diffie-Hellman ratchet*, and the symmetric *root*, *sending* and *receiving* chains.

Diffie-Hellman Ratchet: The Diffie-Hellman ratchet is a sequence of Diffie-Hellman key exchanges on ephemeral keys. Let $EK_P^{(i)}$ denote the ephemeral key of party $P \in \{A, B\}$ in round i. Each round is partitioned into two phases. At the start of the first phase of round i, party A knows $\text{pub}(EK_B^{(i)})$ and generates an ephemeral key $EK_A^{(i)}$. First, A performs a Diffie-Hellman key-exchange between $\text{sec}(EK_A^{(i)})$ and $\text{pub}(EK_B^{(i)})$ to derive a shared secret $ssv_1^{(i)}$. Now, A sends $\text{pub}(EK_A^{(i)})$ to B. All further messages send from A to B are encrypted via a key derived from $ssv_1^{(i)}$ (see below for details) until B sends a response. A response of B starts the second phase, in which B generates a new ephemeral key $EK_B^{(i+1)}$. Then, B performs a Diffie-Hellman key-exchange between $\text{pub}(EK_A^{(i)})$ and $\text{sec}(EK_B^{(i+1)})$ to derive a shared secret $ssv_2^{(i)}$. Now, B sends $\text{pub}(EK_B^{(i+1)})$ to A. All further messages send from B to A are encrypted via a key derived from $ssv_2^{(i)}$, until A sends a response, which ends round i and starts round $i+1$.

Root Chain: The root chain is a sequence of symmetric-key derivations. Given the shared secret $ssv_j^{(i)}$ from the Diffie-Hellman Ratchet and its current *root chain key* $rk_j^{(i)}$, it computes $\left(rk_{j'}^{(i')}, ck_j^{(i,1)}\right) := \mathsf{KDF}\left(rk_j^{(i)}, ssv_j^{(i)}\right)$, where $ck_j^{(i)}$ is the first chain key of a sending/receiving chain and $rk_{j'}^{(i')}$ is the next root chain key, with $(i', j') := (i, 2)$ if $j = 1$, or $(i', j') = (i+1, 1)$ if $j = 2$. The root chain is initialized with $rk_1^{(1)} = rk$, where rk corresponds to the initial ciphertext key generated by X3DH.

Sending Chain / Receiving Chain: Like the root chain, the sending chain and the receiving chain are sequences of symmetric-key key derivations. The receiving chain of A is the sending chain of B and vice versa. The first *chain key* $ck_j^{(i,1)}$ is generated by the root chain. In the first phase of round i, party A sends messages to B. For each message, the chain is advanced by one step, which yields $\left(ck_1^{(i,k+1)}, sk_1^{(i,k)}\right) := \mathsf{KDF}\left(ck_1^{(i,k)}\right)$. The k-th message from A to B in round i (i. e. in the first phase) is encrypted with $sk_1^{(i,k)}$. Similarly, messages from B to A in round i (i e. in the second phase) are encrypted with $sk_2^{(i,k)}$, where $\left(ck_2^{(i,k+1)}, sk_2^{(i,k)}\right) := \mathsf{KDF}\left(ck_2^{(i,k)}\right)$.

Sesame [46] The Sesame protocol [43] enables the usage of multiple devices for users. In general, the protocol describes two scenarios: the *per-user* scenario, where the identity key of the user is used on all devices of that user and the *per-device* scenario, where every device has its own identity key.

Each device has a set of *sessions* on the server, which are initialized via the X3DH protocol and maintained by the double ratchet protocol. Whenever a device of user A

sends a message to user B, it sends this message to every device associated with A or B either via its current active session or by initializing a new session via X3DH. The server then puts the messages in the mailbox of the receiving devices. The receiving device simply obtains the message from the mailbox and decrypts it via the corresponding session key.

The registration of new devices in the system is not explained in the specification and highly depends on whether a per-device or a per-user scenario is used. Later on, we will show that the current implementation in the Signal messenger is vulnerable to impersonation attacks.

3.3.1. Security Analysis w.r.t. Substitution Attacks

In this section, we investigate whether an attacker is able to conduct an algorithm substitution attack in order to circumvent/break end-to-end encryption, allowing them to read messages sent by the victim and its peers. We discuss the requirements of attacks against the end-to-end encryption, and identify possible attack vectors for algorithm substitution attacks in the protocol.

For simplicity, in this analysis, we assume that the protocol messages are sent via an insecure channel, which can be accessed by the attacker. In practice, the Signal protocol is wrapped into a TLS layer, but our discussion above shows how to leak the long-term key of TLS and thus justifies the insecure channel assumption. Without loss of generality, we also assume that we attack A's side of the protocol, as illustrated in Figure 3 in Appendix A.

Prerequisites for decrypting messages In order to decrypt a message, the attacker needs to get access to the respective symmetric key $sk_j^{(i,k)}$. This key directly depends on the chain key $ck_j^{(i,k)}$, which in turn directly depends on the root chain key $rk_j^{(i)}$. The root chain key depends on the previous root chain key, and the shared secret from the DH ratchet.

The attacker can thus choose one of the following approaches:

(1) Leak $sk_j^{(i,k)}$: This allows to decrypt a single message.

(2) Leak $ck_j^{(i,k)}$: This allows to compute an entire send/receive chain, leading to decryption of one or more messages.

(3) Leak $rk_1^{(i)}$ and the private ephemeral key $\mathrm{sec}(EK_A^{(i)})$: This informs the attacker about the current state of the root chain, which they can use to compute the next two states of the root chain and thus learn the next send and receive chains.

(4) Leak one or multiple long-term keys and conduct a Man-in-the-Middle attack against new conversations: If the identity $\mathrm{sec}(IK_A)$ gets leaked, the attacker can register new prekeys SPK_A and $OPK_A^{(i)}$, which allows them to control the next X3DH key exchange. Also, as we will show in Section 4.3, leaking the identity key is sufficient to register new devices via the Sesame protocol and thus completely circumvent Signal's end-to-end encryption.

Leaking via X3DH The X3DH handshake fully relies on the presence of several randomly generated inputs, which are stored on the server: The signed public prekey $\text{pub}(SPK)$, and a list of public one-time prekeys $\text{pub}(OPK^{(1)}), \text{pub}(OPK^{(1)}), \ldots$. Each client device tracks the available prekeys, and generates new ones, if necessary.

While the identity key and the signed prekey are long-lived, the one-time prekeys are replaced on a regular basis, whenever a new encryption session (conversation) is started. The attacker may thus choose to use the universal ASA to embed secret values into these one-time prekeys, and subsequently drain the pool of available prekeys by conducting a lot of X3DH handshakes, so the client is forced to generate new ones. While in theory this straightforward approach is sufficient to implement the proposed attack strategies, it has a few drawbacks in practice: First, the key generation is usually triggered whenever the client restarts or receives a new conversation, which may not be frequent enough to leak a meaningful amount of short-term secrets. Second, if this is compensated by modifying the prekey generation job to generate a large amount of prekeys at once (i.e., a sufficient amount to leak a short-term secret), the high processor usage (and energy consumption, on mobile devices) may be noticed by the user. Last, the server owners (and possible other peers) may detect this type of attack, if the affected device uploads unreasonably large amounts of one-time prekeys, and the extractor consumes these without starting new conversations.

Leaking via Double Ratchet The only source of randomness in the Double Ratchet is provided by the ephemeral keys; all other shared secrets, states and keys are derived deterministically, making the Double Ratchet very resistant against algorithm substitution attacks.

This property restricts the attacker to leaking information via the ephemeral keys $\text{pub}(EK_A^{(i)})$, which are re-generated each time the peers exchange messages. As we show in Section 4.3, embedding secret data into ephemeral keys is computationally cheap and hardly detectable due to the asynchronous nature of the protocol. However, this also thwarts attacks that try to leak an entire conversation: For each ephemeral key $EK_A^{(i)}$, there are two root keys $rk_1^{(i)}$ and $rk_2^{(i)}$, which in turn lead to two sending/receiving chains and multiple message encryption keys $sk_j^{(i,k)}$, where each of them cannot be leaked in a single step. The attacker thus needs to focus on specific parts of the conversation, and leak the involved secrets over multiple rounds. However, this method is sufficient to leak long-term secrets, as we demonstrate by implementing attack approach (4) using ephemeral keys in Section 4.3.

4. Attacks on Implementations

In this section, we show the results of applying ASA on implementations of the analyzed protocols. We describe the changes to the implementations and ASA design decisions. The two most probable ways to detect the presence of ASA is either observing a widely different runtime behavior or by detecting modifications to the correct implementation.

Table 1: Benchmark results for generating 1000 ephemeral keys while embedding λ bits of the secret via the universal ASA.

Protocol	original	$\lambda = 1$		$\lambda = 2$		$\lambda = 4$		$\lambda = 8$	
TLS	0.096ms	0.23ms	(2.37x)	0.24ms	(2.48x)	0.34ms	(3.56x)	2.45ms	(25.54x)
WireGuard	0.68ms	0.83ms	(1.23x)	1.16ms	(1.71x)	2.65ms	(3.92x)	11.28ms	(16.22x)
Signal	0.29ms	1.55ms	(5.35x)	2.46ms	(8.48x)	8.07ms	(27.83x)	94.85ms	(327.08x)

Table 2: Benchmark results for the leakage of λ bits via the IV-replacement ASA.

Protocol	original	abs.	rel.	λ
TLS	0.14ms	0.18ms	1.28x	64
WireGuard	0.014ms	0.016ms	1.16x	128

For all implementations, the number of lines of code that we changed is negligible compared to the rest of the code-base of the implementations. To verify empirically that the change of the runtime of the algorithm is sufficiently small, we did an experimental performance analysis and calculated the corresponding overhead. In all of the attacks, the parameters can be chosen, such that this overhead hardly detectable (see Table 1 and Table 2).

4.1. TLS

To evaluate our theoretical attacks in practice, we modified the widely used OpenSSL[2] library, which offers extensive cryptographic functionality, including an implementation of the TLS protocol. We focussed on leaking the long-term private key $\mathrm{sec}(CERT_S)$.

We captured the generated data using the `tshark` [3] command-line utility, and subsequently invoked a script which did a majority based key material reconstruction once every captured data block was processed.

"Hello" message We applied our IV-replacement attack in the `ssl_fill_hello_random()` function of the `s3lib.c` file, which generates the random string embedded in the "Hello" message of the server. The function `ssl_fill_hello_random()` provides this random string. As pseudorandom function we picked $F_{ak}(x) = \mathsf{AES\text{-}128}(x, ak)$, which was already available in the given code base.

The runtime of this attack was measured in the state machine OpenSSL uses to implement the message flow of the TLS protocol. Therefore, the generation of the complete "Hello" message was measured. This is based on the assumption that a victim, who wants to monitor the runtime of their implementation, does monitor the generation of a whole message, instead of the individual operations used to generate a message. The corresponding times are given in Table 2.

[2] `https://www.openssl.org/`
[3] `https://www.wireshark.org/`

The results show that we are able to leak a significant amount of key material (64 bit) per session with only a very moderate overhead of less than 30% in the running time. This attack works both against TLS 1.2 and against TLS 1.3.

Explicit IV With out loss of generality, we assume that we use the AES-CBC block cipher in this section.

To securely transmit the eIV, it is concatenated with the plaintext and then encrypted using the static IV, so we cannot perform an IV-replacement ASA on eIV: Instead, we need to find an eIV such that $F_{ak}(\text{AES-CBC}(\text{eIV.plain}, \text{sIV})$ encodes the desired secret information, which is achieved on the protocol level by sampling random eIVs.

The data encryption and decryption functionality of TLS is implemented in `tls1_enc()` in `s3record.c`. We adjusted this function to generate a random eIV, encrypt the entire message, and check whether the resulting ciphertext encodes a part of $\text{sec}(CERT_S)$. If it does not, we reset the function's encryption state and try again with a different eIV.

This trial-and-error approach required us to encrypt the entire record several times, increasing the computation time. As an optimization, an attacker may perform this attack at the algorithm level instead, by encrypting a single block (which will contain the eIV) and then checking for embedded secrets. However, this comes with the cost of having to make these adjustments for each available cipher, instead of doing one generic attack on the protocol level.

The resulting measurements for different amounts of leaked bits are shown in Table 1. We conclude that even due to the higher overhead of repeatedly encrypting a record, the required computation time for embedding 8 bits still resides well below a common network latency, making this attack hard to notice in a scenario like serving a low-traffic website. If the victim however transmits huge amounts of data (e.g., by sending a large file), the bandwidth is significantly reduced and the attack may be detected, forcing the attacker to choose a smaller λ.

Note that this attack can only be used against TLS 1.2, as explicit IVs do not exist anymore in TLS 1.3.

4.2. WireGuard

For WireGuard we implemented a proof-of-concept for attacks against both the handshake and the cookie messages, as discussed in Section 3.2.1. We picked the Linux kernel module[4], which can be installed on systems with older Linux kernels, where no built-in WireGuard support is available yet. Both proof-of-concept attacks leak the private component $\text{sec}(SK_R)$ of the static identity key, from which the public component is trivially derived.

4.2.1. Implementation

Handshake The ephemeral key generation for the handshake is used in two places: One is called whenever the peer initiates a new handshake, the other when the peer responds

[4] `https://git.zx2c4.com/WireGuard-linux-compat`

to a handshake message. Both are located in `/src/noise.c`, which we modified to implement our universal ASA: We changed the key generation, such that it samples a new random private key sec $\left(EK^{\text{cand}}\right)$, tests whether the associated public key pub $\left(EK^{\text{cand}}\right)$ contains a part of the secret which is designated to be leaked, and repeats if necessary. Since the code base already offers the `blake2s` hash function, we used it in conjunction with an attacker key ak as the pseudo random function for hiding the leaked secret.

Cookie Message To implement the IV-replacement attack for cookie messages, we modified the generation of the random nonce rn in `/src/cookie.c`, such that it only chooses the first 8 bytes at random. Subsequently, we inserted code that embeds the private key into the remaining 16 bytes of rn by XOR-ing it with a pseudorandom value generated with the `blake2s` hash function. Like described in Section 2.2, the first 8 bytes of randomness as well an an attacker key ak are used for the hashing.

4.2.2. Results

We tested the validity of our modifications on a simple setup consisting of two peers. For both attacks, WireGuard connections could be established correctly with both the original and another modified version, meaning that the attack does not influence the stability of the protocol. Further, we tested that the desired key is actually leaked by the modified implementation, by using `pyshark` [5] to trace the communication between the peers. We were able to reconstruct the leaked bits correctly in all cases.

To evaluate the impact of our attacks on the computation time, we benchmarked our proof-of-concept and compared it to the original implementation of WireGuard on an Intel Core i5-6260U. We measured the time to create a whole message, in order to get an impression of the relative overhead that is introduced by our attacks.

Handshake The results for embedding the private key sec(SK_R) into the handshake response messages can be found in Table 1. We conclude that embedding one or two bits into the ephemeral key does not produce a significant overhead relative to the original code. Increasing the number of embedded bits to four or eight results in a significantly higher relative overhead; however, in absolute terms the delays introduced by our attack may still be difficult to notice, since WireGuard operates over networks, where one can expect latencies in the range of tens to hundreds of milliseconds.

Embedding eight bits into the ephemeral keys results in a feasible attack, since $8 \cdot 32 = 256$ handshake messages have to be recorded by an attacker to reconstruct the key with a 98% chance (see the discussion on the universal ASA in Section 2.2). The default WireGuard configuration swaps the symmetric key every 2 minutes by initiating a new handshake, meaning that at most 9 hours of a running session would have to be recorded in order to leak a victim's key. This means that eavesdropping on a victim for a single work day would be sufficient to obtain their private key and steal their identity.

[5] `https://github.com/KimiNewt/pyshark`

Cookie Message Table 2 shows the results for embedding the private static identity key $\mathrm{sec}(SK_R)$ into cookie messages. As this attack is not probabilistic, one half of the private key can be embedded into the random nonce with nearly no computational overhead, which is almost impossible to detect when communicating via a network connection.

Here it is also noteworthy, that the victim has to be under load to send cookie messages, however, an active attacker can easily cause this load on their victim by sending forged messages or resending recorded handshake initiations. This does not make a detection of the ASA trivial, because such an attack cannot be distinguished from a simple DoS attack. Note, that the load needs to be caused by other handshake messages, whereas the number of parallel requests needed depends on the WireGuard configuration (the default is 4096).

4.3. Signal

Our proof-of-concept attack against the Signal protocol consists of two stages: First, we leak the long-term identity key IK_A of A via an algorithm substitution attack; then, we use the leaked identity key to register a new device, which is controlled by the attacker.

4.3.1. Leaking the identity key

Implementation For our attack, we modified the desktop client implementation of Signal [1]. The desktop client is based on Electron and written in JavaScript and TypeScript. Its core Signal protocol implementation is contained in a single file, which is called `/libtextsecure/libsignal-protocol.js`, and has more than 25.000 lines, where around 20.000 lines are taken up by an `emscripten` runtime and corresponding pre-compiled code. We discuss possible implications of this and other implementation decisions in Section 4.3.3.

To implement the algorithm substitution attack, we modified the mentioned source file and added an alternative key generation function for ephemeral keys in the asymmetric ratchet. The existing key generation function is called in two places: When a new chat conversation is started, and whenever a message is received. We used the universal ASA method: Given an ephemeral key candidate $\mathrm{sec}(EK_A^{(i),\mathrm{cand}})$ and our ASA key ak, we checked whether the value $F_{ak}(\mathrm{pub}(EK_A^{(i),\mathrm{cand}}))$ encoded a part of the identity key IK_A. As pseudorandom function we picked $F_{ak}(x) = \mathsf{HMAC}(x, ak)$, which was already available in the given code base. Finally, we modified the *new conversation* and *message received* event handlers to use our modified key generation.

We tested our implementation by setting up two accounts on Signal's staging (development) servers, exchanging messages between those, and writing the generated keys to the debug log. Afterwards, we used a small script to verify that the keys indeed contained parts of the secret identity key.

Results To benchmark our implementation, we generated 1000 manipulated ephemeral keys and measured the spent computation time on an Intel Core i3-5010U. The results can be found in Table 1. As the measurements show, even encoding 8 secret bits per

ephemeral keys leads to a hardly noticeable overhead of around 95 milliseconds in the average case. Note that, in contrast to TLS and WireGuard, the generation of the new ephemeral keys is done in a non-interactive way, *after* a message is received. Since the Signal protocol is designed to be used in such an asynchronous setting, the perceptibility further diminishes: Peer A cannot be sure whether peer B does read and answer messages immediately, and A also doesn't know the time B needs for typing an answer. We thus conclude that we can efficiently transmit 1 byte of A's secret identity key IK_A per round, without risking detection by the user. If it is known that the users only rarely exchange messages (i. e. the time between two messages is sufficiently long), we can increase this payload even more.

4.3.2. Exploiting the Sesame multi-device feature

After extracting the identity key, we can continue bypassing the Signal messenger's end-to-end encryption: Signal's implementation of the device registration, which is a prerequisite for running the Sesame protocol, solely depends on few long-term secrets (see below for details), which allows an attacker to register arbitrary devices after learning those secrets.

Protocol for adding new devices The protocol for registering a new device, called *provisioning* by the Signal implementation, is illustrated in Figure 4 in Appendix A. We use A to denote the phone (main) instance of A's Signal account, and D to denote the desktop client instance which A tries to register as a new device.

When starting the desktop client, the implementation will open a *provisioning* WebSocket to the Signal servers, which will generate and send a random device UUID $uuid_D$. The desktop client then generates a provisioning key pair PR and encodes $uuid_D$ and $\mathrm{pub}(PR)$ into a QR code, which is presented to the user. Upon scanning the QR code using the Signal app, it will first request a verification code *code* from the server, and then encrypt some of the app's private data with a symmetric key *prs*: $\mathsf{Enc}_{prs}(\{\mathrm{pub}(IK_A), \mathrm{sec}(IK_A), code, pn_A, pk_A\})$. The encrypted data and the encrypted key $\mathsf{Enc}_{PR}(prs)$ is send to the server, which relays it via the provisioning socket to the desktop client. The desktop client uses the private provisioning key $\mathrm{sec}(PR)$ to obtain *prs* and thus decrypt the data packet sent by the app. The desktop client registers with the Signal servers by sending a packet containing the phone number *pn*, the string *code* for verification, a random password pw_D, a random registration ID *regId*, and the device name $name_D$, as chosen by the user and encrypted using the identity key IK_A. Upon receiving the registration packet, the servers return a new device ID $deviceId_D$, completing the protocol. Since the Signal servers require HTTP authentication, the desktop client will include the username $un_D := pn.deviceId$ and the password pw_D in any future communication.

Registering a malicious device Before we can register a malicious device, we evaluate the information which the attacker needs to acquire. The device provisioning protocol only has a single step where private information from the primary instance is required,

namely sending the encrypted identity key to the newly registered device. Thus the attacker is required to obtain the following information in order to conduct the device registration attack:

- the (private) identity key IK_A: As shown in Section 4.3.1, this key can be leaked by an ASA.
- the phone number pn: As it identifies the user, the phone number can be assumed to be known to the attacker. Alternatively, it can also be leaked by the attack with small overhead, as it is typically fairly short.
- the profile key pk_A: The profile key allows accessing certain meta information like the list of contacts; we found that sending the profile key is optional for device registration. If the attacker wants access to that information, they may leak the profile key via an ASA.
- the app's username un_A and password pw_A: These are used in HTTP authentication when communicating with the server. The username directly depends on the phone number and the device ID, and can thus be easily guessed; the password is random and needs to be leaked. Since the authentication data is sent in the clear, but inside the TLS layer, the attacker may also access it either by performing an ASA against TLS, or by gaining (limited) access to the server, which is assumed untrusted by the Signal protocol.

An attacker can then easily build a dummy implementation of the Signal app A, which takes the above information and the content of the displayed QR code, and executes the necessary protocol steps to register a new, attacker-controlled device: The dummy app requests a new verification code from the server, and then forges the private data packet which the server then relays to the desktop app. This entire process does not need any interaction from the account's owner.

Evaluation We implemented the attacker dummy app as described in the last paragraph, and used it to register a new desktop client device, in order to estimate the detectability of our attack. Our first observation is the missing notification of the account's owner, that a new device has been registered: All conversations continue transparently, there is no entry in the chat history that the number of devices has changed. This information is already available at the Sesame protocol layer, so showing it prominently in the chat UI may be a simple countermeasure. As of now, the victim needs to manually check their device list in order to detect the attack. Second, if the attacker has limited control over the Signal servers, they may be able to manipulate the list of devices returned to the victim, thus making their malicious device almost undetectable; the only implication of its presence is the additional session in the Sesame protocol, which is not shown in the UI. Third, the *safety number* feature, which can be used to authenticate the peers via a secure channel, only includes the identity keys of the communication partners; thus, a new device (using the same identity key) is automatically considered trusted. While this improves usability, it poses a security risk.

We thus conclude that the Signal messenger is indeed vulnerable against the registration of a malicious device, where the attacker can read (and write) any messages on the

victim's behalf. The victim has little chance to detect the attack in a timely manner, if they don't check their device list regularly. If the attacker gains control over the Signal servers and suppresses the device entry, the attack is completely undetectable from the Signal app's UI.

4.3.3. On the Signal implementation

For our security analysis, we had to reverse engineer the device provisioning protocol from the desktop client, the Android app, and the server implementation, which turned out to be quite time-consuming. Since the device registration proves to be a crucial part of the messenger's security, the lack of up-to-date documentation is troubling. The same is true for other implementation details, like the server APIs, which, to the best of our knowledge, are not included in the official protocol specification. As of writing this, the latest revision of the API documentation in the wiki is from October 2014 [2].

To implement the ASA in the desktop client, we had to modify its Signal protocol implementation, which is contained in a single source file. This file has more than 25000 lines, where a large majority seems to be a pre-compiled Curve25519 implementation. The sheer amount of lines makes it relatively easy to hide manipulated codes. To detect such manipulation more easily, one could keep auto-generated parts in a separate file and/or generate these on-the-fly during compilation.

4.4. Countermeasures

Clearly, the most severe places to position algorithm substitution attacks, are the ones, where the *IV-replacement ASA* could be used, i.e. in cases where sender-generated randomness is transmitted in the clear. These attacks allowed us to leak sensible information in only a few messages. To prevent this rapid disclosure of information, the following modifications to the affected protocols may be used: The high bandwidth of the attack on TLS comes from the random nonces used in the "Hello" messages. Note that the security analysis of Dowling et al. does not make use of these nonces at all [21]. Hence, a possible mitigation to the described attacks is simply the removal of these nonces, which would however require an update of the standard. For Wireguard, our attack used the random nonce in the cookie messages. To counteract this, the nonce could be derived from values available to peer P', e. g. a hash of the message causing the cookie to be sent (which already contains randomness due to the session ID and the ephemeral key). This way it is possible to remove the nonce from the cookie message, leaving only the universal ASA on the encrypted cookie as a potential target.

Due to their low bandwidth, the attacks based on the *universal ASA* are only suited for long-term keys: the short-term keys are usually refreshed often enough to prevent their leakage via this low-bandwidth channel. Nevertheless, the long-term keys can easily be leaked via this method. As shown above, the forward secrecy of the protocols guarantees that these universal ASAs are always possible. While this is certainly a drawback, forward secrecy also has a very positive aspect. As the universal ASAs can only leak long-term keys, forward secrecy implies that these keys do not allow a purely

passive attacker to read the encrypted messages. If an attacker wants to use these long-term keys, they need to be *active*, e. g. by performing a Man-in-the-Middle attack. It is therefore clearly preferable to prevent IV-replacement attacks, as these attacks can also leak short-term secrets, which can be used by a passive attacker.

An explicit exception for this discussion is Signal, as the leakage of the long-term key allows the registration of new devices, which completely shuts down the protocol's forward secrecy. As described above, the information needed to detect this attack are already present on the side of the client, but inaccessible from the app's UI. To counter this attack, we propose to give a notification that a new device was registered in all chats that the new device participates in, and transition from a user-level authentication to a device-level authentication, such that each pair of devices generates a unique safety number, which both users have to compare whenever a new device is registered. Just recently, an alternative multi-device version of Signal was proposed by Campion et al. [9] that uses per-device IDs and is thus not vulnerable to our attack.

5. Conclusion

In this work, we introduced algorithm substitution attacks against cryptographic protocols. We first showed that such attacks are always possible against any protocol achieving forward secrecy. Afterward, we analyzed the three widely used protocols TLS, WireGuard, and Signal on their vulnerabilities against such attacks. We discovered that TLS and WireGuard are especially vulnerable against these kind of attacks, as the secret long-term key could be leaked using only few messages. While Signal is not as vulnerable, the leak of the secret long-term key led to a catastrophic failure of the protocol's security guarantees due to its multi-device feature. We experimentally verified that all of these attacks are indeed practically relevant and usable. Finally, we suggested countermeasures against the highly efficient IV-replacement ASA.

We believe that many more cryptographic protocols are indeed vulnerable to such attacks. Especially in times where the majority of users download their software from few controlled app stores, it is not unlikely that state level players can apply ASA on select targets with ease. It is therefore important to study how countermeasures developed to prevent ASA against single algorithms can be applied at the protocol-level.

Responsible Disclosure We have informed the Signal organization of our findings. According to their answer, they don't plan to address this issue in the near future.

References

[1] Signal-Desktop (GitHub project). `https://github.com/signalapp/Signal-Desktop`. Accessed 2020-09-28.

[2] Signal Server API Protocol. `https://github.com/signalapp/Signal-Server/wiki/API-Protocol`. Accessed 2020-09-28.

[3] Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. Subversion-resilient signature schemes. In *Proc. CCS*, pages 364–375. ACM, 2015.

[4] Mihir Bellare, Joseph Jaeger, and Daniel Kane. Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In *Proc. CCS*, pages 1431–1440. ACM, 2015.

[5] Mihir Bellare, Joseph Jaeger, and Daniel Kane. Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In *ACM Conference on Computer and Communications Security*, pages 1431–1440. ACM, 2015.

[6] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In *Proc. CRYPTO*, volume 8616 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2014.

[7] Ronen Bergman and Farnaz Fassihi. Iranian hackers found way into encrypted apps, researchers say, 2020. `https://www.nytimes.com/2020/09/18/world/middleeast/iran-hacking-encryption.html`. Accessed 2020-10-13.

[8] Sebastian Berndt and Maciej Liśkiewicz. Algorithm substitution attacks from a steganographic perspective. In *ACM Conference on Computer and Communications Security*, pages 1649–1660. ACM, 2017.

[9] Sébastien Campion, Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. Multi-device for signal. In *ACNS (2)*, volume 12147 of *Lecture Notes in Computer Science*, pages 167–187. Springer, 2020.

[10] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In *Proc. USENIX*, pages 319–335. USENIX Association, 2014.

[11] Rongmao Chen, Xinyi Huang, and Moti Yung. Subvert KEM to break DEM: practical algorithm-substitution attacks on public-key encryption. In *ASIACRYPT (accepted)*, 2020.

[12] Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *EuroS&P*, pages 451–466. IEEE, 2017.

[13] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In *CSF*, pages 164–178. IEEE Computer Society, 2016.

[14] Don Coppersmith. Finding a small root of a univariate modular equation. In *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 155–165. Springer, 1996.

[15] Joseph Cox. How police secretly took over a global phone network for organized crime. Motherboard Tech by VICE, July 2, 2020. `https://www.vice.com/en/article/3aza95/how-police-took-over-encrochat-hacked`. Accessed 2020-10-13.

[16] Jean Paul Degabriele, Pooya Farshim, and Bertram Poettering. A more cautious approach to security against mass surveillance. In *Proc. FSE*, volume 9054 of *Lecture Notes in Computer Science*, pages 579–598. Springer, 2015.

[17] Denis Diemert and Tibor Jager. On the tight security of TLS 1.3: Theoretically-sound cryptographic parameters for real-world deployments. *IACR Cryptol. ePrint Arch.*, 2020:726, 2020.

[18] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. *RFC*, 5246:1–104, 2008.

[19] Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. A formal treatment of backdoored pseudorandom generators. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 101–126. Springer, 2015.

[20] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. `https://www.wireguard.com/papers/wireguard.pdf`, 2020. Accessed 2020-10-08.

[21] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *IACR Cryptol. ePrint Arch.*, 2020:1044, 2020.

[22] Marc Fischlin and Sogol Mazaheri. Self-guarding cryptographic protocols against algorithm substitution attacks. In *CSF*, pages 76–90. IEEE Computer Society, 2018.

[23] Nick Howgrave-Graham. Finding small roots of univariate modular equations revisited. In *IMACC*, volume 1355 of *Lecture Notes in Computer Science*, pages 131–142. Springer, 1997.

[24] Hugo Krawczyk. HMQV: A high-performance secure diffie-hellman protocol. In *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 546–566. Springer, 2005.

[25] Hugo Krawczyk and Pasi Eronen. Hmac-based extract-and-expand key derivation function (HKDF). *RFC*, 5869:1–14, 2010.

[26] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. *RFC*, 7748:1–22, 2016.

[27] David A. McGrew. An interface and algorithms for authenticated encryption. *RFC*, 5116:1–22, 2008.

[28] Mary Meeker. Internet Trends 2019 . https://www.bondcap.com/pdf/Internet_Trends_2019.pdf. Accessed 2020-10-08.

[29] Microsoft. Skype private conversation. https://az705183.vo.msecnd.net/onlinesupportmedia/onlinesupport/media/skype/documents/skype-private-conversation-white-paper.pdf, 2018. Accessed 2020-09-28.

[30] Ilya Mironov and Noah Stephens-Davidowitz. Cryptographic reverse firewalls. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 657–686. Springer, 2015.

[31] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[32] B Moller. Security of cbc ciphersuites in ssl/tls: Problems and countermeasures. *http://www. openssl. org/˜ bodo/tls-cbc. txt*, 2004.

[33] Yoav Nir and Adam Langley. Chacha20 and poly1305 for IETF protocols. *RFC*, 7539:1–45, 2015.

[34] Yoav Nir and Adam Langley. Chacha20 and poly1305 for IETF protocols. *RFC*, 8439:1–46, 2018.

[35] Qualys, Inc. SSL Pulse . https://www.ssllabs.com/ssl-pulse/. Accessed 2020-10-07.

[36] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[37] Phillip Rogaway. Authenticated-encryption with associated-data. In *ACM Conference on Computer and Communications Security*, pages 98–107. ACM, 2002.

[38] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2006.

[39] Alexander Russell, Qiang Tang, Moti Yung, and Hong-Sheng Zhou. Cliptography: Clipping the power of kleptographic attacks. In *ASIACRYPT (2)*, volume 10032 of *Lecture Notes in Computer Science*, pages 34–64, 2016.

[40] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The BLAKE2 cryptographic hash and message authentication code (MAC). *RFC*, 7693:1–30, 2015.

[41] Bruce Schneier. Did nsa put a secret backdoor in new encryption standard?, 2007. https://www.schneier.com/essays/archives/2007/11/did_nsa_put_a_secret.html.

[42] Dan Shumow and Niels Ferguson. On the possibility of a back door in the nist sp800-90 dual ec prng. Presentation at the CRYPTO 2007 Rump Session, 2007.

[43] Open Whisper Systems. Signal protocol specifications. `https://signal.org/docs/`. Accessed 2020-09-28.

[44] Open Whisper Systems. Signal protocol specifications. `https://signal.org/docs/specifications/x3dh/`. Accessed 2020-09-28.

[45] Open Whisper Systems. Signal protocol specifications. `https://signal.org/docs/specifications/doubleratchet/`. Accessed 2020-09-28.

[46] Open Whisper Systems. Signal protocol specifications. `https://signal.org/docs/specifications/sesame/`. Accessed 2020-09-28.

[47] WhatsApp. Whatsapp encryption overview. `https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf`, 2017. Accessed 2020-09-28.

[48] Claud Xiao. Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store. Palo Alto Networks Blog, Sept. 17, 2015. `https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/`. Accessed 2020-10-14.

[49] Adam Young and Moti Yung. The dark side of "black-box" cryptography or: Should we trust capstone? In *Proc. CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 1996.

[50] Adam Young and Moti Yung. Kleptography: Using cryptography against cryptography. In *Proc. EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1997.

A. Appendix: Illustrations for the Signal protocol

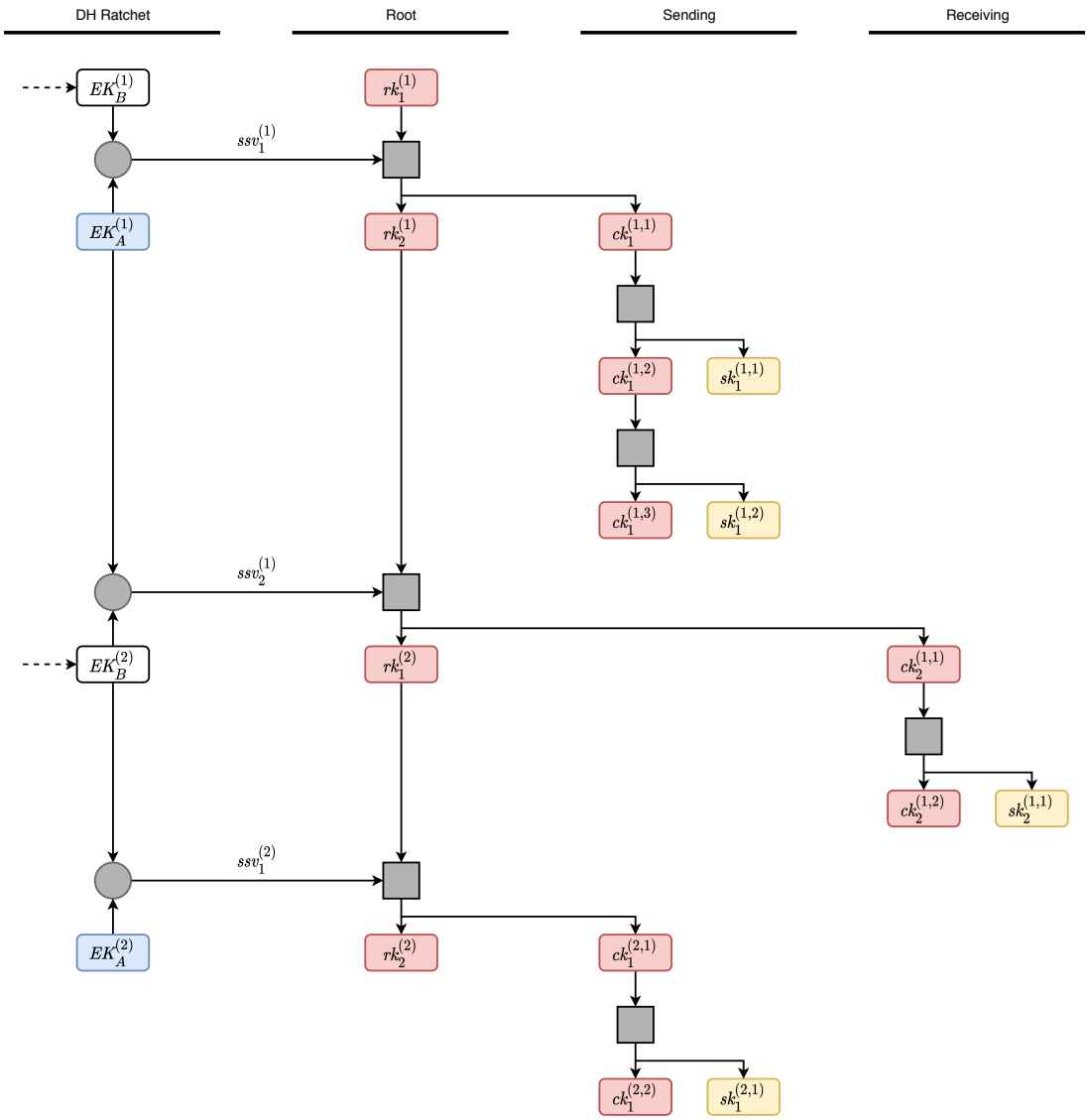

Figure 3: The Signal Double Ratchet protocol, from A's perspective.

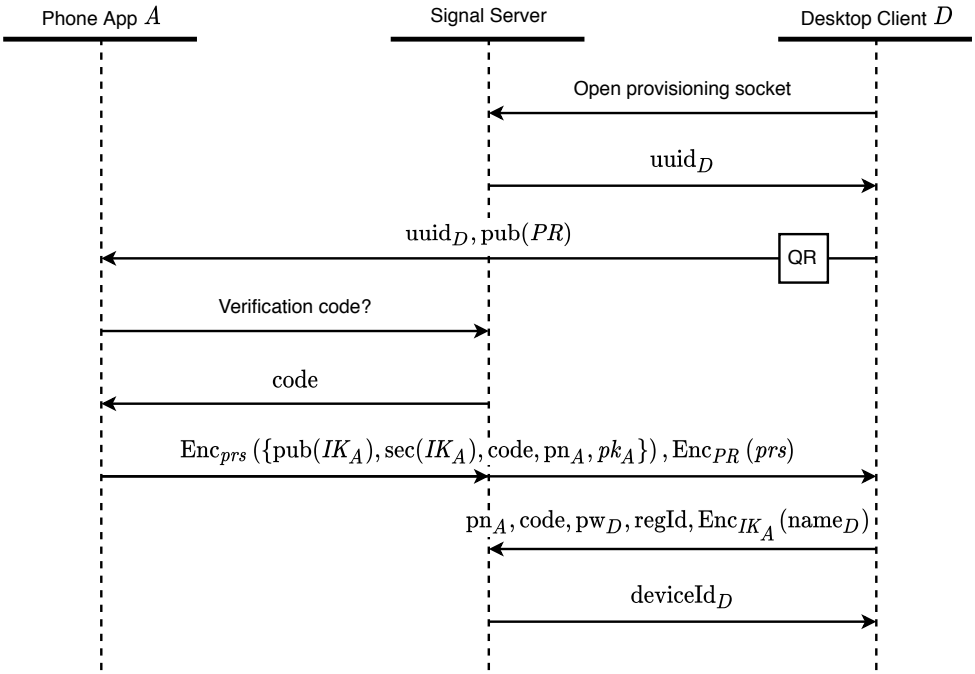

Figure 4: The Signal device provisioning protocol.