Stealth Key Exchange and Confined Access to the Record Protocol Data in TLS 1.3

Marc Fischlin

Cryptoplexity, Technische Universität Darmstadt, Germany
www.cryptoplexity.de
marc.fischlin@tu-darmstadt.de

Abstract.

We show how to embed a covert key exchange sub protocol within a regular TLS 1.3 execution, generating a stealth key in addition to the regular session keys. The idea, which has appeared in the literature before, is to use the exchanged nonces to transport another key value. Our contribution is to give a rigorous model and analysis of the security of such embedded key exchanges, requiring that the stealth key remains secure even if the regular key is under adversarial control. Specifically for our stealth version of the TLS 1.3 protocol we show that this extra key is secure in this setting under the common assumptions about the TLS protocol.

As an application of stealth key exchange we discuss sanitizable channel protocols, where a designated party can partly access and modify payload data in a channel protocol. This may be, for instance, an intrusion detection system monitoring the incoming traffic for malicious content and putting suspicious parts in quarantine. The noteworthy feature, inherited from the stealth key exchange part, is that the sender and receiver can use the extra key to still communicate securely and covertly within the sanitizable channel, e.g., by pre-encrypting confidential parts and making only dedicated parts available to the sanitizer. We discuss how such sanitizable channels can be implemented with authenticated encryption schemes like GCM or ChaChaPoly. In combination with our stealth key exchange protocol, we thus derive a full-fledged sanitizable connection protocol, including key establishment, which perfectly complies with regular TLS 1.3 traffic on the network level. We also assess the potential effectiveness of the approach for the intrusion detection system Snort.

Keywords. Key exchange, secure channel, sanitization, TLS

1 Introduction

Common key exchange protocols allow two parties to agree on a session key. We investigate here the notion of *stealth* key exchange where the parties can create another joint key, called the stealth key, within an execution. The steps to generate this extra key are embedded covertly into the regular execution, such that outsiders remain oblivious if the stealth mode has been used or not. The derived stealth key should be secure even if an adversarial parties gets to know—or can even control— the regularly established session key in such an execution.

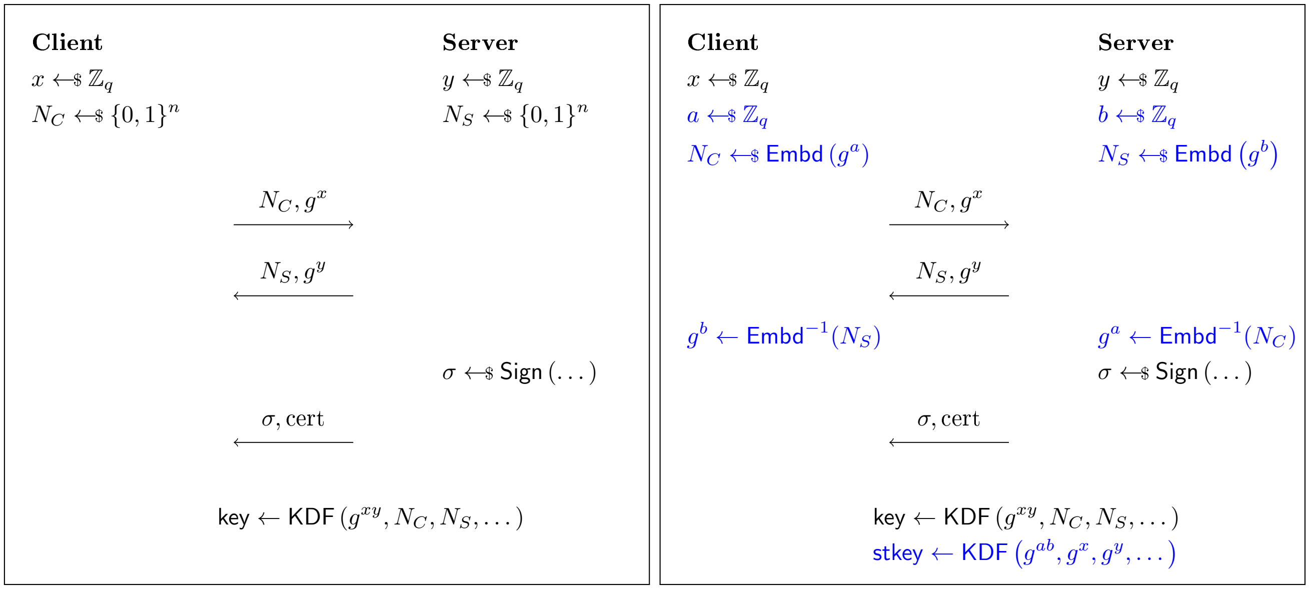
Client
Server
$x \leftarrow\!\!\$\ \mathbb{Z}_q$
$y \leftarrow\!\!\$\ \mathbb{Z}_q$
$N_C \leftarrow\!\!\$\ \{0,1\}^n$
$N_S \leftarrow\!\!\$\ \{0,1\}^n$
N_C, g^x
N_S, g^y
$\sigma \leftarrow\!\!\$\ \mathsf{Sign}(\dots)$
σ, cert
$\mathsf{key} \leftarrow \mathsf{KDF}(g^{xy}, N_C, N_S, \dots)$
Client
Server
$x \leftarrow\!\!\$\ \mathbb{Z}_q$
$y \leftarrow\!\!\$\ \mathbb{Z}_q$
$a \leftarrow\!\!\$\ \mathbb{Z}_q$
$b \leftarrow\!\!\$\ \mathbb{Z}_q$
$N_C \leftarrow\!\!\$\ \mathsf{Embd}(g^a)$
$N_S \leftarrow\!\!\$\ \mathsf{Embd}(g^b)$
N_C, g^x
N_S, g^y
$g^b \leftarrow \mathsf{Embd}^{-1}(N_S)$
$g^a \leftarrow \mathsf{Embd}^{-1}(N_C)$
$\sigma \leftarrow\!\!\$\ \mathsf{Sign}(\dots)$
σ, cert
$\mathsf{key} \leftarrow \mathsf{KDF}(g^{xy}, N_C, N_S, \dots)$
$\mathsf{stkey} \leftarrow \mathsf{KDF}(g^{ab}, g^x, g^y, \dots)$

Figure 1: Simplified version of TLS 1.3 (left) and stealth mode (right).

1.1 The Approach

The idea of building stealth key exchange protocols relies on the widespread deployment of nonces in key exchange protocols, e.g., in TLS 1.3 each party sends a random 256-bit value in the `Hello` messages. In terms of security these nonces should ensure that sessions are unique, but usually one does not need to assume anything beyond this property. One can thus use these nonce values to embed further useful information which can be used to derive the additional stealth key. This idea already appears in several anti-censorship protocols like Telex [WWGH11] and Decoy Routing [KEJ+11].[1]

Let us concretely consider the TLS 1.3 key exchange in the (EC)DHE mode with server authentication. For a simplified version of this protocol see (the left part of) Figure 1. Besides the client and server nonces N_C and N_S, the parties also run a Diffie-Hellman key exchange protocol (over an elliptic curve), deriving a joint secret g^{xy} from the parties' shares g^x and g^y. This secret g^{xy} is then used in the key derivation step, applied to nonces N_C, N_S and additional information.

The idea is now to embed suitable elliptic curve points for yet another Diffie-Hellman key exchange in the TLS 1.3 nonces. Specifically, both parties on top generate another Diffie-Hellman key g^{ab} by embedding their corresponding shares g^a and g^b into the nonces N_C and N_S. The stealth key is then computed as in the original protocol, but by swapping the role of the Diffie-Hellman values and the nonces. That is, the stealth key is now derived via the key derivation function, applied to the secret g^{ab} for "nonces" g^x and g^y (and the other data). See the right part of Figure 1.

The final step is to make sure that the embedding of the extra Diffie-Hellman values remains hidden. Here we use the Elligator proposal of Bernstein et al. [BHKL13] which allows to efficiently create elliptic curve points which are statistically indistinguishable from uniform bit strings. We discuss the exact embedding algorithms within. Once this is accomplished, we have implemented the stealthiness.

1.2 Applications

We briefly discuss here some applications of stealth key exchange. The applications partly also serve as a motivation for our security model in the next subsection. We remark once more that we discuss related

[1] We give a comprehensive comparison to related works in Section 2, after having discussed the main ideas.

concepts in more detail in Section 2.

The first application is a weak form of deniable communication. Since the stealth mode cannot be distinguished from an actual key exchange execution, an outsider cannot know if the parties have agreed on the extra key. If now the parties use this key to encrypt the communication upfront, before pushing it through the channel secured by the actual session key, then they can claim to have sent random data.

Another application is to reduce the trust in Trusted Execution Environments (TEEs). Such environments nowadays usually support, among others, the generation, storage, and computation of Diffie-Hellman keys. Hence, when used within a protocol like TLS, the user may hope to benefit from the additional security guarantees of the TEE (albeit the security of such TEEs may be weaker than expected, e.g., see [CSFP20] for a discussion for TrustZone, for instance). When using the TEE, however, the user remains oblivious about how the Diffie-Hellman keys are generated or stored within the TEE, or even if they are leaked, opening up the possibility of key escrow. With a stealth key exchange the users may generate the additional stealth key and pre-encrypt transmitted data under that key. In this sense the user profits from a "good" TEEs and, at the same time, reduces the risk for a "bad" TEE.[2]

A third example where stealth key exchange may be useful is malware detection for TLS-encrypted communication. So far, one had to share the session key with the middlebox (e.g., an intrusion detection system) in order to allow scanning for potential threats. The sharing of ad-hoc session keys is a challenging problem in itself, but so far practical approaches follow an all-or-nothing principle: either the middlebox has access to the entire communication or no access at all. The stealth key exchange would allow the users to use the extra stealth key to pre-encrypt parts of the communication and only give the middlebox access to other parts. And it remains up to the users to decide which parts remain end-to-end encrypted. We note that the actual implementation of this idea is far from trivial and presented more comprehensively in Section 6, with a closer look at the potential integration into the intrusion detection system Snort in Section 7.

1.3 Security of Stealth Key Exchange

We next discuss what kind of security properties we expect from stealth key exchange protocols. This has previously not been captured rigorously, according to common security models for key exchange. Intuitively, there are two relevant properties. First, we demand that the stealth key is confidential, even if the session key derived in the same execution is disclosed, or, following the TEE application example, even if the adversary gets to influence the session key. Confidentiality of the stealth key here refers to the common notion of indistinguishability from random. We also assume, vice versa, that the session key remains secure if the stealth key is revealed. The second property of a stealth key exchange is that one cannot tell apart executions in which a stealth key is generated from those running merely a regular key exchange execution. This hides the fact whether a stealth key has been agreed upon or not.

We give a security definition in the game-based style of Bellare and Rogaway [BR94], capturing potential correlations between the session key, the stealth key, and the stealthiness of the execution. That is, classical key exchange protocols define confidentiality of the session key via a secret challenge bit b, determining if the adversary gets to learn the session key ($b = 0$) or a random string instead ($b = 1$) in a challenge. The task of the adversary is to predict the bit b. For our security definition we use the same challenge bit b to seize all secrecy properties simultaneously: If the bit b is 0 then we let the parties run in stealth mode, and also hand the adversary the session key resp. the stealth key if requested. If, on the other hand, the bit b is 1, then the parties run in regular mode, and if the adversary asks to be challenged about a key then we return a random (session or stealth) key instead.

[2] Note that our approach uses the random nonces to establish the stealth key. Hence, at least the nonce generation cannot be outsourced to the TEE and its random generator.

We note that the security model with its session-centric view of one party's communication in an execution introduces some interesting effect for the stealthiness. That is, a party may start in stealth mode, with the goal of establishing another key, whereas the intended communication partner does not and instead runs in regular mode. Unless the two parties have already established a shared secret before, they cannot secretly coordinate if they both want to run in stealth mode when the key exchange begins. They can learn this after completion of the key exchange protocol, of course, for example, by trying to use the extra key.

1.4 Stealth TLS 1.3

We next prove that the stealth version of TLS 1.3 satisfies the strong security guarantees of a stealth key exchange protocol, when using an appropriate embedding. For the nonce embeddings we use Elligator 2 for $\mathsf{Curve25519}$ [BHKL13], since $\mathsf{Curve25519}$ is also one of the recommended elliptic curves for TLS 1.3. Hence, our security proof shows that the Elligator embedding allows to derive a stealth key which is as secure as the regular TLS channel key (for $\mathsf{Curve25519}$), and remains secure if the session key is leaked or even determined by the adversary. In other words, deriving another fresh key within a given TLS 1.3 is possible, and the fact that this extra key is derived cannot be spotted from the outside.

One could argue that stealth key exchange does not improve over the trivial solution to run another execution with the partner, say, another TLS 1.3 exchange, to generate yet another key. However, such extra executions may be easy to detect and may be prohibited, e.g., for political reasons. The stealth key exchange mode, on the other hand, goes undetected. Another difference lies in the availability of the secret to authorized parties. If a government enforces key escrow for *any* connection, then simply running two instances would not allow to create a key shared only by the communication partners. We note that our intrusion detection system case displays an example where (partial) access to the data may be desired. Remarkably, the embedding technique can be used to provide such a trade-off. Finally, and this depends on the embedding and the protocol, the stealth mode may be faster than two executions, especially in terms of latency.

1.5 Sanitizable Channel Protocols

As a concrete application of the stealth version of TLS 1.3 we show how to lift the mode to accomplish (controlled) sanitization for a TLS 1.3 channel. Going back to the intrusion detection example, we let the sender and receiver run the stealth key exchange protocol, agreeing on the stealth key for establishing an end-to-end protected connection, and letting the receiver use a static Diffie-Hellman part shared with the detection system. The latter implies that detection system, also called sanitizer, and receiver and sender all share the regular session key of the connection. We note that the static Diffie-Hellman share demolishes forward secrecy of the regular key, but our security proof shows that the stealth key is nonetheless forward secure.

The idea is now to let the sender and receiver use the stealth key to conceal information from the sanitizer, protecting the inner data m_{sec} with the stealth key to derive an inner ciphertext c_{sec}. Then the sender inserts this inner ciphertext c_{sec} together with the accessible part m_{plain} of the message through the regular channel protocol for the session key. This allows the sanitizer to check for the plain part only, hiding the m_{sec}-part from the sanitizer. In fact, we use a more fine-grained distinction into secure, confidential, authenticated, and plain message parts.

We show the above approach is secure if the underlying authenticated encryption scheme is secure, in a suitable model for sanitizable channels. We emphasize that the final ciphertexts are slightly longer than if encrypting m_{sec} and m_{plain} directly, because the inner ciphertext c_{sec} also includes an authentication tag. Nonetheless, when using either of the two suggested authenticated encryption methods in TLS 1.3, GCM or ChaChaPoly, the final ciphertext is a legitimate ciphertext according to TLS 1.3 standards. Thus, when executing the stealth key exchange protocol together with the sanitizable channel, this perfectly complies with the TLS 1.3 standard on the network level.

An interesting feature for the sanitizable channel protocol is that we can preserve the stealthiness from the key exchange to the channel protocol. This means that even the sanitizer cannot know if the sender and receiver actually exchange additional messages in the inner ciphertext. For this we use a common property of the authenticated encryption schemes, namely, that one cannot distinguish actual ciphertexts created with the secret key from random bits. Our security model will capture this stealth property of the sanitizable channel, such that our TLS 1.3 based solution, shown secure in this model, also provides this extra feature.

We finally discuss how our sanitizable channel protocol could be integrated into a network intrusion detection system like the open-source program Snort. Snort comes with a predefined set of rules for checking network traffic, of which roughly half touch upon HTTP traffic. Suppose we grant Snort, as the sanitizer, access to HTTP meta-information like the header data by putting these data in the accessible part m_{plain}, but hide the actual HTTP content from Snort in the inner ciphertext c_{sec}. Then we can still cover a vast majority of all HTTP-related rules in Snort but now work over encrypted communication.

2 Related Work

Our result relies on several ideas and techniques appearing in the literature. We discuss here —and delineate from— the most relevant works.

2.1 Steganography

Stealth key exchange is related to steganographic techniques in cryptography which can be traced back to Simmons' work about the prisoner's problem [Sim83]. The case of public-key steganography has been studied extensively, starting with the initial idea mentioned in [AP98, Cra98] to the first formalization by von Ahn and Hopper [vH04]. Several other works focusing on steganographic techniques for public-key encryption followed, e.g., [BC05, Hop05, LK06, BL18]. We note that only the work by von Ahn and Hopper [vH04] discusses key exchange but merely for passive adversaries; all other works in this realm consider encryption.

The most important difference to our setting here is that steganographic schemes embed a message into a regular communication, whereas stealth key exchange "only" aims to generate an extra key. This may sound like a subtle difference but has crucial consequences for the design. Steganographic schemes often embed bits of the message via rejection sampling [BC05], such that for transmitting each bit covertly many samples and one ciphertext are necessary. In fact, Dedíc et al. [DIRR09] show that an exponential number of samples is required unless one exploits specifics of the communication channel. We can bypass the lower bounds since we are only interested in the partners agreeing on an additional secret key.

2.2 Embeddings

The idea of embedding elliptic curve points as bit strings in an indistinguishable way dates back to Möller [Möl04]. In his solution, he uses the fact that the x-coordinate of the point either denotes a valid curve point or a valid point on the twist. This allows to represent public keys as random strings. Möller's idea has been used in StegoToros [WWY$^+$12] to include stegographic techniques in TOR.

The most widely used embedding today is the Elligator approach of Bernstein et al. [BHKL13]. It comes in two flavors, Elligator 1 and Elligator 2. Elligator 1 is based on an approach by Fouque et al. [FJT13] and works for some elliptic curves. Elligator 2 is more general and in particular works with $\mathsf{Curve25519}$

one of the options in TLS 1.3 for elliptic curves. This is why we use Elligator 2 here. We also remark that Bernstein et al. [BHKL13] discuss issues with the covertness of other elliptic curves available in TLS 1.3, especially with NIST's curve P-256 which may not easily yield almost uniform bit strings. The reason is basically that the order of curve P-256 is not sufficiently close to a power of 2.

Tibouchi [Tib14] presents an improvement for Elligator, denoted as Elligator Squared, which overcomes the issue of repeated sampling to find a suitable curve point and may thus be less vulnerable against time-based side channels. Aranha et al. [AFQ$^+$14] further improve the efficiency for Elligator Squared. Unfortunately, the size of the embedded bit string in Elligator Squared is twice as large as in the Elligator case, such that we could not embed it easily into the 256-bit nonce in TLS 1.3 for the same security level. That is, we had to use a 128-bit curve instead, which provided at most 64 bits of security.

2.3 Analyses of TLS 1.3

TLS 1.3 [Res18] has been developed between 2014 and 2018 by the IETF. The process has been accompanied by a number of scientific analyses during the standardization, both cryptographically [DFGS15, KMO$^+$15, KW16] as well as by formal methods and symbolic analyses [BBD$^+$15, BFK16, DFK$^+$17, CHSv16, CHH$^+$17]. The most relevant analysis for us here is the one in [DFGS15] (see also [DFGS21] for an updated version) as it uses a similar security model (but in the multi-stage setting). Noteworthy, since we give a reduction to the security of the basic TLS 1.3 protocol, the latest results about tight security proofs of TLS 1.3 [DG21, DJ21] immediately transfer to our setting. Note that we do not investigate the pre-shared key mode of TLS 1.3 such that corresponding tightness results as in [DDGJ22] do not apply to our setting here.

For the sanitizable channel protocol we use that GCM is a secure authenticated encryption scheme with associated data (AEAD) when used with a pseudorandom permutation [MV04] like AES in TLS 1.3. The same holds for the composition of ChaCha20 and poly1305 [Pro14], assuming ChaCha20 is pseudorandom and poly1305 is a universal hash function. In our security proof we use additional common properties of such AEAD schemes, namely, that ciphertexts cannot be distinguished from random and that the length of the ciphertext can be deduced from the length of the input message. Both AEAD schemes used in TLS 1.3 satisfy these properties (under the aforementioned assumptions).

2.4 Middleboxes

It is well known that end-to-end encrypted payload and packet inspection by middleboxes are usually irreconcilable. Clearly, the privacy requirements of the users are very important. However, De Carné de Carnavalet and van Oorschot [dCdCvO20] give an overview over cases where accessing secured data may still be desirable. This includes legal reasons like lawful interception or fraud detection, security reasons like malware download protection or intrusion detection, performance reasons like caching and compression, and other reasons like parental control. Note that some cases are even in the interest of the end users.

A simple solution is to make sure that the middlebox has access to the channel key such it can access the payload in clear. In previous TLS versions this could be implemented relatively smoothly by using static keys in the key exchange, for which the middlebox knows the secret keys. But this, of course, sacrifices forward security and was one of the reasons to forgo this option in TLS 1.3. Nonetheless, Green et al. [GDH$^+$17] describe a TLS 1.3 variant with static keys to resurrect accessibility, at the cost of forward secrecy.

Another option is to split the end-to-end connection into two connections, one from each user to the middlebox. However, De Carné de Carnavalet and Mannan [dCdCM16] point out potential vulnerabilities due to sloppy certificate checks of middleboxes. Other potential vulnerabilities are unwanted modifications of the content or defaulting to weak cryptography due to the middleboxes. The middlebox-aware TLS protocol maTLS [LSL$^+$19] attenuates this by introducing auditable middleboxes, yet still breaking end-to-end security.

More sophisticated alternatives for the middlebox problem are the BlindBox protocol [SLPR15] and the recently proposed concept of zero-knowledge middleboxes (ZKMB) [GAZ$^+$21]. In the (most basic version of the) BlindBox protocol the sender sends encrypted tokens in addition to the protected communication, secured under a token key derived also from the channel key. The middlebox holds a (secret) set of detection rules in form of keywords. The client provides the middlebox at the beginning of the connection with the encrypted versions of the keywords such that detection is possible. This is done obliviously, without revealing the token key. The overhead of the cryptographic operations make BlindBox an order of magnitude slower than original connections.

As pointed out by the authors of the ZKMB solution [GAZ$^+$21] the issue with BlindBox is that it modifies the actual connection protocol. Preserving the protocol structure is an important compatibility property. The ZKMB protocol overcomes this limitation for showing policy compliance. The idea is that the client and server establish a regular connection, and the client proves in zero-knowledge to the middlebox that the encrypted payload obeys certain rules. Hence, the client-server connection entirely runs the original connection protocol. Relying on recent progress in efficient zero-knowledge proofs the overhead for long-lived connections is only a few milliseconds. For regular TLS connections the overhead in terms of time and storage for precomputations is still significant, though.

Our stealth TLS 1.3 variant comes close in spirit to the multi-context TLS (mcTLS) solution [NSV$^+$15]. In mcTLS the parties generate an end-to-end TLS connection but, at the same time, each party also establishes a connection with the middlebox. This results in different symmetric keys, one shared between the end points, and one shared between each party and the middle box. The different keys can now be used to protect the payload in such a way that the middlebox is able to access data encrypted with the key shared with the sending party, called context encryption in [NSV$^+$15].

Our solution for middleboxes follows the same idea of using context encryption, but has several advantages. First, our solution does not need to modify the TLS 1.3 protocol on the outer layer; only the pre-encryption the inner data inreases the length of the outer encryption (which remains a valid channel encryption). This is an important compatibility property accomplished with the ZKMB protocol [GAZ$^+$21]. Second, and related to the necessary but not necessarily sufficient compatibility property, we achieve stealthiness (almost) for free. Third, mcTLS puts additional trust in the middlebox in terms of certificate verifications.

Finally, let us remark that the BlindBox solution and especially the ZKMB protocol have an advantage in terms of flexibility and security over our approach. Both protocols support checking of general properties which are hidden from the middlebox. In contrast, our solution only allows for context encryption, dividing the payload coarsely into visible and protected parts. In addition, the deployment of the (semi-)static keys diminishes forward secrecy. In return, our solution blends in easily into the existing protocol and does not require any modifications on the network layer.

2.5 Anti-censorship

Closely related to the issue of middleboxes in secure connections is the question of anti-censorhsip. The idea of using covert data to prevent censorship has been put forward in several works before, and some approaches share some of the techniques used here. Arguably, the most prominent examples in this regard are Telex [WWGH11], Cirripede [HNCB11], and Decoy Routing [KEJ$^+$11]. All three approaches are based on similar principles, but differ in details. The idea is to have a client in a TLS connection covertly trigger a dedicated decoy server on the path to the actual server. This allows to bypass censorship since the decoy server, once alerted, will contact the server on behalf of the client and relay the communication. In order to do so, the client and the decoy server need to be establish a joint secret which the client uses in the connection to the actual server and which is thus known to the decoy server. The approaches differ in the way how the decoy server is triggered and how the joint secret between client and decoy server is computed.

Both Telex and Decoy Routing let the client embed a secret tag into nonce in a TLS connection. Specifically, the client holds a public key g^s of the decoy server and embeds $g^r|H(g^{rs})$ in the nonce for randomness r, hash function H, and a (short) Diffie-Hellman key g^{rs}. The decoy server is able to detect that the second half equals the hash while outsiders should not be able to distinguish the cases. The client is then supposed to use $\mathsf{KDF}(g^{rs})$ as the secret in the key establishment with the server, such that the decoy server also holds the session key. We note that the follow-up design of TapDance [WSH14] explicitly uses Elligator 2 for the embedding.

Decoy Routing [KEJ+11] also uses the nonce in the client hello message to trigger a special event, but relies on a pre-shared secret between client and station to embed the tag via HMAC. It also uses this pre-shared secret to agree on the client's secret in the connection. On the other hand, Cirripede [HNCB11] once more uses the Diffie-Hellman based approach, but uses a pre-shared secret during registration to ensure that client and decoy server know the same connection secret.

The main difference to our work here is that all the aforementioned approaches are mainly interested in the covertness to bypass censorship. In contrast, we are interested in the (combined) stealthiness and key secrecy in an end-to-end connection, albeit our application examples show that third parties can get involved if desired. Another difference is that we provide a rigorous cryptographic analysis of the achieved properties. The final point is that we work with TLS 1.3 whereas the earlier proposals of course considered earlier versions.

2.6 Anamorphic Encryption

Recently, Persiano et al. [PPY22] introduced the notion of anamorphic public-key encryption. The idea is to allow the sender and receiver covertly transmit information, even if an observer gets to determine the message to be sent, and gets access to the secret key of the recipient. Their approach is to have an additional insdistinguishable key generation algorithm which, on top of the public and secret key, outputs another special key, the double key. When sharing this double key with the sender, the two parties can covertly communicate. Persiano et al. [PPY22] give constructions based on rejection sampling and based on the Naor-Yung paradigm. In [KPP+23] the idea was extended to signature schemes.

Anamorphic encryption, like the approach of embedding information into nonces, displays similar ideas to covertly communicate in the presence of observers. There are, nonetheless, major differences between our work and anamorphic encryption. At foremost, we work in the domain of key exchange, implicitly solving the question on how the stealth (or, double) key is securely shared between sender and receiver. Then, our solution even works in the setting where the observer chooses the ephemeral secrets on the receiver's side (cf. the TEE example), whereas in anamorphic public-key encryption the receiver presents a suitable secret key to the observer. A disadvantage of our solution is that, when referring to communication of data, our embedding of the covertly sent messages in the channel protocol increases the length of the ciphertext, such that we can hardly hide the fact that we are using a scheme with allows for covert communication. In contrast, in anamorphic encryption the "anomorphic" ciphertexts are indistinguishable from the ones of a given innocuous system.

2.7 Sanitizable Cryptography

The notion of sanitizable signature schemes has been introduced by Ateniese et al. [ACdMT05]. Such schemes allow a designated party, called the sanitizer, to modify a signed message according to some predefined rule, such that authenticity of the derived message is still verifiable. We lift here this idea to channel protocols. As the intrusion detection system in our setting plays the role of the designated party being able to make admissible changes to the payload, we use the term sanitizable channel here.

Many works in the area of sanitizable cryptography nowadays focus on signature schemes, with only a few exceptions. One is the work by Fehr and Fischlin [FF15] which covers sanitizable signcryption schemes. Such schemes combine (public-key) encryption with signatures, making sure that the sanitizer does not learn the original message when sanitizing the signature, nor possibly even the resulting sanitized message. The work does not investigate symmetric-key channel protocols.

Access control encryption, introduced by Damgård et al. [DHO16] and subsequently extended by [KW17, FGKO17, WC21], also involves a sanitizer which ensures that only admissible information can be passed from senders to receivers. Access control encryption rather implements the classical access control requirements (like the no-read rule and the no-write rule) and moreover aims to provide anonymity. All of the aforementioned solutions are geared towards public-key cryptography and indeed use public-key operations to achieve the security properties. Neither of the works looks into real-world channel protocols with a single sender and receiver sharing a symmetric key.

2.8 Other Notions of Stealth Key Exchange

The term “stealth” has been used in connection with key exchange before, yet with different meanings. Rafat [Raf19] explicitly used the term stealth key exchange in order to describe a (plain) Diffie-Hellman key exchange executed over a seemingly covert channel, such as a frequently changing web site. In a sense, this means to execute a key exchange protocol over a steganographic communication channel. Patgiri and Muppalaneni [PM22] propose an (unauthenticated) key exchange protocol, called Stealth, which runs four Diffie-Hellman key exchanges and uses these keys to encrypt messages in a nested but unauthenticated form. Neither of the proposals aims at embedding another key in an existing key exchange protocol execution, nor provides a formal security analysis.

3 Security Model for Stealth Key Exchange

We start by presenting the security model for stealth key exchange. We follow the classical game-based model of Bellare and Rogaway [BR94]. We only consider the single-stage setting where the parties agree on a single session key upon termination of the key exchange phase. TLS 1.3, in contrast, is a so-called multi-stage protocol [FG14] in which several keys are derived —and possibly deployed— during the key exchange phase.

We assume that we are given a two-party key exchange protocol Π. The protocol should be correct in the sense that, if two parties faithfully execute the protocol then they derive the same session key. We capture this more liberally by demanding that in such an execution the two parties output the same session identifier sid which identifies connected sessions. The choice of sid is part of the protocol description. We will later stipulate as a security requirement that identical session identifiers sid also imply identical session keys.

3.1 Attack Model

We assume a set of user identities $\mathcal{U}$, each user u being equipped with a key pair $(\mathsf{sk}_u, \mathsf{pk}_u)$ generated at the outset of the attack, together with a valid certificate cert_u containing the public key pk_u. We assume that algorithm KGen is used to create each certified key pair. The certificates and thus also the public keys are known to the adversary. Let $\mathcal{C}$ be an initially empty set of corrupt users. If the adversary later corrupts a user $\mathsf{id} \in \mathcal{U}$ then id is added to $\mathcal{C}$. We note in the initialization of a session we allow a party's identity to be set to $*$, indicating that this party does not authenticate towards the other party. The understanding here is that $*$ matches any entry from $\mathcal{U}$, i.e., $\mathsf{id} = *$ for any $\mathsf{id} \in \mathcal{U}$ and also $* = *$.

There is also a global bit b for defining security, chosen randomly at the outset and hidden from the adversary. This bit determines if the adversary gets to see the actual (session or stealth) key or a random value. Here, we assume that the session key and the stealth key are chosen according to some efficient distributions $\mathcal{D}_{\mathsf{regular}}$ resp. $\mathcal{D}_{\mathsf{stealth}}$. The bit also decides if to run in stealth or regular mode, for sessions where the adversary does not explicitly determine the choice.

Sessions capture the state of a communicating party within the key exchange protocols. They are described by a tuple

$$(\mathsf{label}, \mathsf{owner}, \mathsf{party}, \mathsf{partner}, \mathsf{role}, \mathsf{mode}, \mathsf{state}, \mathsf{sid}, \\ \mathsf{key}, \mathsf{stkey}, \mathsf{isTested}, \mathsf{isRevealed}, \mathsf{isCorrPrtner}),$$

where label is a unique administrative identifier, owner is a user identity, party and $\mathsf{partner}$ are the user identities indicating the intended communication partners (with $\mathsf{party} \in \{\mathsf{owner}, *\}$, where $\mathsf{party} = *$ or $\mathsf{partner} = *$ denotes that the party does not authenticate), $\mathsf{role} \in \{\mathsf{initiator}, \mathsf{responder}\}$ describes the role of the session, $\mathsf{mode} \in \{\mathsf{regular}, \mathsf{stealth}\}$ describes the mode, $\mathsf{state} \in \{\mathsf{accept}, \mathsf{reject}, \mathsf{running}\}$ the status of the execution, sid the session identifier (initialized to $\perp$ and set upon acceptance), key the session key (initialized to $\perp$ and set upon acceptance), stkey the stealth key (initialized to $\perp$ and set upon acceptance in mode $\mathsf{stealth}$), Boolean values $\mathsf{isTested}$ and $\mathsf{isRevealed}$ (with sub types $\mathsf{regular}$ and $\mathsf{stealth}$, all four entries set to $\mathtt{false}$ in the beginning), and Boolean value $\mathsf{isCorrPrtner}$ initialized to $\mathtt{false}$. We sometimes write $\mathsf{label.owner}$, $\mathsf{label.partner}$ etc. for the corresponding entries in the tuple for the unique identifier label.

The adversary can communicate with each session and change its status through oracle queries. We highlight here two important aspects related to the stealthiness. One is that the adversary can, upon initializing a session, determine the mode, i.e., if the session should execute a regular protocol execution or run in stealth mode. But we also allow the adversary to leave this entry unspecified, in which case we assign the mode according to the challenge bit b. We then need to prevent trivial attacks in which the adversary checks (via Test or Reveal queries) if there exists a stealth key or not, thereby learning the secret bit b.

The other important point refers to the independence of the stealth key from the session key. Since we want the stealth key to be confidential even if the adversary has control over the cryptographic secrets for the regular key exchange part (cf. the TEE example), we also admit the adversary to optionally provide the ephemeral and long-term secrets upon session initialization. If the adversary chooses to do so, then the session key is marked as revealed, but the stealth key can still be tested. We can also view this as a possibility to disclose the secrets for deniability reasons, but still be able to use the stealth key securely. Like session identifiers the precise definition of this auxiliary data is part of the protocol description, potentially also causing the protocol to abort immediately if aux is not sound.

$\mathsf{Init}\,(\mathsf{owner}, \mathsf{party}, \mathsf{partner}, \mathsf{role}, [\mathsf{mode}], [\mathsf{aux}])$**:** Initializes a session for user $\mathsf{owner} \in \mathcal{U}$, with $\mathsf{party} \in \{\mathsf{owner}, *\}$, with intended partner $\mathsf{partner} \in \mathcal{U} \cup \{*\}$, role, and if the optional argument mode is presented, in the corresponding mode. If no mode is determined then we use $\mathsf{mode} \leftarrow \mathsf{regular}$ if $b = 0$ and $\mathsf{mode} \leftarrow \mathsf{stealth}$ if $b = 1$. In this case, i.e., if no mode argument is passed on, we also set $\mathsf{isRevealed.stealth} \leftarrow \mathtt{true}$; else we still let $\mathsf{isRevealed.stealth} \leftarrow \mathtt{false}$. This is to prevent trivial attacks on the bit b by testing for the existence of a stealth key if no mode value is given.

Also set $\mathsf{state} \leftarrow \mathsf{running}$, $\mathsf{sid} \leftarrow \mathsf{key} \leftarrow \mathsf{stkey} \leftarrow \perp$ and $\mathsf{isTested.regular} \leftarrow \mathsf{isTested.stealth} \leftarrow \mathsf{isRevealed.regular} \leftarrow \mathtt{false}$. If $\mathsf{partner} \in \mathcal{C}$ is corrupt then mark $\mathsf{isCorrPrtner} \leftarrow \mathtt{true}$, else $\mathsf{isCorrPrtner} \leftarrow \mathtt{false}$. Generate a new identifier label and store the passed values in the corresponding entries of the tuple. If the optional argument aux is present then the party will use this value in the regular session as auxiliary input, but we set $\mathsf{isRevealed.regular} \leftarrow \mathtt{true}$; if no value aux is passed then the party follows the protocol description. Returns label to the adversary.

$\mathsf{Send}\,(\mathsf{label}, m)$**:** Sends protocol message m to the session with label. Here, m may be empty if the session owner is the initiator and should start sending the first message. If the session label accepts when processing the incoming message and changes to state $\mathsf{state} \leftarrow \mathsf{accept}$, then $\mathsf{label.sid}$ must be set according to the protocol description to a value different from $\perp$. In this case, the session must also set a session key $\mathsf{label.key}$ and, if run in stealth mode, $\mathsf{label.mode} = \mathsf{stealth}$, also a stealth key $\mathsf{label.stkey}$.

$\mathsf{Corrupt}\,(\mathsf{id})$**:** Takes as input a user identity id and returns $\mathsf{sk_{id}}$. Sets in all running sessions $\mathsf{label.state} = \mathsf{running}$ with this intended partner $\mathsf{label.partner} = \mathsf{id}$ the corruption entry $\mathsf{label.isCorrPrtner} \leftarrow \mathtt{true}$. Note that completed sessions are not affected, in order to implement forward secrecy.

$\mathsf{Reveal}\,(\mathsf{label}, \mathsf{mode})$**:** Takes as input a session label and a requested mode. If the session has not accepted, $\mathsf{label.state} \neq \mathsf{accept}$, or has been revealed before, $\mathsf{label.isRevealed.mode} = \mathtt{true}$, then immediately return $\perp$. Else, if the adversary wants to learn the session key, $\mathsf{mode} = \mathsf{regular}$, then return key and set $\mathsf{label.isRevealed.regular} \leftarrow \mathtt{true}$. If the adversary requests the stealth key, $\mathsf{mode} = \mathsf{stealth}$, and the session has been run in stealth mode, $\mathsf{label.mode} = \mathsf{stealth}$, then return the stealth key stkey and set $\mathsf{label.isRevealed.stealth} \leftarrow \mathtt{true}$. In any other case return $\perp$.

$\mathsf{Test}\,(\mathsf{label}, \mathsf{mode})$**:** Takes as input a session label and a requested mode. If the key has been tested before, $\mathsf{label.isTested.mode} = \mathtt{true}$, or the session has not accepted, $\mathsf{label.state} \neq \mathsf{accept}$, then immediately return $\perp$. Else, if $b = 1$ then return the session key key (if $\mathsf{mode} = \mathsf{regular}$) resp. the stealth key stkey (if $\mathsf{mode} = \mathsf{stealth}$), where potentially $\mathsf{stkey} = \perp$. If $b = 0$, on the other hand, pick a random key $k \leftarrow\!\!\$\ \mathcal{D}_{\mathsf{mode}}$ and return k. In either case, $b = 0$ or $b = 1$, set $\mathsf{label.isTested.mode} \leftarrow \mathtt{true}$.

We assume that the adversary eventually stops and outputs a guess b^* for b. We denote by $\mathbf{Exp}^{\mathrm{StKE}}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}}$ the above experiment of adversary $\mathcal{A}$ against the key exchange protocol Π, in which one first creates the certified keys for the users in $\mathcal{U}$ via algorithm KGen, and picks a challenge bit $b \leftarrow\!\!\$\ \{0, 1\}$, and then lets the adversary interact with the oracles as specified above.

3.2 Security Requirements

We follow the common security notions for session matching and key secrecy. The matching property says that identical session identifiers imply identical keys. Note that for stealth keys this can only hold if both parties were running in stealth mode. Uniqueness refers to the fact that at most two sessions should be partnered. The opposite role property states that in two partnered sessions one party takes the role of the initiator and the other party the role of the responder. Authentication says that partnered sessions point to the same intended partner. Note that here we use that $*$ matches any identity from $\mathcal{U}$ (and $*$ itself) by definition, such that unauthenticated parties always obey this property. We remark that our session matching coincides with the notion in [DFGS21] when considering only single-stage security for the final keys.

Definition 3.1 (Session Matching) *Let Π be a stealth key exchange protocol for users $\mathcal{U}$ and key generation algorithm* KGen*, and $\mathcal{A}$ be an adversary. Consider experiment $\boldsymbol{Exp}^{StKey}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}}$ as above. Let $\boldsymbol{Exp}^{Match}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}}$ denote the event that any of the four following properties is violated during the execution of the experiment:*

Matching Keys: *For any acceptingsessions* $\mathsf{label}, \mathsf{label}'$ *with* $\mathsf{label.sid} = \mathsf{label}'.\mathsf{sid} \neq \perp$ *we have* $\mathsf{label.key} = \mathsf{label.key}' \neq \perp$ *and, furthermore, if* $\mathsf{label.mode} = \mathsf{label}'.\mathsf{mode} = \mathit{stealth}$*, then also* $\mathsf{label.stkey} = \mathsf{label}'.\mathsf{stkey} \neq \perp$*.*

Uniqueness: *There do not exist three distinct acceptingsessions* $\mathsf{label}, \mathsf{label}', \mathsf{label}''$ *such that* $\mathsf{label.sid} = \mathsf{label}'.\mathsf{sid} = \mathsf{label}''.\mathsf{sid} \neq \bot$.

Opposite Roles: *There do not exist distinct accepting sessions* $\mathsf{label}, \mathsf{label}'$ *such that* $\mathsf{label.sid} = \mathsf{label}'.\mathsf{sid} \neq \bot$ *but* $\mathsf{label.role} = \mathsf{label}'.\mathsf{role}$.

Authentication: *For any distinct accepting sessions* label, label' *with* $\mathsf{label.sid} = \mathsf{label}'.\mathsf{sid} \neq \bot$ *we have* $\mathsf{label.party} = \mathsf{label}'.\mathsf{partner}$ *as well as* $\mathsf{label.partner} = \mathsf{label}'.\mathsf{party}$.

For the common asymptotic security notions we demand that for any efficient adversary $\mathcal{A}$ the probability of $\mathbf{Exp}^{\mathrm{Match}}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}}$ is negligible.

Since we subsume both key secrecy and the indistinguishability of regular and stealth executions under one notion, we rather call the combined property indistinguishability. This property says that the adversary cannot predict the challenge bit b significantly better than guessing. For this, we need to exclude some trivial attacks, though. The first two properties say that a tested key in a session cannot be revealed, and that the tested key cannot be revealed or tested in a partnered session. Recall that excluding testing on both sides is usually an admissible strategy, since the adversary can already deduce the response for the second test itself, as partnering is usually publicly verifiable.

The third property captures cases where the adversary could already know a tested key trivially. This can either be because the partner is not authenticated ($\mathsf{partner} = *$) or if the partner has been corrupted before the session has been completed ($\mathsf{isCorrPrtner} = \mathtt{true}$). Recall that, if the adversary corrupts the partner of a session after completion, then the $\mathsf{isCorrPrtner}$ predicate is not set. This ensures forward secrecy. To strengthen the notion, we even allow corrupt or unauthenticated partners if the session has been involved in an genuine execution run exclusively by the honest instance of the partner, i.e., if there is another session label' partnered with the tested session.

Definition 3.2 (Indistinguishability) *Let* Π *be a stealth key exchange protocol for users* $\mathcal{U}$ *and key generation algorithm* KGen*, and* $\mathcal{A}$ *be an adversary. Consider experiment* $\boldsymbol{Exp}^{StKey}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}}$ *as above. The adversary* $\mathcal{A}$ *wins the experiment* $\boldsymbol{Exp}^{StKey}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}}$*, denoted as event* $\boldsymbol{Exp}^{Ind}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}}$ *being equal to* 1*, if* $b^* = b$ *and, in addition, all the following points are satisfied:*

No Reveal **nor** Test **for the same key:** *For any accepting session* label *and any* $\mathsf{mode} \in \{\mathit{regular}, \mathit{stealth}\}$, *if* $\mathsf{label.isTested.mode} = \mathtt{true}$ *then we have* $\mathsf{label.isRevealed.mode} = \mathtt{false}$.

No Reveal **nor** Test **on partner for tested key:** *For any accepting session* label *and any* $\mathsf{mode} \in \{\mathit{regular}, \mathit{stealth}\}$ *with* $\mathsf{label.isTested.mode} = \mathtt{true}$ *there does not exist a session* $\mathsf{label}' \neq \mathsf{label}$ *with* $\mathsf{label.sid} = \mathsf{label}'.\mathsf{sid}$ *such that* $\mathsf{label}'.\mathsf{isRevealed.mode} = \mathtt{true}$ *or* $\mathsf{label}'.\mathsf{isTested.mode} = \mathtt{true}$ *(or both).*

No tested key with unauthenticated or already corrupt partner ***(unless there is a matching honest session):*** *For any accepting session* label *and any* $\mathsf{mode} \in \{\mathit{regular}, \mathit{stealth}\}$ *such that* $\mathsf{label.isTested.mode} = \mathtt{true}$*, either* $\mathsf{label.partner} \neq *$ *and* $\mathsf{label.isCorrPrtner} = \mathtt{false}$*, or there exists an accepting session* $\mathsf{label}' \neq \mathsf{label}$ *with* $\mathsf{label.sid} = \mathsf{label}'.\mathsf{sid}$.

In the usual asymptotic notation we would now demand that the protocol Π provides indistinguishability (for $\mathcal{U}$ and KGen) if for any efficient adversary the probability of $\mathbf{Exp}^{\mathrm{Ind}}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}}$ returning 1 is at most negligibly above $\frac{1}{2}$.

4 Stealth TLS Version

We next describe our stealth version of TLS 1.3, called sTeaLS, and prove it to be secure. For this we assume that the client and server use an elliptic curve for the Diffie-Hellman steps which supports efficient embeddings. As a concrete example, the parties may use $\mathsf{Curve25519}$ with Elligator 2 as explained in Section 4.2.

4.1 Protocol Description

We describe here the the case of both parties running either in regular or in stealth mode. A schematic protocol description can be found in Figure 2. If only one party runs in stealth mode it still tries to compute the stealth key as described within, and will succeed with overwhelming probability to compute another key—although the other party does not hold the stealth key.

The protocol follows the idea outlined in the introduction. In regular mode it executes a (EC)DHE-variant of the TLS 1.3 protocol with optional authentication of the parties. The protocol starts with the parties computing the early secrets $(\mathsf{key}_{\mathtt{bind}}, \mathsf{key}_{\mathtt{cets}}, \mathsf{key}_{\mathtt{eems}})$ from the pre-shared key (preset to 0 for the (EC)DHE case). Since we are only interested in the the stealthiness of the final traffic application keys (for client and server), denoted as $\mathsf{key}_{\mathtt{cats}}$ and $\mathsf{key}_{\mathtt{sats}}$ in the protocol, we assume that all intermediate keys are made immediately available to the adversary (which can be formally implemented in multi-stage settings via a Reveal query).

The actual protocol execution start with the client sending a client hello message $\mathtt{CH}$, which includes a 256-bit nonce N_C, and a client key share $\mathtt{CKS}$ carrying a Diffie-Hellman contribution g^x. In the regular mode the client picks the nonce N_C randomly, whereas in stealth mode N_C is the embedding of another Diffie-Hellman share g^a. We note that some mild restrictions on g^a apply, i.e., it must be suitable for the embedding (see Section 4.2). We write $a \leftarrow\$ E_q$ for the sampling according to this restriction. The server answers accordingly with the server hello $\mathtt{SH}$ and (random or embedded) Nonce N_S and server key share $\mathtt{SKS}$ with value g^y. We remark that, formally, the key share messages are part of the hello messages but it is convenient for us to make them explicit. We also require that Diffie-Hellman shares like g^x and g^y can be represented with 256 bits, as is the case for example for $\mathsf{Curve25519}$.

The authentication is done via signatures σ_C on the client side resp. σ_S on the server side for the data exchanged so far. When sending this signature in the client certificate verify message $\mathtt{CCV}$ the client also includes the certificate in the $\mathtt{CCERT}$ message. Analogously for the server (which goes first to save a round trip). We note that we assume that the other party checks the signature and the certificate, and also that the certificate identity matches the pre-specified peer identity. These messages are protected under the handshake traffic secrets of the client $(\mathsf{key}_{\mathtt{chts}})$ and server $(\mathsf{key}_{\mathtt{shts}})$, respectively. Once more we assume that these intermediate keys are handed to the adversary, such that we can in particular decrypt the actually exchanged protocol messages.

The parties also use message authentication keys $\mathsf{key}_{\mathtt{CFIN}}$ and $\mathsf{key}_{\mathtt{SFIN}}$ to compute a MAC over the communication data. Unlike the signature step this part is mandatory. However, remarkably it does not serve a basic security purpose for the security of the keys [DFGS21]. In particular, we again assume that the keys, derived from the handshake secrets are available to the adversary.

The final step is to compute the session key key, given by the client application traffic secret $\mathsf{key}_{\mathtt{cats}}$ and the server application traffic secret $\mathsf{key}_{\mathtt{sats}}$. The additional exporter master secret $\mathsf{key}_{\mathtt{ems}}$ and resumption master secret $\mathsf{key}_{\mathtt{rms}}$ are once more irrelevant for us and can be made available to the adversary. The stealth key is now computed by swapping the nonces and the Diffie-Hellman shares, i.e., using nonces $N_C^* \leftarrow g^x$ and $N_S^* \leftarrow g^y$ and key shares $g^a \leftarrow \mathsf{Embd}_{256}^{-1}(N_C)$ and $g^b \leftarrow \mathsf{Embd}_{256}^{-1}(N_S)$ with Diffie-Hellman key g^{ab}. Run the signature steps and the key derivation steps as in the original protocol for these swapped values.

Since we give a reduction for our stealth version to TLS 1.3 directly, we do not detail the multiple

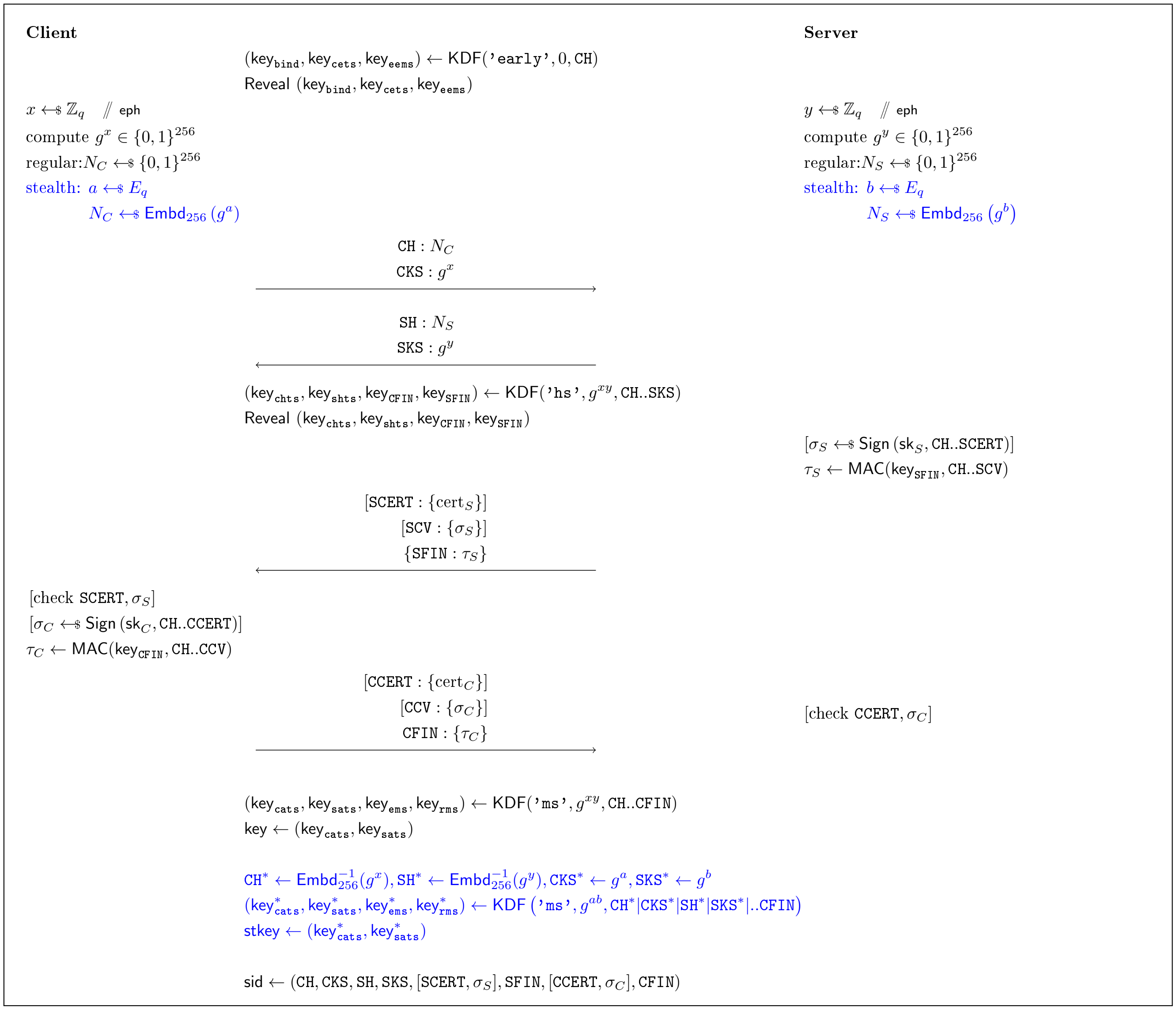
Client
Server
$(\mathsf{key}_{\texttt{bind}}, \mathsf{key}_{\texttt{cets}}, \mathsf{key}_{\texttt{eems}}) \leftarrow \mathsf{KDF}(\texttt{'early'}, 0, \texttt{CH})$
Reveal $(\mathsf{key}_{\texttt{bind}}, \mathsf{key}_{\texttt{cets}}, \mathsf{key}_{\texttt{eems}})$
$x \leftarrow\!\!\$\ \mathbb{Z}_q$ // eph
compute $g^x \in \{0,1\}^{256}$
regular:$N_C \leftarrow\!\!\$\ \{0,1\}^{256}$
stealth: $a \leftarrow\!\!\$\ E_q$
$N_C \leftarrow\!\!\$\ \mathsf{Embd}_{256}\left(g^a\right)$
$y \leftarrow\!\!\$\ \mathbb{Z}_q$ // eph
compute $g^y \in \{0,1\}^{256}$
regular:$N_S \leftarrow\!\!\$\ \{0,1\}^{256}$
stealth: $b \leftarrow\!\!\$\ E_q$
$N_S \leftarrow\!\!\$\ \mathsf{Embd}_{256}\left(g^b\right)$
$\texttt{CH} : N_C$
$\texttt{CKS} : g^x$
$\texttt{SH} : N_S$
$\texttt{SKS} : g^y$
$(\mathsf{key}_{\texttt{chts}}, \mathsf{key}_{\texttt{shts}}, \mathsf{key}_{\texttt{CFIN}}, \mathsf{key}_{\texttt{SFIN}}) \leftarrow \mathsf{KDF}(\texttt{'hs'}, g^{xy}, \texttt{CH}..\texttt{SKS})$
Reveal $(\mathsf{key}_{\texttt{chts}}, \mathsf{key}_{\texttt{shts}}, \mathsf{key}_{\texttt{CFIN}}, \mathsf{key}_{\texttt{SFIN}})$
$[\sigma_S \leftarrow\!\!\$\ \mathsf{Sign}\,(\mathsf{sk}_S, \texttt{CH}..\texttt{SCERT})]$
$\tau_S \leftarrow \mathsf{MAC}(\mathsf{key}_{\texttt{SFIN}}, \texttt{CH}..\texttt{SCV})$
$[\texttt{SCERT} : \{\mathrm{cert}_S\}]$
$[\texttt{SCV} : \{\sigma_S\}]$
$\{\texttt{SFIN} : \tau_S\}$
$[\text{check } \texttt{SCERT}, \sigma_S]$
$[\sigma_C \leftarrow\!\!\$\ \mathsf{Sign}\,(\mathsf{sk}_C, \texttt{CH}..\texttt{CCERT})]$
$\tau_C \leftarrow \mathsf{MAC}(\mathsf{key}_{\texttt{CFIN}}, \texttt{CH}..\texttt{CCV})$
$[\texttt{CCERT} : \{\mathrm{cert}_C\}]$
$[\texttt{CCV} : \{\sigma_C\}]$
$\texttt{CFIN} : \{\tau_C\}$
$[\text{check } \texttt{CCERT}, \sigma_C]$
$(\mathsf{key}_{\texttt{cats}}, \mathsf{key}_{\texttt{sats}}, \mathsf{key}_{\texttt{ems}}, \mathsf{key}_{\texttt{rms}}) \leftarrow \mathsf{KDF}(\texttt{'ms'}, g^{xy}, \texttt{CH}..\texttt{CFIN})$
$\mathsf{key} \leftarrow (\mathsf{key}_{\texttt{cats}}, \mathsf{key}_{\texttt{sats}})$
$\texttt{CH}^* \leftarrow \mathsf{Embd}_{256}^{-1}(g^x), \texttt{SH}^* \leftarrow \mathsf{Embd}_{256}^{-1}(g^y), \texttt{CKS}^* \leftarrow g^a, \texttt{SKS}^* \leftarrow g^b$
$(\mathsf{key}^*_{\texttt{cats}}, \mathsf{key}^*_{\texttt{sats}}, \mathsf{key}^*_{\texttt{ems}}, \mathsf{key}^*_{\texttt{rms}}) \leftarrow \mathsf{KDF}\left(\texttt{'ms'}, g^{ab}, \texttt{CH}^*|\texttt{CKS}^*|\texttt{SH}^*|\texttt{SKS}^*|..\texttt{CFIN}\right)$
$\mathsf{stkey} \leftarrow (\mathsf{key}^*_{\texttt{cats}}, \mathsf{key}^*_{\texttt{sats}})$
$\mathsf{sid} \leftarrow (\texttt{CH}, \texttt{CKS}, \texttt{SH}, \texttt{SKS}, [\texttt{SCERT}, \sigma_S], \texttt{SFIN}, [\texttt{CCERT}, \sigma_C], \texttt{CFIN})$

Figure 2: Stealth version of TLS 1.3. Here [] denote optional authentication steps of the parties, and {} denote protocol messages secure under the handshake traffic secret keys. We note that the exponents a and b are chosen from a suitable subset $E_q \subseteq \mathbb{Z}_q$ which allow for embedding the curve points into strings (see Section 4.2).

key derivation steps in the protocol. Instead, we represent them abstractly as a key derivation function $\mathsf{KDF}(\texttt{'derive'}, \mathrm{IKM}, \mathrm{context})$, applied in a certain derivation context $\texttt{'derive'}$ for intermediate keying material IKM (in our setting, a Diffie-Hellman value) and context information, namely the transcript hash over all previously exchanged communication data. In TLS 1.3 this key derivation is implemented via nested executions of the HKDF key derivation function.

For the desciption of security game it remains to specify the session identifier and the admissible auxiliary input aux. As in [DFGS21] the session identifier is given by the communication transcript,

$$\mathsf{sid} = (\texttt{CH}, \texttt{CKS}, \texttt{SH}, \texttt{SKS}, [\texttt{SCERT}, \sigma_S], \texttt{SFIN}, [\texttt{CCERT}, \sigma_C], \texttt{CFIN}),$$

containing the authentication data if the parties authenticate.

For the auxiliary information we demand that $\mathsf{aux} = (\mathsf{eph}, \mathsf{sk})$ contains the ephemeral Diffie-Hellman secret $x \in \mathbb{Z}_q$ to be used, as well as the long-term signing key sk of the party if authentication is required and Init is called with $\mathsf{party} \neq *$ (else $\mathsf{sk} = \perp$ is admissible). We also require that secret keys are uniquely determined given the public key, and that the correctness of the secret key can be checked efficiently. This holds for example for the ECDSA algorithm or the RSA-PSS algorithm (if the secret key is given in the factorization-based representation), and assuming the collision resistance of the deployed hash function, also for EdDSA. All these algorithms are proposed by TLS 1.3 as admissible signature algorithms [Res18]. We let the protocol immediately abort if the input aux contains improper values in this regard. If the data are sound then the party can execute the protocol entirely with these given cryptographic values.

4.2 Embedding

We briefly discuss one option for the embedding algorithm Embd here. It closely follows the Elligator 2 approach in [BHKL13]. This embedding can be applied for instance to $\mathsf{Curve25519}$ [Ber06] which is one of the elliptic curve options in TLS 1.3 [Res18]. Other options exist, such as Elligator 1. Abstractly we need that the random mapping Embd maps a large portion of the elliptic curve points to a string which is statistically close to a uniform string. We denote by Δ^n_{Embd} the statistical distance to uniformly distributed n-bit strings.

$\mathsf{Curve25519}$ is the elliptic curve $y^2 = x^3 + Ax^2 + Bx \bmod q$ for $A = 486662$, $B = 1$, and prime $q = 2^{255} - 19$. For this curve Bernstein et al. [BHKL13] design an injective mapping $\iota : S \to E(\mathbb{F}_q)$ from a set S of strings to the elliptic curve. Here the set S can be described by a standard encoding σ of bit strings of length $b = \lfloor \log q \rfloor = 254$ into elements from $\mathbb{F}_q$, namely, $\sigma(x_0 \dots x_{b-1}) = \sum x_i 2^i$. We assume that each string is encoded with leading 0's to consist of exactly b bits. Now S is defined as $S = \sigma^{-1}(\{0, 1, \dots, (q-1)/2\})$. Note that by the choice of q these are all bit strings of length 254, except for a negligible subset.

Given S and σ, one can define the embedding $\psi : \mathbb{F}_q \to E(\mathbb{F}_q)$ as follows. Let u be a non-square in $\mathbb{F}_q$ (like $u = 2$ for $\mathsf{Curve25519}$) and $\sqrt{\ }$ be a square-root function over $\mathbb{F}_q$ (e.g., taking the element from 0 to $(q-1)/2$ for the two roots $a, -a$ for some a^2). Let $\chi : \mathbb{F}_q \to \{\pm 1, 0\}$ defined as $\chi(a) = a^{(q-1)/2}$ indicate if a is zero ($\chi(a) = 0$), a non-zero square ($\chi(a) = 1$), or a non-square ($\chi(a) = -1$). For any $r \in \mathbb{F}_q^*$ set

$$v \leftarrow -A/(1 + ur),\ \epsilon \leftarrow \chi(v^3 + Av^2 + Bv),$$
$$x \leftarrow \epsilon v - (1 - \epsilon)A/2, y \leftarrow -\epsilon\sqrt{x^3 + Ax^2 + Bx}.$$

Then $\psi(r) = (x, y)$ describes the curve point for r. One additionally sets $\psi(0) = (0, 0)$ such that ψ is now defined over $\mathbb{F}_q$.

Set $\iota := \psi \circ \sigma$. For the inverse $\psi^{-1} : \psi(\mathbb{F}_q) \to \mathbb{F}_q$ define $\sqrt{\mathbb{F}_q^2}$ as the set of preimages of squares under

$\sqrt{}$, and

$$\psi^{-1}((x,y)) \leftarrow \begin{cases} \sqrt{-x/((x+A)u)} & y \in \sqrt{\mathbb{F}_q^2} \\ \sqrt{-(x+A)/ux)} & y \notin \sqrt{\mathbb{F}_q^2} \end{cases}.$$

This also defines the inverse $\iota^{-1} := \sigma^{-1} \circ \psi^{-1}$. Note that since ι is injective around half of the elliptic curve points have a preimage under ψ. Hence, when picking an elliptic curve point we need on the average two attempts to find a point in the range of ψ.

For our application to stealth TLS we are not entirely done yet. Recall that ψ maps 254-bit strings to elliptic curve points such that, when applying ψ^{-1} to a suitable random curve point P, we get an almost uniform 254-bit string. Our algorithm $\mathsf{Embd}_{256}(P)$ now simply computes $\psi^{-1}(P)$ and appends two random high-order bits. The (deterministic) inverse $\mathsf{Embd}_{256}^{-1}(s)$ drops these two bits and applies ψ to the remaining string.

As pointed out in [BHKL13] the sampling via ψ^{-1} and thus via Embd_{256} is statistically close to uniform. This is due to the fact that the order of the field is $2^{255} - 19$ and thus $(q+1)/2$ very close to 2^{254}. Another point is that the actual Curve25519 works in a prime order subgroup (with cofactor 8), such that extra care must be taken to hide public keys in strings if using the genuine Curve25519 algorithms. One option is then to use a base point generating the full group instead, the other option is to add a low-order point to the Curve25519 point. See Loup Valliant's page elligator.org for more implementation details. Let us point out that the deployment of the embedding may introduce timing-based side channels. Since the embedding is computationally more expensive than simply picking nonces, this may reveal if the party runs in stealth mode via time measurements. We neglect this issue here since previous analyses of TLS 1.3 did not consider such side channels or randomness leakage either.

4.3 Advanced Security Features

As explained, our goal is not to re-prove TLS security, but instead to give a reduction from the indistinguishability our stealth variant to the key secrecy of the regular version of TLS. By construction, the stealth key computation can be thought of as a TLS version in which we swap the nonce and curve point for deriving the key. It is therefore natural to define a swapped version of TLS, denoted swTLS 1.3, which already includes the exchange of the two values for computing the key. Our security proof will then use a reduction to the regular TLS 1.3 protocol for attacking the session key, and to the swapped version swTLS 1.3 for attacking the stealth key. Both protocols are required to provide key secrecy against adversary which can determine nonces, as we discuss first.

Key Indistinguishability against Nonce-Setting Adversaries. The first requirement, for both TLS 1.3 and swTLS 1.3, says that key secrecy still holds if we let the adversary determine the nonce value in executions of the honest parties. This appears to be a reasonable assumption in light of previous results about TLS 1.3. That is, Dowling et al. [DFGS21] do not make any assumptions about the nonces in the key secrecy proof (but only for session matching). Davis and Günther [DG21] only require that the pair of nonce and ephemeral group element is unique in their tight key secrecy proof. If we let the adversary determine the nonce then the minor term for collisions in their security bound decreases from $S^2 \cdot 2^{-256} \cdot \frac{1}{q}$ to $S^2 \cdot \frac{1}{q}$ for the number S of executions. Only the result by Diemert and Jager [DJ21] in their tightness result about key secrecy uses that the nonces are unique.

Formally, we need to specify how an adversary $\mathcal{B}$ can interact with the standard TLS protocol, and here we mean our stealth TLS protocol in mode regular (with the intermediate keys being immediately exposed). Adversary $\mathcal{B}$ is also allowed to choose nonces. The experiment is almost identical to our model for stealth attacks, with two exceptions:

- Init, Reveal, and Test do not take an additional input mode (since TLS 1.3 only runs in regular mode).
- Init does not take the optional aux input. Instead, it takes an optional nonce input which the session owner then uses as a nonce in the protocol execution. The stipulation here is that $\mathcal{B}$ never chooses the same value nonce twice.

We note that formally we can subsume the changes under our model by always requiring $\mathsf{mode} = \mathsf{regular}$ for each oracle call and session, and by interpreting the optional aux as the optional nonce input. The latter is admissible because it depends on the protocol what to do with this input, if present. We accordingly write $\mathbf{Exp}^{\text{Secrecy-NS}}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}}$ (NS for *n*once *s*etting) for the adversary winning this experiment in predicting the challenge bit b and obeying the other restrictions.

The Swapped TLS Protocol. We next discuss the $\mathsf{swTLS\,1.3}$ variant and its security. In this variant we exchange the nonce in the hello messages with the key share value in all subsequent evaluations of the signature algorithm and the key derivation function. In our presentation of the core protocol messages where the hello message only consists of the nonce:

$$\mathtt{CH}|\mathtt{CKS}|\mathtt{SH}|\mathtt{SKS} \mapsto \mathtt{CKS}|\mathtt{CH}|\mathtt{SKS}|\mathtt{SH}$$

in all applications of KDF and of Sign. Again, strictly speaking the key shares are part of the hello messages. According to that terminology we exchange the key share entry with the nonce entry in the hello messages. We leave all other steps unchanged, including also session identifiers.

We note that we do not require TLS 1.3 to be secure in the original and in the swapped order *simultaneously*. Indeed, this infringes with any of the known proofs in [DFGS21, DG21, DJ21] which require the input to the signature to be unique, whereas adaptive swapping could easily violate this. We only require that both $\mathsf{TLS\,1.3}$ and $\mathsf{swTLS\,1.3}$ are individually secure according to the nonce-setting key secrecy experiment above.

Once more, consulting [DFGS21, DG21], the security proofs show key secrecy (in the nonce-setting scenario) for $\mathsf{swTLS\,1.3}$ as well, assuming the hardness of the underlying Diffie-Hellman problem and security of the deployed cryptographic primitives. The reason is that these proofs rely on abstract collision-resistance of the hash function for the transcript hash used in key derivation and signing. Since (bijectively) changing the order of the inputs does not infringe with collision resistance, these results also show security of the swapped version.

Another property of $\mathsf{swTLS\,1.3}$ we require is that we are also able to swap nonce values nonce with elliptic curve points Z in the hello messages. For this we extract the nonce-embedded point $\mathsf{Embd}_n^{-1}(\mathsf{nonce})$ again, and vice versa interpret the point Z as a 256-bit nonce value. The latter is possible by assumption about the deployed group and holds for instance for $\mathsf{Curve25519}$. This swapping has the effect that we now work with a Diffie-Hellman problem over “embeddable” points only. Nevertheless, it is reasonable to assume for $\mathsf{Curve25519}$ and Elligator 2 that the problem is still hard, since half of the points allow for such an embedding.

5 Security Proof of Stealth TLS 1.3

We show security of our stealth protocol. We note that correctness of sTeaLS holds obviously. If two parties faithfully execute the protocol, then they obtain the same session identifier. With the session matching property below it follows that they also have the same session and stealth keys then.

5.1 Session Matching

Proposition 5.1 *Let* sTeaLS *be the stealth TLS 1.3 protocol (for a set of users* $\mathcal{U}$ *and key generation algorithm* KGen*). Then for any adversary* $\mathcal{A}$ *initializing at most* S *sessions we have*

$$\Pr\left[\boldsymbol{Game}^{Match}_{\mathcal{A},\mathsf{sTeaLS},\mathsf{KGen},\mathcal{U}}\right] \leq S^2 \cdot \frac{1}{q} \cdot 2^{-n} + S \cdot \Delta^n_{\mathsf{Embd}},$$

where $n = 256$ *is the nonce length,* q *is the size of the underlying elliptic curve, and* Δ^n_{Embd} *is the statistical distance from uniform for the embedding algorithm in* sTeaLS*.*

Proof. We have to show the four properties, matching keys, uniqueness, opposite roles, and authentication. For matching keys note that identical session identifiers

$$\mathsf{sid} = (\mathtt{CH}, \mathtt{CKS}, \mathtt{SH}, \mathtt{SKS}, [\mathtt{SCERT}, \sigma_S], \mathtt{SFIN}, [\mathtt{CCERT}, \sigma_C], \mathtt{CFIN})$$

imply that the Diffie-Hellman shares are identical, as well as all the other inputs to the key derivation function, such that the parties derive the same keys. Note that this also holds for the stealth key for which we swap the key share and nonce entries. The other property which holds unconditionally is the authentication property: If a party authenticates for entry $\mathsf{id} \neq *$, then it needs to provide a certificate with the correct identity, else the other party aborts. Since the certificate is part of the session identifier sid for authentication, it follows that the identity entries match for identical session identifiers. For unauthenticated parties the entry $*$ matches any other value anyway, such that, overall, the authentication property holds in all cases.

Next we show uniqueness and the opposite-roles property simultaneously. For this we first assume, in a thought experiment, that for sessions in $\mathsf{stealth}$ mode the nonces are not generated by the embedding algorithm but are chosen as random strings. Since we have at most S sessions and the statistical distance of this modification for each session is at most Δ^n_{Embd}, this can increase the adversary's success probability by at most $S \cdot \Delta^n_{\mathsf{Embd}}$. For this modified protocol we can now apply the same line of reasoning as in [DFGS21], saying that the probability of a collision among two client sessions (initiated with $\mathsf{role} = \mathsf{initiator}$) or two server sessions (initiated with $\mathsf{role} = \mathsf{responder}$) on the random nonces (of length $n = 256$) and random group elements (for group size q) is at most $S^2 \cdot \frac{1}{q} \cdot 2^{-n}$. Only if both entries match the session identifiers can be identical. But this means that we cannot have threefold collisions among any kind of sessions resp. colliding sid for identical roles, except with that probability. □

5.2 Indistinguishability

The indistinguishability proof is more elaborate. Recall that we reduce the security of the stealth protocol to the security of $\mathsf{TLS\,1.3}$ resp. $\mathsf{swTLS\,1.3}$ in the nonce-setting scenario. For the theorem's statement it is convenient to denote by $\mathbf{Adv}^{\mathrm{x}}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}} := \Pr\left[\mathbf{Exp}^{\mathrm{x}}_{\mathcal{A},\Pi,\mathsf{KGen},\mathcal{U}} = 1\right] - \frac{1}{2}$ the advantage over the guessing probability for any type of experiment.

Theorem 5.2 *For any key generation* KGen *algorithm and and user set* $\mathcal{U}$*, and any adversary* $\mathcal{A}$ *initializing at most* S *sessions, there exist adversaries* $\mathcal{B}$ *and* $\mathcal{C}$ *(with roughly the same efficiency as* $\mathcal{A}$*) such that*

$$\begin{aligned}&\boldsymbol{Adv}^{Ind}_{\mathcal{A},\mathsf{sTeaLS},\mathsf{KGen},\mathcal{U}}\\ &\quad\leq\; 2S \cdot \left(\boldsymbol{Adv}^{Secrecy\text{-}NS}_{\mathcal{B},\mathsf{TLS\,1.3},\mathsf{KGen},\mathcal{U}} + \boldsymbol{Adv}^{Secrecy\text{-}NS}_{\mathcal{C},\mathsf{swTLS\,1.3},\mathsf{KGen},\mathcal{U}}\right) + S \cdot \Delta^n_{\mathsf{Embd}}\end{aligned}$$

where $n = 256$*,* q *is the order of the group, and* Δ^n_{Embd} *is the statistical distance from uniform for the embedding algorithm in* sTeaLS*.*

Proof. We proceed in a number of game hops. Let $\mathbf{Game}_i$ be the i-th game in the sequence of games, starting with $\mathbf{Game}_0$ being $\mathbf{Exp}^{\mathrm{Ind}}_{\mathcal{A},\mathsf{sTeaLS},\mathsf{KGen},\mathcal{U}}$. We will eventually turn $\mathbf{Game}_0$ into a game $\mathbf{Game}_2$ which is either $\mathbf{Exp}^{\text{Secrecy-NS}}_{\mathcal{B},\mathsf{TLS\,1.3},\mathsf{KGen},\mathcal{U}}$ or $\mathbf{Exp}^{\text{Secrecy-NS}}_{\mathcal{C},\mathsf{swTLS\,1.3},\mathsf{KGen},\mathcal{U}}$, and account for the differences in the games by collecting the probabilities. For this we let $\mathbf{Adv}_i := \Pr[\mathbf{Game}_i] - \frac{1}{2}$ be the corresponding advantages in the game.

$\mathbf{Game}_1$. Our first step is to use the embedding algorithm also in the regular mode. That is, in $\mathbf{Game}_1$ in each session with $\mathsf{mode} = \mathsf{regular}$, instead of picking the nonce $N \leftarrow\!\!\$\ \{0,1\}^n$ randomly, pick some $c \leftarrow\!\!\$\ E_q \subseteq \mathbb{Z}_q$ and compute N as $N \leftarrow\!\!\$\ \mathsf{Embd}_n(g^c)$. The only difference to stealth executions is that we do not use the covert key in the following. The difference to $\mathbf{Game}_0$ is given by the statistical distance between the two sampling procedures, times the maximal number S of sessions:

$$\mathbf{Adv}_0 \leq \mathbf{Adv}_1 + S \cdot \Delta^n_{\mathsf{Embd}}.$$

$\mathbf{Game}_2$. In the next game hop we assume that the adversary only makes a single Test oracle query for a session, and announces at the beginning for which number t of initialized session this will happen and also what type of mode the query will be ($\mathsf{regular}$ or $\mathsf{stealth}$). Denote this type prediction by mode_t. It follows by a hybrid argument (see for example [DFGS21, Appendix A]) that the reduction to a single Test query will increase the advantage by a factor S at most, and predicting the type by guessing it will incur a factor 2. Hence,

$$\mathbf{Adv}_1 \leq 2S \cdot \mathbf{Adv}_2.$$

We next bound the adversary's success probability in the two cases, a Test call for the regular session key and for the stealth key.

Bounding the Case $\mathsf{mode}_t = \mathsf{regular}$. When testing the session key, we turn the adversary $\mathcal{A}$ against sTeaLS into one $\mathcal{B}$ against $\mathsf{TLS\,1.3}$, obeying the necessary restrictions in experiment $\mathbf{Exp}^{\text{Secrecy-NS}}_{\mathcal{B},\mathsf{TLS\,1.3},\mathsf{KGen},\mathcal{U}}$. Note that $\mathcal{B}$ also knows the correct initialization number t, on which the Test call is made, from the beginning on. Algorithm $\mathcal{B}$ is also aware of the fact that the Test query is for the session key.

Adversary $\mathcal{B}$ runs a black-box simulation of $\mathcal{A}$, essentially relaying all communication between $\mathcal{A}$ and the oracles, with the following changes:

- If $\mathcal{A}$ requests to initialize any session, then our adversary $\mathcal{B}$ first checks the validity of the inputs, e.g., that the sk entry in the potential $\mathsf{aux} = (\mathsf{eph}, \mathsf{sk})$ input is only $\perp$ if no authentication occurs, $\mathsf{party} = *$, and otherwise that it constitutes a matching secret key to the public key. For any mismatch $\mathcal{B}$ immediately returns $\perp$, emulating perfectly the protocol description for invalid aux. Else, $\mathcal{B}$ initializes a new session in its experiment, but samples $c \leftarrow\!\!\$\ E_q \subseteq \mathbb{Z}_q$ and passes $\mathsf{nonce} \leftarrow\!\!\$\ \mathsf{Embd}_n(g^c)$ as the optional nonce argument. The knowledge of c allows $\mathcal{B}$ to later compute the stealth key for this session (once the session has accepted) and to reveal it.

  Note that if $\mathcal{A}$ does not provide the optional argument mode upon initialization, with the intention to make it depend on the secret bit b, then $\mathsf{isRevealed.stealth}$ would be set to $\mathtt{true}$ in the attack and lead to an answer $\perp$ in a Reveal query for that session. Hence, $\mathcal{B}$ can ignore this case of an undetermined argument mode, since $\mathcal{B}$ can answer Reveal queries for the stealth key with $\perp$ and since the (only) Test query is for a regular key. If the adversary initializes the t-th session, to be tested later, then we may assume that no optional argument $\mathsf{aux} = (\mathsf{eph}, \mathsf{sk})$ is passed on, else the security experiment would set $\mathsf{isRevealed.regular} \leftarrow \mathtt{true}$ and this session could not be successfully tested on the regular key anymore.

In any case adversary $\mathcal{B}$ stores aux for the session (if provided) and returns the administrative identifier label to $\mathcal{A}$.

- If the adversary $\mathcal{A}$ calls $\mathsf{Send}(\mathsf{label}, m)$ for some session then $\mathcal{B}$ forwards this request to its own Send oracle, with one exception: If upon initialization adversary $\mathcal{A}$ has provided auxiliary information $\mathsf{aux} = (\mathsf{eph}, \mathsf{sk})$ then our algorithm $\mathcal{B}$ does not forward the Send request, but instead computes the answer locally with the help of all the data.
- If adversary $\mathcal{A}$ makes a $\mathsf{Corrupt}(\mathsf{id})$ call the $\mathcal{B}$ forwards this call to its own game, and returns the answer.
- If the adversary $\mathcal{A}$ calls $\mathsf{Reveal}(\mathsf{label}, \mathsf{mode})$ then, for $\mathsf{mode} = \mathsf{regular}$, adversary $\mathcal{B}$ makes a call to $\mathsf{Reveal}(\mathsf{label})$ in its own game and hands back the response. If, on the other hand $\mathsf{mode} = \mathsf{stealth}$, then $\mathcal{B}$ either answers $\perp$ if the session has not accepted or if $\mathsf{label.isRevealed.stealth} = \mathtt{true}$ (e.g., if upon initialization of the session no mode has been specified). Or, $\mathcal{B}$ locally computes the stealth key stkey with the help of the communication data and the exponent c for creating the nonce in the session, and returns the key.
- If $\mathcal{A}$ makes the $\mathsf{Test}(\mathsf{label},\mathsf{mode})$ call then it must be for $\mathsf{mode} = \mathsf{regular}$ and $\mathcal{B}$ can simply forward the request as $\mathsf{Test}(\mathsf{label})$ to its own game, and return the answer.

This describes our adversary $\mathcal{B}$. We first note that $\mathcal{B}$ provides a perfect simulation of $\mathbf{Game}_2$ when interacting with the $\mathsf{TLS}\,1.3$ protocol (in the nonce-setting case). We finally need to check the freshness conditions, and show that if $\mathcal{A}$ in its attack is successful then so is $\mathcal{B}$ in its attack. To see this consider the three cases:

No Test and Reveal for same session: We note that $\mathcal{B}$ would only make a Reveal query to the t-th session if $\mathcal{A}$ would do so in its simulation. But then $\mathcal{A}$ would not be successful either. Note that if $\mathcal{A}$ provided $\mathsf{aux} = (\mathsf{eph}, \mathsf{sk})$ upon initializing the t-th session, and $\mathcal{B}$ would run a local copy instead, then in the game $\mathsf{label.isRevealed.regular}$ would be set to $\mathtt{true}$, such that $\mathcal{A}$ could not win. We conclude that $\mathcal{B}$ only violates this property if $\mathcal{A}$ does.

No Test and Reveal for partner: Assume that there is a partnered session to the t-th session. If $\mathcal{A}$ made a Reveal query to the partner, or provided $\mathsf{aux} = (\mathsf{eph}, \mathsf{sk})$ in the partner session, then it cannot succeed anymore for the t-th session. Since $\mathcal{B}$ would only make a Reveal query to a partner if $\mathcal{A}$ did, and not even initiate a partner session if receiving aux from $\mathcal{A}$, it follows that $\mathcal{B}$ merely violates this property if $\mathcal{A}$ does.

No corrupt partner, and no unauthenticated partner unless there is another matching honest execution: Here we observe that, according to the other two cases, if $\mathcal{B}$ would *not* initiate the matching honest execution, this can only be because it received aux from $\mathcal{A}$ upon initialization and instead run a local copy. But then this would infringe with the second property, because then the other execution in $\mathcal{A}$'s game would set $\mathsf{isRevealed.regular}$ to $\mathtt{true}$ when handing over aux. It follows that $\mathcal{B}$ obeys this property if $\mathcal{A}$ does.

In summary, we have now shown how to turn any successful $\mathcal{A}$ into a successful nonce-setting attacker $\mathcal{B}$ against TLS, such that we can bound the case $\mathsf{mode}_t = \mathsf{regular}$ by

$$\mathbf{Adv}_2 \leq \mathbf{Adv}^{\text{Secrecy-NS}}_{\mathcal{B},\mathsf{TLS}\,1.3,\mathsf{KGen},\mathcal{U}}$$

in case $\mathsf{mode}_t = \mathsf{regular}$.

Bounding the Case $\mathsf{mode}_t = \mathsf{stealth}$. Next assume that $\mathsf{mode}_t = \mathsf{stealth}$. In this case we build an adversary $\mathcal{C}$ attacks the swapped $\mathsf{swTLS\,1.3}$ protocol. The reduction $\mathcal{C}$ is very similar to $\mathcal{B}$ above, but instead swaps the nonces and curve points when relaying communication (such that the internal change of the input order in the transcript hash of $\mathsf{swTLS\,1.3}$ eventually mimics the attack of $\mathcal{A}$ on the stealth TLS version in $\mathbf{Game}_2$):

- Whenever $\mathcal{C}$ receives an Init query of $\mathcal{A}$ with input $\mathsf{aux} = (\mathsf{eph}, \mathsf{sk})$, then $\mathcal{C}$ checks that either $\mathsf{party} = *$ or that sk is the unique secret key to the public key pk of that user. If not, then $\mathcal{C}$ immediately aborts this session and returns $\perp$, as in the protocol description for invalid aux. Else $\mathcal{C}$ asks to initiate a session of $\mathsf{swTLS\,1.3}$ and sets the nonce value in this initialization to be $\mathsf{nonce} \leftarrow g^{\mathsf{eph}}$. Note that, unlike $\mathcal{B}$, our algorithm $\mathcal{C}$ here does not run this session locally, but instead calls the game to execute the session for the given nonce. This is where we need the security against nonce-setting adversaries. The session will thus choose an "embeddable" curve point Z as its share and use the (same) signature key sk to sign, when progressing in the execution. For a given value aux algorithm $\mathcal{C}$ internally notes that $\mathsf{isRevealed.regular} \leftarrow \mathtt{true}$ for this session, according to the attack model. If $\mathcal{A}$ does not hand over aux upon initialization, then $\mathcal{C}$ chooses c and sets the nonce to $\mathsf{nonce} \leftarrow g^c$ when calling Init in its game. In this case $\mathsf{isRevealed.regular} \leftarrow \mathtt{false}$ in the internal simulation of $\mathcal{C}$.

  In either case, $\mathcal{C}$ knows the secret exponent for the nonce value and can thus compute a stealth key if required to do so. We remark that if $\mathcal{A}$ does not provide an input mode for this initialization, then $\mathcal{C}$ sets $\mathsf{isRevealed.stealth} \leftarrow \mathtt{true}$ according to the game anyway, and can later answer Reveal queries for the stealth key easily with $\perp$. The same holds if $\mathsf{mode} = \mathsf{regular}$ is passed on, in which case the session is not supposed to be able to compute a stealth key. Hence, the only case where $\mathcal{C}$ needs to provide the stealth key is when $\mathsf{mode} = \mathsf{stealth}$ is used by $\mathcal{A}$ for initialization.

- Whenever $\mathcal{C}$ receives an incoming protocol message for a party, via a Send query of $\mathcal{A}$, and this message contains a nonce $\mathsf{nonce} \in \{0,1\}^n$ and a curve point Z as hello and key share entries, then $\mathcal{C}$ computes $\mathsf{nonce}' \leftarrow Z$ and $Z' \leftarrow \mathsf{Embd}_n^{-1}(\mathsf{nonce})$, and forwards the message with nonce' and Z' instead of nonce and Z to its Send oracle. If $\mathcal{C}$ receives a message containing a nonce nonce and curve point Z as a response from a Send call, then $\mathcal{C}$ swaps the two values analogously, $\mathsf{nonce}' \leftarrow Z$ and $Z' \leftarrow \mathsf{Embd}_n^{-1}(\mathsf{nonce})$, before handing the answer back to $\mathcal{A}$. Note that we can view a curve point Z as an n-bit string by assumption about the curve, allowing $\mathcal{C}$ to move the curve point to the nonce entry.

- A $\mathsf{Corrupt}(\mathsf{id})$ query of $\mathcal{A}$ in the simulation is immediately relayed in $\mathcal{C}$'s game.

- For a $\mathsf{Reveal}(\mathsf{label}, \mathsf{mode})$ query of $\mathcal{A}$ our algorithm $\mathcal{C}$ can either compute the correct answer for $\mathsf{mode} = \mathsf{regular}$, because $\mathcal{C}$ knows that $\mathsf{isRevealed.regular} = \mathtt{true}$ or, if $\mathsf{isRevealed.regular} = \mathtt{false}$, knows the ephemeral secret. If, on the other hand, $\mathsf{mode} = \mathsf{stealth}$ then $\mathcal{C}$ calls its external $\mathsf{Reveal}(\mathsf{label})$ oracle for $\mathsf{swTLS\,1.3}$ to get the answer. Since $\mathcal{C}$ swaps nonces and curve points on the external interface, and the $\mathsf{swTLS\,1.3}$ protocol swaps the input to the transcript hash, it follows that the external session key corresponds to the internal stealth key in $\mathcal{A}$'s simulation.

- The Test query of $\mathcal{A}$ for the t-th session and $\mathsf{mode} = \mathsf{stealth}$, adversary $\mathcal{C}$ makes the Test query in its game to get the answer.

The simulation is perfect by construction. The swapping of nonces and points on $\mathcal{C}$'s interface between $\mathcal{A}$ and $\mathsf{swTLS\,1.3}$, combined with the input re-ordering for signing in $\mathsf{swTLS\,1.3}$, ensures that the stealth key from $\mathcal{A}$'s point of view correspond exactly to the session keys in $\mathsf{swTLS\,1.3}$. We observe that this uses the fact that the signature key sk is uniquely determined by the public key, such that $\mathcal{A}$'s expectation to use the given (and correct) sk for signing matches the key used in the $\mathsf{swTLS\,1.3}$ protocol. Hence, if $\mathcal{A}$

predicts the challenge bit b in $\mathbf{Game}_2$ for the case $\mathsf{mode}_t = \mathsf{stealth}$, then so does $\mathcal{C}$ against swTLS 1.3. It remains to argue that $\mathcal{C}$, analogously to $\mathcal{B}$, does not violate the freshness conditions:

No Test and Reveal for same session: Algorithm $\mathcal{C}$ only makes a Reveal query to the t-th session if $\mathcal{A}$ does so in the simulation for the stealth key; in any other case $\mathcal{C}$ can answer based on its local data. In case of such a Reveal query of $\mathcal{A}$, however, $\mathcal{A}$ could not win.

No Test and Reveal for partner: Next presume that there is a partnered session to the t-th session. If $\mathcal{A}$ made a Reveal query to the partner session for the stealth key, then it could not win anymore when testing the stealth key in the t-th session. However, in any other case, $\mathcal{C}$ would not make a Reveal query to a partner, because all other Reveal queries are for unpartnered sessions.

No corrupt partner, and no unauthenticated partner unless there is another matching honest execution: Here we use the fact that $\mathcal{C}$ initializes exactly the same sessions as $\mathcal{A}$ does. Hence, if $\mathcal{C}$ violates any of the properties, then so does $\mathcal{A}$. It follows that $\mathcal{C}$ does not infringe with this property unless $\mathcal{A}$ does.

We have thus shown that we can transfer any successful adversary $\mathcal{A}$ into a successful nonce-setting attacker $\mathcal{C}$ against swTLS 1.3, such that we can bound the case $\mathsf{mode}_t = \mathsf{stealth}$ by

$$\mathbf{Adv}_2 \leq \mathbf{Adv}^{\text{Secrecy-NS}}_{\mathcal{C},\mathsf{swTLS\,1.3},\mathsf{KGen},\mathcal{U}}.$$

This concludes the proof. □

On the Auxiliary Input Information. Let us revisit the auxiliary information $\mathsf{aux} = (\mathsf{eph}, \mathsf{sk})$ in our security model, potentially passed on by adversary $\mathcal{A}$ upon initialization. The secret key argument sk may be equal to $\perp$ if the session owner does not authenticate, $\mathsf{party} = *$, in which case only the ephemeral secret eph enters the protocol execution. In our TEE example we assume that such secrets are stored and maintained by a trusted environment and are never handed out; the TEE would perform all operations involving these secrets in its protected space. Indeed, in our reductions the algorithms do not need to know eph explicitly. It would suffice that the adversary, representing the TEE, would give g^{eph} and perform the Diffie-Hellman computations involving eph, on behalf of the reductions, and merely hand back the result. However, this would significantly increase the complexity of the security model since we would then have to determine when to call for the adversary's assistance.

The case of the secret signing key $\mathsf{sk} \neq \perp$ is more delicate. If we would ask the adversary instead to sign the data with the protected key sk if required, then our reduction $\mathcal{B}$ would still succeed, but our reduction $\mathcal{C}$ to the swapped version would not work anymore. The reason is that $\mathcal{C}$ uses the external instance of the swTLS 1.3 protocol to run the simulated instance. By checking that sk is correct and the fact that it is up to $\mathcal{C}$ to compute the signature, the reduction can simply use the externally given signature from the swTLS 1.3 instance. Hence, besides refining the model, one would also need to follow a different proof strategy if one would like to allow for adversarial signatures.

6 Sanitizable Stealth Channels

We next discuss the notion of sanitizable channels. Readers who are merely interested in the idea of how to derive a lightweight and read-only sanitizable channel in TLS 1.3 may skip this section and consult Appendix A instead.

The terminology of sanitizable channels follows the case of signature schemes [ACdMT05] where a designated party can make admissible modifications to a signed message. In sanitizable channels the sender and receiver exclusively share a stealth key stkey, e.g., generated in stealth mode in the key exchange step, as well as a channel key chkey. The channel key is also available to the sanitizing party like an intrusion detection system on the receiver's side. Knowledge of the channel key chkey enables the sanitizer to read or write (parts of the transmitted payload), whereas the stealth key still allows the parties to communicate securely from end to end. In addition, we expect the entire message to be protected from outsiders in the common way.

We first present the general design of such sanitizable channels. In Section 6.5 we discuss the specific case of the TLS 1.3 record protocol and how one can support partly access for the sanitizer. The latter corresponds to the application example for Intrusion Detection Systems presented in Section 7.

6.1 Preliminaries

Messages and Modifications. Any message $m = (m_{\mathsf{sec}}, m_{\mathsf{conf}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$ transmitted over the sanitizable channel may consist of four parts:

- m_{sec} is the part transmitted securely between the end points, confidential, authenticated, and inaccessible to the sanitizer.

- m_{conf} is the part hidden from the sanitizer, but which the sanitizer may modify, e.g., for pruning encrypted data in transit.

- m_{auth} is the part which the sanitizer can read but not modify undetectedly, e.g., to check for viruses in that part.

- m_{plain} is fully available to the sanitizer and can be modified, e.g., to be able to detach viruses if detected.

It is convenient to write $|m|_{\forall} = |m'|_{\forall}$ if the lengths of each components in the two message vectors match, i.e., if $|m_{\mathsf{sec}}| = |m'_{\mathsf{sec}}|$, $|m_{\mathsf{conf}}| = |m'_{\mathsf{conf}}|$, $|m_{\mathsf{auth}}| = |m'_{\mathsf{auth}}|$, and $|m_{\mathsf{plain}}| = |m'_{\mathsf{plain}}|$.

We assume that the admissible sanitization operation are captured via a set $\mathcal{MOD}$ which contains modifications $\textsc{Mod}$ applied to message tuples, $\textsc{Mod}(m)$, but where only the m_{conf}- and m_{plain}-part are actually modified and the m_{sec}- and m_{auth}-part are unchanged. Usually, these modifications only allow simple operations on m_{conf} such a truncation or adding values, but may substitute the entire m_{plain} part. Note that the admissible sanitizer is indeed not supposed to change other message parts, but our attacker may later try to do so, of course. We say that two modifications $\textsc{Mod}$ and $\textsc{Mod}'$ are *length-equivalent* if for any admissible message m we have $|\textsc{Mod}(m)|_{\forall} = |\textsc{Mod}'(m)|_{\forall}$. This means that the two modifications always output message components of the same length for identical input messages.

Since the two parties may not even establish a stealth key stkey during the key exchange step, preventing them from communicating confidentially besides the sanitizer, we also allow the sender to set the parts for m_{sec} and m_{conf} to a value of the form $\diamond^{\ell}$. The intention here being that the parties put a nonsensical placeholder of predetermined length ℓ instead. The length ℓ will allow us to deduce how many random bits we need to put, instead of applying the encryption algorithm. Similarly, since the parties cannot authenticate the message parts against the sanitizer, we assume that m_{auth} is then also of the form $\diamond^{\ell}$.

Key Establishment. We assume that the sender and the receiver have executed the key exchange protocol. The two parties may, or may have not, used the stealth mode to generate a stealth key stkey. For sure, they have generated a session key chkey in such a way that the sanitizer also knows this key chkey (but the sanitizer remains oblivious about the existence of the stealth key). One option is to let the receiver securely pass the session key to the sanitizer upon establishment, albeit this appears to be very inconvenient in the firewall setting. An alternative is to let the sanitizer provide the ephemeral secret of the receiver in the key exchange step, being able to compute chkey from the transcript of communication. This requires the sanitizer to either communicate with the receiver while the key exchange protocol runs, or by sharing a local key with the receiver from which the ephemeral secret is derived. Alternatively, the receiver may re-use a sanitizer-provided ephemeral secret in multiple executions. In fact, this corresponds to the static Diffie-Hellman share solution for TLS 1.3 [GDH+17]. The disadvantage in the latter case is that this solution infringes with forward security (yet, forward security in the stealth part of the connection is still preserved).

Another possibility in the TLS stealth scenario, which hides the usage of a static key towards the sender and outsiders, is to use the static public key g^s of the sanitizer together with the embedded Diffie-Hellman share of the receiver. That is, the receiver embeds g^b into its nonce N_S, independently of the question if it wants to run in stealth mode or not. It now uses the key derivation function on shared keying material g^{bs}, together with the nonce N_C of the client it has received in the first step and its own (embedded) nonce N_S (similar to TLS 1.3 handshake key derivation). The receiver then uses this derived secret as its own Diffie-Hellman secret y when computing g^y as its key share in the connection. We note that the receiver can still compute the stealth key with the help of b with the sender's embedded share g^a, without the sanitizer being able to derive this stealth key.

In the definition of a sanititzable channel protocol below we abstract away all these mechanisms and assume a key generation algorithm ChKGen which returns the keys and the initial states of the parties. In TLS 1.3 the state of the parties for the record layer is simply a counter, incremented each time a ciphertext is processed. The counter value is added to a random offset, called `client_write_iv` resp. `server_write_iv` in TLS. The random offsets are formally part of the secret keys chkey and stkey.

Channel Protocol. A $\mathcal{MOD}$-sanitizable stealth channel protocol consists of efficient probabilistic algorithms $\mathcal{CH} = (\mathsf{ChKGen}, \mathsf{ChSend}, \mathsf{ChRcv}, \mathsf{ChSanit})$, where ChKGen takes a parameter $\mathsf{mode} \in \{\mathsf{regular}, \mathsf{stealth}\}$ and returns a key pair $(\mathsf{chkey}, \mathsf{stkey})$ —where $\mathsf{stkey} = \bot$ for $\mathsf{mode} = \mathsf{regular}$— together with a pair of a sender, receiver, and sanitizer initial state, $(\mathsf{st}_S, \mathsf{st}_R, \mathsf{st}_{\mathrm{San}})$. Algorithm ChSend takes as input the keys chkey, stkey, a parameter $\mathsf{mode} \in \{\mathsf{stealth}, \mathsf{regular}\}$, and the state state, and an admissible message $m = (m_{\mathsf{sec}}, m_{\mathsf{conf}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$, and returns a ciphertext c and the updated state state. For $\mathsf{mode} = \mathsf{regular}$ only messages of the form $m = (\diamond^{\ell_{\mathsf{sec}}}, \diamond^{\ell_{\mathsf{conf}}}, \diamond^{\ell_{\mathsf{auth}}}, m_{\mathsf{plain}})$ are admissible input messages, meaning that the sender only transmits the actual payload but not any stealth information (except for the potential length of the stealth data). Algorithm ChRcv takes as input the keys chkey, stkey (possibly $\mathsf{stkey} = \bot$), the receiver state st_R, and a ciphertext c, and outputs a message $m = (m_{\mathsf{sec}}, m_{\mathsf{conf}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$ as well as the updated state st_R. Note that ChRcv needs to be able to cope with sanitized and potentially unsanitized ciphertexts, without being told explicitly. Similarly, the receiver always tries to recover potential stealth messages, i.e., implicitly uses $\mathsf{mode} = \mathsf{stealth}$. Finally, algorithm $\mathsf{ChSanit}$ receives as input the key chkey, the current state $\mathsf{st}_{\mathrm{San}}$, and the description of a modification $\textsc{Mod} \in \mathcal{MOD}$, and returns a new ciphertext c_{San} and the updated state.

We next tie all algorithms together through the completeness notions, where we assume the common decryptable properties for stealth and non-stealth ciphertext. On top, we stipulate that the sanitizer algorithm always works on either kind of ciphertext. A $\mathcal{MOD}$-sanitizable stealth channel protocol $\mathcal{CH} = (\mathsf{ChKGen}, \mathsf{ChSend}, \mathsf{ChRcv}, \mathsf{ChSanit})$ is *complete* if the following holds:

- For any $(\mathsf{chkey}, \mathsf{stkey}, \mathsf{st}_S^0, \mathsf{st}_R^0, \mathsf{st}_{\mathrm{San}}^0) \leftarrow\$ \mathsf{ChKGen}()$, any admissible messages $m^1, m^2, \ldots, m^j$, any sequence of modes $\mathsf{mode}^1, \ldots, \mathsf{mode}^j$, any ciphertext sequence

$$(\mathsf{st}_S^i, c^i) \leftarrow\$ \mathsf{StSend}(\mathsf{chkey}, \mathsf{stkey}, \mathsf{mode}^i, \mathsf{st}_S^{i-1}, m^i)$$

for $i = 1, 2, \ldots, j$, we always have

$$(\mathsf{st}_R^i, m^i) = \mathsf{StRcv}(\mathsf{chkey}, \mathsf{stkey}, \mathsf{st}_R^{i-1}, c^i)$$

for $i = 1, 2, \ldots, j$.

- For any $(\mathsf{chkey}, \mathsf{stkey}, \mathsf{st}_S^0, \mathsf{st}_R^0, \mathsf{st}_{\mathrm{San}}^0) \leftarrow\!\!\$\ \mathsf{ChKGen}()$, any admissible messages $m^1, m^2, \ldots, m^j$, any sequence of modes $\mathsf{mode}^1, \ldots, \mathsf{mode}^j$, any ciphertext sequence

  $$(\mathsf{st}_S^i, c^i) \leftarrow\!\!\$\ \mathsf{ChSend}(\mathsf{chkey}, \mathsf{stkey}, \mathsf{mode}^i, \mathsf{st}_S^{i-1}, m^i)$$

  for $i = 1, 2, \ldots, j$, any sequence of admissible operations $\textsc{Mod}^i \in \mathcal{MOD}$ for $i = 1, 2, \ldots, j$, any $(c_{\mathrm{San}}^i, \mathsf{st}_{\mathrm{San}}^i) \leftarrow\!\!\$\ \mathsf{ChSanit}(\mathsf{chkey}, c^i, \mathsf{st}_{\mathrm{San}}^{i-1})$ for $i = 1, 2, \ldots, j$, we always have

  $$(\mathsf{st}_R^i, \textsc{Mod}^i(m^i)) = \mathsf{ChRcv}(\mathsf{chkey}, \mathsf{stkey}, \mathsf{st}_R^{i-1}, c_{\mathrm{San}}^i)$$

  for $i = 1, 2, \ldots, j$.

Note that our completeness notion works in the case that either all ciphertexts reach the receiver without modification, or that are ciphertext all sanitized. One could mix these two properties but our solution only achieves this all-or-nothing property.

6.2 Security Model

To define security of our sanitizable channel we follow the security notion of Bellare et al. [BKN04]. This notion allows the adversary to create ciphertexts via a left-or-right sender oracle, the choice of which message to encrypt made according to a secret challenge bit b. The adversary can also decrypt arbitrary ciphertexts via a receiver oracle, where the receiver oracle suppresses the actual message response unless the adversary manages to create a valid out-of-sync ciphertext, i.e., which has not been created at the same point by the sender. In this case the adversary will learn the message but only if $b = 0$. The latter follows the idea of combining indistinguishability and integrity into a single notion, e.g., as done for IND-CCA3 security of authenticated encryption of Shrimpton [Shr04]. That is, if the adversary manages to create a new valid ciphertext and thus breaks integrity, then it will also learn the bit b and can then break indistinguishability.

The formal security experiment for sanitizable channel protocols appears in Figure 3. In our case we simultaneously consider two security modes. One is security against outsiders, i.e., where the adversary is not the sanitizer. In this case, we demand the common channel security of [BKN04] for the overall protocol. This should even hold if we augment the adversary's capabilities by granting access to a sanitization oracle, which the adversary can query about arbitrary ciphertexts. Since the sanitizer may modify parts of the message we extend the left-or-right security of the sending oracle and allow the adversary to pass two possible modifications $\textsc{Mod}^0, \textsc{Mod}^1$ (as long as these modifications show identical output-length behavior for messages).

The second security mode covers insider attacks, i.e., where the adversary is the sanitizer. In this case, however, the adversary is only allowed to query the sending oracle for message pairs with equal m_{auth} and m_{plain} parts, because the sanitizing adversary may access these parts in clear. Another modification to the other case is that now the adversary is supposed to learn the secret bit b if it manages to make the receiver output a different $(m_{\mathsf{sec}}, m_{\mathsf{auth}})$ pair than the intended one and thus break integrity as a sanitizer.[3]

We remark that the stealthiness of our key exchange protocol actually allows us to show a stronger notion for our sanitizable channel. Inheriting this from the key exchange step, knowledge of the channel key does not allow to deduce if a stealth key has been established or not. In this sense, even the sanitizer may not know if the sender has actually sent the confidential part $(m_{\mathsf{sec}}, m_{\mathsf{conf}})$ or merely put random

[3] Noteworthy, this rather resembles message integrity than ciphertext integrity for this inner message part. This is inevitable for a general definition since the outer ciphertext is under control of the sanitizer.

```
$\mathbf{Exp}^{\text{IND-CCA}}_{\mathcal{CH},\mathcal{A},\mathcal{MOD}}$
$b \leftarrow\!\$ \{0,1\},\ \mathtt{ctr}_S, \mathtt{ctr}_R \leftarrow 0,\ \mathcal{C}, \mathcal{M} \leftarrow [\,]$
$(\mathsf{chkey}, \mathsf{stkey}, \mathsf{st}_S, \mathsf{st}_R, \mathsf{st}_{\mathrm{San}}) \leftarrow\!\$ \mathsf{ChKGen}(\mathsf{stealth})$
$\textsc{insider}, \textsc{out-of-sync} \leftarrow \mathtt{false}$
$\mathsf{st}_{\mathcal{A}} \leftarrow\!\$ \mathcal{A}^{\mathsf{OSanKey}}()$
$b^* \leftarrow\!\$ \mathcal{A}^{\mathsf{OSnd},\mathsf{ORcv},\mathsf{OSan}}(\mathsf{st}_{\mathcal{A}})$
return $b = b^*$

$\mathsf{OSanKey}()$
$\textsc{insider} \leftarrow \mathtt{true}$
return $\mathsf{chkey}$

$\mathsf{OSnd}(\mathsf{mode}^0, m^0, \mathsf{mode}^1, m^1)$
if $|m^0|_\forall \neq |m^1|_\forall$ then return $\perp$
if $\textsc{insider}$ and $(m^0_{\mathsf{auth}}, m^0_{\mathsf{plain}}) \neq (m^1_{\mathsf{auth}}, m^1_{\mathsf{plain}})$ then return $\perp$
$(\mathsf{st}_S, c) \leftarrow\!\$ \mathsf{ChSend}(\mathsf{chkey}, \mathsf{stkey}, \mathsf{mode}^b, \mathsf{st}_S, m^b)$
$\mathtt{ctr}_S \leftarrow \mathtt{ctr}_S + 1,\ \mathcal{C}[\mathtt{ctr}_S] \leftarrow \{c\},\ \mathcal{M}[\mathtt{ctr}_S] \leftarrow (m_{\mathsf{sec}}, m_{\mathsf{auth}})$
return $c$

$\mathsf{OSan}(c, \textsc{Mod}^0, \textsc{Mod}^1)$
if $\textsc{insider}$ or $\textsc{Mod}^0, \textsc{Mod}^1 \notin \mathcal{MOD}$ or $\textsc{Mod}^0, \textsc{Mod}^1$ not length-equivalent then return $\perp$
$(\mathsf{st}_{\mathrm{San}}, c_{\mathrm{San}}) \leftarrow \mathsf{ChSanit}(\mathsf{chkey}, \mathsf{st}_{\mathrm{San}}, c, \textsc{Mod}^b)$
for $i = 1$ to $\mathtt{ctr}_S$ do if $c \in \mathcal{C}[i]$ then $\mathcal{C}[i] \leftarrow \mathcal{C}[i] \cup \{c_{\mathrm{San}}\}$
return $c_{\mathrm{San}}$

$\mathsf{ORcv}(c)$
$\mathtt{ctr}_R \leftarrow \mathtt{ctr}_R + 1, (\mathsf{st}_R, m) \leftarrow \mathsf{ChRcv}(\mathsf{chkey}, \mathsf{stkey}, \mathsf{st}_R, c)$
if $m = (m_{\mathsf{sec}}, m_{\mathsf{conf}}, m_{\mathsf{auth}}, m_{\mathsf{plain}}) \neq \perp$ then
  if $\textsc{insider}$ then
    if $\mathtt{ctr}_R > \mathtt{ctr}_S$ or $(m_{\mathsf{sec}}, m_{\mathsf{auth}}) \notin \{\mathcal{M}[\mathtt{ctr}_R], (\diamond^{|m_{\mathsf{sec}}|}, \diamond^{|m_{\mathsf{auth}}|})\}$ then $\textsc{out-of-sync} \leftarrow \mathtt{true}$
  else
    if $\mathtt{ctr}_R > \mathtt{ctr}_S$ or $c \notin \mathcal{C}[\mathtt{ctr}_R]$ then $\textsc{out-of-sync} \leftarrow \mathtt{true}$
  if $\textsc{out-of-sync}$ and $b = 0$ then return $m$
return $\perp$
```

Figure 3: IND-CCA notion for sanitizable stealth channels

.

bits (when given the length information $\ell_{\mathsf{sec}}, \ell_{\mathsf{conf}}$ instead). We thus allow the adversary to also pass the operation mode $\mathsf{mode}^0, \mathsf{mode}^1 \in \{\mathsf{regular}, \mathsf{stealth}\}$ when requesting the encryption of a message pair m^0, m^1 to the left-or-right sender oracle. Since the adversary can control the mode via the send oracle we always let key generation run in mode $\mathsf{stealth}$ in the attack.

Since we opted for the receiver to not know in advance if the sender uses the stealth transportation, we need to account for another potential attack when the adversary is the sanitizer. Namely, the adversary may simply use the channel key chkey and overwrite any information protected under the stealth key stkey. Hence, we exclude this from happening by requiring the adversary to create a new pair $(m_{\mathsf{sec}}, m_{\mathsf{auth}})$ different from $(\diamond^*, \diamond^*)$.

Definition 6.1 (IND-CCA) *For a $\mathcal{MOD}$-sanitizable stealth channel $\mathcal{CH}$ = $(\mathsf{ChKGen}, \mathsf{ChSend}, \mathsf{ChRcv},$ $\mathsf{ChSanit})$ and an adversary $\mathcal{A}$ let*

$$\boldsymbol{Adv}^{\text{IND-CCA}}_{\mathcal{CH},\mathcal{A},\mathcal{MOD}} := \Pr\big[\boldsymbol{Exp}^{\text{IND-CCA}}_{\mathcal{CH},\mathcal{A},\mathcal{MOD}} = 1\big] - \frac{1}{2}$$

for the experiment $\textbf{\textit{Exp}}^{\textit{IND-CCA}}_{\mathcal{CH},\mathcal{A},\mathcal{MOD}}$ *in Figure 3.*

With the usual asymptotic requirement we would now demand that the advantage of every efficient adversary $\mathcal{A}$ is negligible.

6.3 Construction

We next describe the construction of a sanitizable (stealth) channel. It is based on any authenticated encryption schemes with associated data, with some mild additional requirements for the AEAD scheme. We present here first in detail a construction which does not support confidential-only message parts m_{conf} such that we omit this part here (and also omit putting an empty message symbol ϵ for sake of simplicity). We discuss at the end of this part how to extend the construction to also allow for such confidential parts.

Authenticated Encryption with Associated Data. An authenticated encryption scheme with associated data (AEAD) [Rog02] consists of three efficient algorithms, AEKGen for key generation, AEEnc for encryption and AEDec for decryption. The encryption and decryption algorithm take as input a uniformly distributed key $\mathsf{key} \leftarrow\!\!\$\ \mathsf{AEKGen}()$ from some key space $\mathcal{K}$. In addition, the encryption algorithm takes a nonce value nonce, associated data AD, and a message m. It returns a ciphertext $c \leftarrow \mathsf{AEEnc}(\mathsf{key}, \mathsf{nonce}, \mathsf{AD}, m)$. The decryption algorithm takes as input a nonce nonce, associated data AD, and a ciphertext, and outputs a message m or an error symbol.

We assume that both encryption and decryption are deterministic. Furthermore there is a length function $\mathsf{AElen}(|m|)$ which determines the ciphertext output length of AEEnc given the input-message length only. It is convenient for us to define the inverse length function as well, stating that $\mathsf{AElen}^{-1}(\mathsf{AElen}(\ell)) = \ell$ for any input message of length ℓ. We note that these are all properties which Rogaway [Rog02] already assumes as well, and that schemes like GCM and ChaChaPoly obey.

We use Rogaway's original security definitions for AEAD schemes [Rog02]. The first one is IND$-CPA which states that the adversary cannot distinguish ciphertexts $\mathsf{AEEnc}(\mathsf{key}, \mathsf{nonce}, \mathsf{AD}, m) \in \{0,1\}^{\mathsf{AElen}(|m|)}$ from random strings $c \leftarrow\!\!\$\ \{0,1\}^{\mathsf{AElen}(|m|)}$. Formally we can capture this via an experiment $\mathbf{Exp}^{\text{IND-\$CPA}}_{\mathsf{AEAD},\mathcal{A}}$ by picking a key $\mathsf{key} \leftarrow\!\!\$\ \mathsf{AEKGen}()$ and a secret bit $b \leftarrow\!\!\$\ \{0,1\}$, and giving an adversary $\mathcal{A}$ oracle access to $\mathsf{AEEnc}(\mathsf{key}, \cdots)$ if $b = 0$, or to the random sampler if $b = 1$, allowing multiple and adaptive queries $(\mathsf{nonce}, \mathsf{AD}, m)$. The experiment outputs 1 if the adversary predicts b. Let

$$\mathbf{Adv}^{\text{IND\$-CPA}}_{\mathsf{AEAD},\mathcal{A}} := \Pr\Big[\mathbf{Exp}^{\text{IND\$-CPA}}_{\mathsf{AEAD},\mathcal{A}}\Big] - \frac{1}{2}.$$

The other security property defined by Rogaway [Rog02] is (ciphertext) integrity. The corresponding experiment $\mathbf{Exp}^{\text{INT-CTXT}}_{\mathsf{AEAD},\mathcal{A}}$ again picks a key $\mathsf{key} \leftarrow\!\!\$\ \mathsf{AEKGen}()$, and the allows the adversary to query $(\mathsf{nonce}, \mathsf{AD}, m)$ to an encryption oracle. The goal of the adversary is to output a valid ciphertext c and values $\mathsf{nonce}, \mathsf{AD}$ such that $\mathsf{AEDec}(\mathsf{key}, \mathsf{nonce}, \mathsf{AD}, c) \neq \perp$ but such that c was never a response to an encryption query $(\mathsf{nonce}, \mathsf{AD}, m)$. Let

$$\mathbf{Adv}^{\text{INT-CTXT}}_{\mathsf{AEAD},\mathcal{A}} := \Pr\big[\mathbf{Exp}^{\text{INT-CTXT}}_{\mathsf{AEAD},\mathcal{A}}\big].$$

In our proofs we use the fact that we can also consider an adversary which outputs a sequence of q potential forgeries, $(\mathsf{nonce}_i, \mathsf{AD}_i, c_i)$ and wins if one of these ciphertexts is valid and has not been a response to an encryption query before. Shrimpton [Shr04] shows that this increases the advantage by a factor of at most q.

Sanitizable Channel. We next describe our sanitizable channel protocol. The formal description appears in Figure 4. The idea is to use the stealth key stkey within the AEAD scheme to protect confidentiality and integrity of m_{sec}; if there is no stealth key then the sender simply puts random bits. In case of a stealth key we protect the integrity of m_{auth} by including this message part in the authenticated associated data for encrypting m_{sec}. As a nonce we use a counter value. Denote the resulting ciphertext part by c_{sec}.

```
$\mathsf{ChKGen}(\mathsf{mode})$
$\mathsf{chkey} \leftarrow\!\!\$\ \mathsf{AEKGen}()$
$\mathsf{stkey} \leftarrow\!\!\$\ \mathsf{AEKGen}()$
$\textbf{if } \mathsf{mode} \neq \mathsf{stealth} \textbf{ then } \mathsf{stkey} \leftarrow \bot$
$\mathsf{st}_S, \mathsf{st}_R, \mathsf{st}_{\mathrm{San}} \leftarrow 0$
$\textbf{return}$
  $(\mathsf{chkey}, \mathsf{stkey}, \mathsf{st}_S, \mathsf{st}_R, \mathsf{st}_{\mathrm{San}})$
```

```
$\mathsf{ChSend}(\mathsf{chkey}, \mathsf{stkey}, \mathsf{mode}, \mathsf{st}_S, m)$
$m = (m_{\mathsf{sec}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$
$\mathsf{st}_S \leftarrow \mathsf{st}_S + 1$
$\textbf{if } \mathsf{stkey} \neq \bot \textbf{ and } \mathsf{mode} = \mathsf{stealth} \textbf{ then}$
  $c_{\mathsf{sec}} \leftarrow \mathsf{AEEnc}(\mathsf{stkey}, 0\|\mathsf{st}_S, m_{\mathsf{auth}}, m_{\mathsf{sec}})$
$\textbf{else}$
  $c_{\mathsf{sec}} \leftarrow\!\!\$\ \{0,1\}^{\mathsf{AElen}(|m_{\mathsf{sec}}|)}$
$m_{\mathsf{stealth}} \leftarrow (c_{\mathsf{sec}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$
$\mathsf{AD} \leftarrow \mathsf{ad}(|m_{\mathsf{stealth}}|)$
$c \leftarrow \mathsf{AEEnc}(\mathsf{chkey}, 0\|\mathsf{st}_S, \mathsf{AD}, m_{\mathsf{stealth}})$
$\textbf{return } c$
```

```
$\mathsf{ChSanit}(\mathsf{chkey}, \mathsf{st}_{\mathrm{San}}, c, \textsc{Mod})$
$\textbf{if } \mathsf{st}_{\mathrm{San}} = \bot \textbf{ or } \textsc{Mod} \notin \mathcal{MOD} \textbf{ then return } \bot$
$\mathsf{AD} \leftarrow \mathsf{ad}^{-1}(|c|)$
$\mathsf{st}_{\mathrm{San}} \leftarrow \mathsf{st}_{\mathrm{San}} + 1$
$m_{\mathsf{stealth}} \leftarrow \mathsf{AEDec}(\mathsf{chkey}, 0\|\mathsf{st}_{\mathrm{San}}, \mathsf{AD}, c)$
$\textbf{if } m_{\mathsf{stealth}} = \bot \textbf{ then}$
  $\mathsf{st}_{\mathrm{San}} \leftarrow \bot$
  $\textbf{return } \bot$
$m_{\mathsf{san}} \leftarrow \textsc{Mod}(m_{\mathsf{stealth}})$
$\mathsf{AD} \leftarrow \mathsf{ad}(|m_{\mathsf{san}}|)$
$c_{\mathrm{San}} \leftarrow \mathsf{AEEnc}(\mathsf{chkey}, 1\|\mathsf{st}_{\mathrm{San}}, \mathsf{AD}, m_{\mathsf{san}})$
$\textbf{return } c_{\mathrm{San}}$
```

```
$\mathsf{ChRcv}(\mathsf{chkey}, \mathsf{stkey}, \mathsf{st}_R, c)$
$\textbf{if } \mathsf{st}_R = \bot \textbf{ then return } \bot$
$\mathsf{st}_R \leftarrow \mathsf{st}_R + 1$
$\mathsf{AD} \leftarrow \mathsf{ad}^{-1}(|c|)$
$m \leftarrow \mathsf{AEDec}(\mathsf{chkey}, 0\|\mathsf{st}_R, \mathsf{AD}, c)$
$\textbf{if } m = \bot \textbf{ then}$
  $m \leftarrow \mathsf{AEDec}(\mathsf{chkey}, 1\|\mathsf{st}_R, \mathsf{AD}, c)$
$\textbf{if } m = \bot \textbf{ then}$
  $\mathsf{st}_R \leftarrow \bot$
  $\textbf{return } \bot$
$m = (c_{\mathsf{sec}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$
$\textbf{if } \mathsf{stkey} \neq \bot \textbf{ then}$
  $m_{\mathsf{sec}} \leftarrow \mathsf{AEDec}(\mathsf{stkey}, 0\|\mathsf{st}_R, m_{\mathsf{auth}}, c_{\mathsf{sec}})$
$\textbf{else}$
  $m_{\mathsf{sec}} \leftarrow \bot$
$\textbf{if } m_{\mathsf{sec}} = \bot \textbf{ then}$
  $m_{\mathsf{sec}} \leftarrow \diamond^{\mathsf{AElen}^{-1}(|c_{\mathsf{sec}}|)}$
  $m_{\mathsf{auth}} \leftarrow \diamond^{|m_{\mathsf{auth}}|}$
$\textbf{return } (m_{\mathsf{sec}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$
```

Figure 4: Sanitizable Channel Protocol based on AEAD scheme.

We will use a counter to update the nonces for the encryption steps. Since the sender and the sanitizer share the channel key chkey, and the sanitizer may re-encrypt the data, we use a one-bit prefix and $0\|\mathtt{ctr}$ if the sender needs a nonce, and $1\|\mathtt{ctr}$ for the sanitizer. We note that for TLS 1.3 encryption and decryption use a random offset which can be considered to formally be a part of the key (such that the encryption and decryption process first xor the offset to the counter value). For sake of compatibility we also use a one-bit prefix $0\|\mathtt{ctr}$ for the nonce of the inner stealth encryption, although we never need 1-prefixes anywhere.

Finally, the message parts m_{auth} and m_{plain} are added to c_{sec} in plain. Then we use the channel key chkey, known also by the sanitizer, to encrypt the "message" $(c_{\mathsf{sec}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$ under chkey for associated data AD and extended counter value $0\|\mathtt{ctr}$. Note that the sanitizer can access the encapsulated "message" if it knows the correct counter value and associated data. For the associated data we assume that they are computable from the length of the input message resp. recoverable from the length of the ciphertext. This matches the approach in the TLS 1.3 record protocol where the associated data consists of constants and the length of ciphertext. Formally, we thus have a function $\mathsf{AD} \leftarrow \mathsf{ad}(|m|)$ for encryption and $\mathsf{AD} \leftarrow \mathsf{ad}^{-1}(|c|)$ with the idea that $\mathsf{ad}(|m|) = \mathsf{ad}^{-1}(|c|)$ for any valid ciphertext c for the message m.

Once the outer encryption is undone with the help of chkey, the sanitizer can apply arbitrary operations on m_{plain}. The modification options are described the admissible operations $\textsc{Mod}$, forming the set $\mathcal{MOD}$. We note that the sanitizer re-encrypts the entire message, consisting of the unaltered c_{sec} and m_{auth}, and the modified m_{plain} part with the AEAD scheme for key chkey, counter value $1\|\mathtt{ctr}$, and associated data AD.

The receiver will try both possibilities to decrypt, under counter value $0\|\mathsf{st}_R$ (for sender ciphertexts) and $1\|\mathsf{st}_R$ (for sanitized ciphertexts), and work with the message for which decryption succeeds. We remark that for a random ciphertext decryption will fail with overwhelming probability such that, strictly speaking, our scheme has a negligible decryption error. If both decryptions fail then the receiver closes the channel by setting $\mathsf{st}_R \leftarrow \perp$. Note that, by construction, our solution thus requires that the counter value of the sanitizer and the receiver are in sync. This means that the sanitizer in our solution needs to at least learn about each ciphertext sent to the receiver.

Extension to Confidential Message Parts. We outline here how we could incorporate confidential message parts into the construction. Assume that the stealth key stkey consists of two parts, $\mathsf{stkey}_{\mathsf{sec}}$ and $\mathsf{stkey}_{\mathsf{conf}}$, e.g., by stretching the key stkey pseudorandomly. Then we use the key $\mathsf{stkey}_{\mathsf{sec}}$ as before in the AEAD scheme for securing m_{sec} and authenticating m_{auth} via ciphertext part c_{sec}. In addition, we use a key derivation function KDF for key $\mathsf{stkey}_{\mathsf{conf}}$, label $0\|\mathsf{st}_S$ and length parameter $|m_{\mathsf{conf}}|$ to generate a pseudorandom output, which we xor to m_{conf} to get a ciphertext part c_{conf}. Finally, one uses the channel key chkey as before to encrypt $(c_{\mathsf{sec}}, c_{\mathsf{conf}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$ under the AEAD scheme.

The sanitizer, knowing the channel key chkey but not the stealth keys $\mathsf{stkey}_{\mathsf{sec}}, \mathsf{stkey}_{\mathsf{conf}}$, can access $(c_{\mathsf{sec}}, c_{\mathsf{conf}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$ for a transmitted ciphertext, and can then modify the unauthenticated parts m_{plain} and c_{conf}. For the selected encryption of m_{conf} via xor this means that the sanitizer can for instance randomly flip bits to invalidate some parts of m_{conf}. If we choose any other symmetric-key encryption scheme for creating c_{conf}, then the sanitizer could perform other compliant operations. The sanitizer would then re-encrypt the resulting elements under chkey again.

6.4 Security Proof

We next show security of our construction in Figure 4 in the previous section (without the extension to confidential message parts) for arbitrary modifications on the plain part m_{plain}. That is, we consider the set

$$\mathcal{MOD}_{\mathsf{plain}} = \left\{ \textsc{Mod} \;\middle|\; \textsc{Mod}(m_{\mathsf{sec}}, m_{\mathsf{auth}}, m_{\mathsf{plain}}) = (m_{\mathsf{sec}}, m_{\mathsf{auth}}, m'_{\mathsf{plain}}) \right\}.$$

Recall that the security experiment requires the modification to be length-preserving, meaning here that the modified message m'_{plain} needs to be as long as m_{plain}.

Theorem 6.2 *The sanitizable channel protocol in Figure 4 is an IND-CCA secure $\mathcal{MOD}_{\mathsf{plain}}$-sanitizable stealth channel if the AEAD scheme* AEAD *is IND\$-CPA and INT-CTXT. More precisely, for any adversary $\mathcal{A}$ against the sanitizable stealth channel, making in total at most q queries to the sanitization and receiver oracle, there exist adversaries $\mathcal{B}_{out}$, $\mathcal{C}_{out}$, $\mathcal{B}_{in}$, and $\mathcal{C}_{in}$ (with roughly the same running time as $\mathcal{A}$) such that*

$$\begin{aligned} \mathbf{Adv}^{\textit{IND-CCA}}_{\mathcal{CH},\mathcal{A}} \;\leq\;& 2q \cdot \mathbf{Adv}^{\textit{INT-CTXT}}_{\mathsf{AEAD},\mathcal{B}_{out}} + 2 \cdot \mathbf{Adv}^{\textit{IND\$-CPA}}_{\mathsf{AEAD},\mathcal{C}_{out}} + \\ & 2q \cdot \mathbf{Adv}^{\textit{INT-CTXT}}_{\mathsf{AEAD},\mathcal{B}_{in}} + 2 \cdot \mathbf{Adv}^{\textit{IND\$-CPA}}_{\mathsf{AEAD},\mathcal{C}_{in}}. \end{aligned}$$

Proof. We distinguish between the two attack strategies, when $\mathcal{A}$ acts as an outsider (not requesting chkey at the outset) resp. as an insider (learning chkey at the beginning and triggering INSIDER to be set to $\mathtt{true}$).

Outsider Attacks. We start with $\mathcal{A}$ mounting an outsider attack. In this case we play against the AEAD scheme for key chkey, formally describing a reduction $\mathcal{B}_{\text{out}}$ which uses $\mathcal{A}$ and its attack on the channel protocol against the INT-CTXT and IND\$-CPA properties of AEAD. Our first step is to argue that the adversary $\mathcal{A}$ can never make OUT-OF-SYNC become $\mathtt{true}$, unless one breaks integrity of the AEAD scheme. Also, the adversary never manages to submit a valid ciphertext c to the sanitizer oracle which has not been the response of the sender oracle for the same counter value. To this end we build the following reduction $\mathcal{B}_{\text{out}}$ against INT-CTXT property of the AEAD scheme

- Algorithm $\mathcal{B}_{\text{out}}$ generates another key $\mathsf{stkey} \leftarrow\!\!\$\ \mathsf{AEKGen}()$ and picks the challenge bit $b \leftarrow\!\!\$\ \{0,1\}$ internally. Algorithm $\mathcal{B}_{\text{out}}$ also initializes the counter values $\mathsf{st}_S, \mathsf{st}_R, \mathsf{st}_{\text{San}}$ as in the scheme, and the game's counter values $\mathtt{ctr}_S, \mathtt{ctr}_R$. It also initializes the arrays $\mathcal{C}[\,]$ and $\mathcal{M}[\,]$ as in the game to be empty, and two other internal arrays $\mathcal{C}_{\mathsf{red}}[\,]$ and $\mathcal{M}_{\mathsf{red}}[\,]$ also to be empty.
- When $\mathcal{A}$ makes a call $(\mathsf{mode}^0, m^0, \mathsf{mode}^1, m^1)$ to its ChSend oracle then $\mathcal{B}_{\text{out}}$ simulates the oracle as follows: $\mathcal{B}_{\text{out}}$ immediately returns $\perp$ if $|m^0|_\forall = |m^1|_\forall$ does not hold. Else it increments st_S and creates c^b_{sec} from m^b_{sec} as in the protocol. Here, the ciphertexts may be picked randomly, if the corresponding mode mode^b equals $\mathsf{regular}$. Algorithm $\mathcal{B}_{\text{out}}$ next creates the "stealthified" message $m^b_{\mathsf{stealth}} \leftarrow (c^b_{\mathsf{sec}}, m^b_{\mathsf{auth}}, m^b_{\mathsf{plain}})$ and calls its encryption oracle for the unknown key chkey about nonce $0\|\mathsf{st}_S$, associated data $\mathsf{AD} = \mathsf{ad}(|m^b_{\mathsf{stealth}}|)$, and the message m^b_{stealth} to get a ciphertext c. It returns this ciphertext c to $\mathcal{A}$, increments $\mathtt{ctr}_S$ and stores c in $\mathcal{C}[\mathtt{ctr}_S]$ as well as the message m^b_{stealth} in $\mathcal{M}[\mathtt{ctr}_S]$.
- When $\mathcal{A}$ calls the sanitize oracle about a ciphertext c and modifications $\textsc{Mod}^0, \textsc{Mod}^1$, then $\mathcal{B}_{\text{out}}$ first increments st_{San} and puts $(0\|\mathsf{st}_{\text{San}}, \mathsf{AD}, c)$ for $\mathsf{AD} \leftarrow \mathsf{ad}^{-1}(|c|)$ as a potential forgery in its list. Then $\mathcal{B}_{\text{out}}$ checks that $c = \mathcal{C}[\mathsf{st}_{\text{San}}]$. If not, then $\mathcal{B}$ aborts. Else it recovers $m \leftarrow \mathcal{M}[\mathsf{st}_{\text{San}}]$, applies $m_{\mathsf{san}} \leftarrow \textsc{Mod}^b(m)$, and calls its encryption oracle about $(1\|\mathsf{st}_{\text{San}}, \mathsf{AD}, m_{\mathsf{san}})$ for $\mathsf{AD} \leftarrow \mathsf{ad}(|m_{\mathsf{san}}|)$ to get a ciphertext c_{San}. It returns c_{San} to $\mathcal{A}$ and stores c_{San} in $\mathcal{C}_{\mathsf{red}}[\mathsf{st}_{\text{San}}]$ and m_{san} in $\mathcal{M}_{\mathsf{red}}[\mathsf{st}_{\text{San}}]$.
- When $\mathcal{A}$ calls the receiving oracle about a ciphertext c then $\mathcal{B}_{\text{out}}$ first checks that $\mathsf{st}_R \neq \perp$ and then increments st_R. Then it checks if c is in $\mathcal{C}[\mathsf{st}_R]$ and, if so sets $m \leftarrow \mathcal{M}[\mathsf{st}_R]$. Else it checks if c equals $\mathcal{C}_{\mathsf{red}}[\mathsf{st}_R]$ and, if so, set $m \leftarrow \mathcal{M}_{\mathsf{red}}[\mathsf{st}_R]$. In any other case it sets $m \leftarrow \perp$ and continues as in the game. In any case it adds $(0\|\mathsf{st}_R, \mathsf{AD}, c)$ and $(1\|\mathsf{st}_R, \mathsf{AD}, c)$ for $\mathsf{AD} \leftarrow \mathsf{ad}^{-1}(|c|)$ to its list of potential forgeries.

This concludes the description of our adversary $\mathcal{B}$. We note that for $\mathcal{A}$ to make OUT-OF-SYNC $= \mathtt{true}$ as an outsider, it needs to provide a ciphertext c sent to the receiver oracle which has not been created for the counter value by the sender nor by the sanitizer. Here we use the fact that the local counter values correspond exactly to the game's values $\mathtt{ctr}_S$ and $\mathtt{ctr}_R$. Analogously, a new valid ciphertext c submitted to the sanitizer would equally be found by $\mathcal{B}_{\text{out}}$. It follows that $\mathcal{B}_{\text{out}}$ will capture such a forgery (for empty associated data) in its list of at most $2q$ decryption processes, and thus succeeds in its integrity experiment with the same advantage as $\mathcal{A}$ does in triggering OUT-OF-SYNC $= \mathtt{true}$, times $2q$.

With the above reduction we now have that $\mathcal{A}$ never makes the receiver oracle return anything but $\perp$. We can thus easily simulate this oracle from now on. Accordingly, we can always find the correct message m in $\mathcal{M}[\mathsf{st}_{\text{San}}]$ for sanitizing the ciphertext, such that we do not need access to the decryption function for key chkey anymore in the entire attack. The next step is now obvious and uses the IND\$-CPA property: Whenever the game is now supposed to create a ciphertext under key chkey, we sample a uniform bit string of the corresponding length instead. We can easily turn this into a reduction $\mathcal{C}_{\mathrm{out}}$ with oracle access to the encryption function or the random sampler. We skip the details since they are straightforward.

In this final game the adversary $\mathcal{A}$ is now perfectly oblivious about the secret bit b and cannot do better than guessing.

Insider Attacks. We next consider the case that the adversary $\mathcal{A}$ asks for the channel key chkey at the beginning of the experiment and makes $\textsc{insider}$ being set to $\mathtt{true}$. The strategy is identical to the outsider case. We first show, via a reduction $\mathcal{B}_{\mathrm{in}}$ to the INT-CTXT property of the AEAD scheme for key stkey, that the adversary $\mathcal{A}$ cannot make $\textsc{out-of-sync}$ being set to $\mathtt{true}$ via "bad" decryption queries. Note that we do not need to take care of the decryption queries in the sanitization step, because this only involves the channel key chkey known by $\mathcal{B}_{\mathrm{in}}$. Furthermore, the admissible modifications $\mathcal{MOD}$ only affect the public part m_{plain}. In more detail:

- Algorithm $\mathcal{B}_{\mathrm{in}}$ generates the channel key $\mathsf{chkey} \leftarrow\!\!\$\ \mathsf{AEKGen}()$ itself and also selects the random challenge bit $b \leftarrow\!\!\$\ \{0,1\}$. It initializes the counter values $\mathsf{st}_S, \mathsf{st}_R, \mathsf{st}_{\mathrm{San}}$ as in the scheme, and the counter values $\mathtt{ctr}_S, \mathtt{ctr}_R$, as well as the arrays $\mathcal{C}[]$ and $\mathcal{M}[]$.
- When $\mathcal{A}$ queries its send oracle about $(\mathsf{mode}^0, m^0, \mathsf{mode}^1, m^1)$ then $\mathcal{B}_{\mathrm{in}}$ uses its encryption oracle to compute c_{sec} (or, samples it at random if $\mathsf{mode}^b = \mathsf{regular}$) and proceeds otherwise as in the game. It stores the final intermediate ciphertexts c_{sec} in $\mathcal{C}_{\mathsf{red}}[\mathsf{st}_S]$ and the original input message m^b in $\mathcal{M}_{\mathsf{red}}[\mathsf{st}_S]$.
- If $\mathcal{A}$ calls the sanitization oracle about a ciphertext c and two operations $\textsc{Mod}^0, \textsc{Mod}^1$, then $\mathcal{B}_{\mathrm{in}}$ simply executes the protocol steps with knowledge of chkey. Note that this is possible since all operations can be carried out on the plain part m_{plain}. Furthermore, the c_{sec} part remains unchanged for $\mathcal{MOD}_{\mathsf{plain}}$.
- When $\mathcal{A}$ calls the receiving oracle about c then $\mathcal{B}_{\mathrm{in}}$ runs the first steps according to the protocol. In particular,it obtains a message $m_{\mathsf{stealth}} = (c_{\mathsf{sec}}, m_{\mathsf{auth}}, m_{\mathsf{plain}})$. If c_{sec} does not match $\mathcal{C}[\mathsf{st}_R]$ then $\mathcal{B}_{\mathrm{in}}$ outputs the tuple $(0\|\mathsf{st}_R, m_{\mathsf{auth}}, c_{\mathsf{sec}})$ to its list of potential forgeries and sets $\mathsf{st}_R \leftarrow \perp$ and returns $\perp$. Else, if the value matches, then $\mathcal{B}_{\mathrm{in}}$ looks up m_{sec} in $\mathcal{M}[\mathsf{st}_R]$ and uses this value to complete the steps of the receiving oracle.

Our adversary $\mathcal{B}_{\mathrm{in}}$ perfectly simulates the game for $\mathcal{A}$, up to a step where $\mathcal{A}$ potentially forces a valid forgery c_{sec} in the receiving oracle. However, in order to make $\textsc{out-of-sync} = \mathtt{true}$, the adversary would need to make $(m_{\mathsf{sec}}, m_{\mathsf{auth}})$ to deviate from the stored values (or use a fresh counter value). In either case the inner ciphertext must be valid or else $m_{\mathsf{sec}}, m_{\mathsf{auth}} \in \diamond^*$ and the event is not triggered. If the counter value is new or m_{auth} as the associated data is new, we immediately get a contradiction to the integrity game. If only m_{sec} is new, then by the completeness of the AEAD scheme the ciphertext part c_{sec} cannot match the value stores in $\mathcal{C}[\mathsf{st}_R]$ for the original message. Hence, this also breaks integrity.

The final step, now that we eliminated each application of the decryption key stkey through lookups or by using $\perp$, we can once more give a reduction $\mathcal{C}_{\mathrm{in}}$ to the IND\$-CPA property of the encryption part $\mathsf{AEEnc}(\mathsf{stkey}, \cdots)$. In this step we exploit the fact that for insiders the parts $(m^0_{\mathsf{auth}}, m^0_{\mathsf{plain}}) = (m^1_{\mathsf{auth}}, m^1_{\mathsf{plain}})$ must be equal such that $\mathcal{C}_{\mathrm{in}}$ can simulate this part without knowledge of the challenge bit b. □

6.5 Read-Only Access in the Record Protocol by TLS

In this section we argue that a read-only sanitizer, i.e., which may access m_{stealth} but does not modify it to any m_{san}, can be easily embedded into the TLS 1.3 record protocol. Note that we can enforce read-only access by putting covertly sent data in $m_{\textsf{sec}}$ and immutable parts in$m_{\textsf{auth}}$ according to out terminology, letting the receiver only accept empty $m_{\textsf{plain}}$-parts.

Recall that TLS 1.3 uses random offsets `client_write_iv` resp. `server_write_iv` which are added to the counter value and then used as nonce. Formally, we assume that these offsets are part of the keys $\textsf{chkey}$ resp. $\textsf{stkey}$ —which indeed coincides with the key deriviation process in the TLS 1.3 handshake. In this sense it is understood that the authenticated encryption for the extended key $(\textsf{key}, \textsf{offset})$ encrypts as $\textsf{AEEnc}(\textsf{key}, \textsf{nonce} \oplus \textsf{offset}, \textsf{AD}, m)$ and decryption works correspondingly. Note that since the nonce values are under adversarial control in the AEAD security experiments anyway, this does not weaken the security of the AEAD scheme.

Next, recall that the TLS 1.3 record protocol uses as associated data the concatentation of the constant `ContentType opaque_type = application_data; /* 23 */` and the constant `ProtocolVersion legacy_record_version = 0x0303;`, followed by the (expected) length of the ciphertext in bytes. Hence, given the message for encryption one can deduce the ciphertext length, and given the ciphertext length the value is readily available anyway. We can therefore easily define our functions $\textsf{ad}$ and $\textsf{ad}^{-1}$ for computing the associated data from the message resp. ciphertext length, as required by our scheme.

Finally, note that for read-only sanitizers we can omit the prefix bit 0 or 1 for the counters and work with the plain counter value directly when encrypting and decrypting. This does not weaken the overall security of our channel protocol if the sanitizer only has rad-only access and can never modifies the message $m_{\textsf{stealth}}$. It follows that the outer channel encryption in our general scheme, with the choices above, is a valid TLS 1.3 record protocol message.

There are, however, two things to consider regarding the length of the nested ciphertext c. First note that, compared to subliminal communication, an outsider can observe that ciphertexts in this version are longer than when using the original record protocol. As explained inthe introducton, we do not aim to hide this fact. Secondly, TLS 1.3 sets an upper bound of $2^{14} + 256$ bytes for the length of ciphertexts, requiring that input messages are of at most 2^{14} bytes (or else need to be fragmented) [Res18]. This needs to be taken into account with the ciphertext expansion due to the double encryption here. Indeed, we need to make sure that the combined length of $(c_{\textsf{sec}}, m_{\textsf{auth}})$ is at most 2^{14} bytes, resulting in an overall bound of $2^{14} - 256$ for $m_{\textsf{sec}}$ and $m_{\textsf{auth}}$ and possibly further fragmentations. Let us stress once more that our goal is not to hide the fact that we are using the stealth channel. If this is obeyed, then c is a perfectly legit TLS 1.3 record protocol ciphertext which supports read-only access for the sanitizer.

7 Towards Integration into Intrusion Detection Systems

In this section we describe how one can use our sanitizable channels in combination with a network intrusion detection and prevention system like the well-known open-source system Snort (`https://www.snort.org/`). We assume that the keys have already been established, as described in Section 6.1 about setting up the sanitizable channel. The reader may for now think of the intrusion detection system using a static Diffie-Hellman key which the receiver obtains when logging into the local network, and which is then used in the stealth key exchange step to establish the channel key (accessible also by the intrusion detection system). The stealth key is only available to the sender and receiver.

Snort in version 3 comes with a set of 4,031 predefined rules, called the Community Ruleset. This set is updated frequently, we refer here to the one of February 6th, 2023. The rules allow to detect malicious network behavior of various types. As an example, consider the rule with identifier *sid 26261* for detecting potential phishing attacks (parts omitted for readability):

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any ( msg: ↩
"MALWARE-OTHER Fake postal receipt HTTP Response phishing attack"; ↩
flow:to_client,established; http_header; content: ↩
```

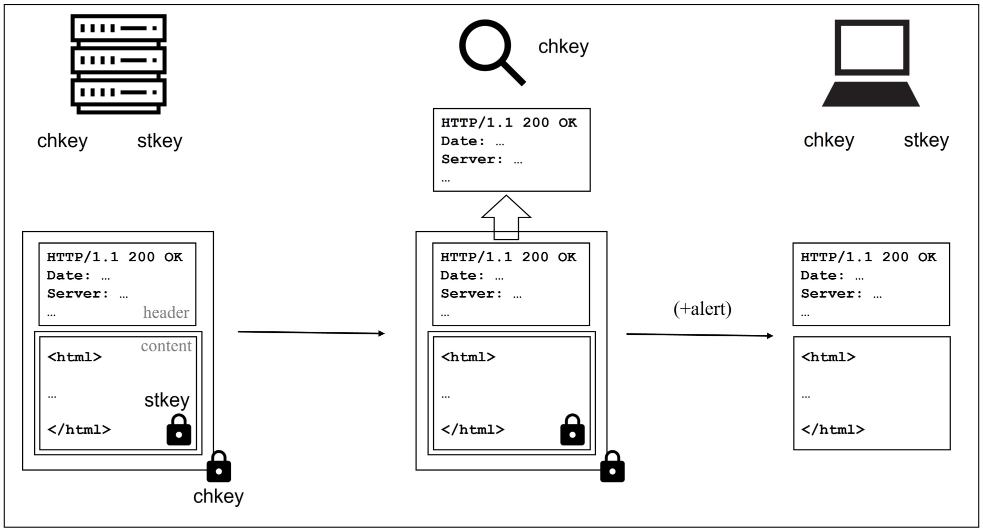

Figure 5: IDS checking HTTP header information in sanitizable channel.

```
"|3B 20|filename=Postal-Receipt.zip|0D 0A|",fast_pattern,nocase; ↩
... classtype:trojan-activity; sid:26261; rev:3; )
```

The rule checks if the incoming network traffic on HTTP ports contains suspicious file names in the HTTP header. The HTTP header contains meta-information about the actual HTTP content and the sending party. In secured HTTPS connections the header is also encrypted and thus inaccessible to an intrusion detection system like Snort.

With our sanitizable channel protocol, combined with the stealth key exchange, we could give Snort as the sanitizer access to the HTTP header information (and similar meta-data such as the HTTP status code and URI) by placing this information into the m_{plain}-part or m_{auth}-part, protected under the channel key chkey shared also with the sanitizer. We put the HTTP content into the inner m_{sec}-part, protected by the outer channel key chkey as well as the inner stealth key stkey only known by the sender and receiver (see Figure 5). Then Snort can access the header information and apply qualified rules, whereas the actual HTTP content remains hidden from Snort. From the outside, the communication still appears to be a valid HTTPS resp. TLS connection, integrating smoothly into existing network environments.

To estimate the usefulness we note that the Community Ruleset currently lists roughly half of the rules with reference to HTTP fields `http_*` (altogether 2,011 rules). Of this set, 470 rules use the `http_header` field and *no* reference to the body `http_client_body`. If we also grant Snort access to other HTTP data such as the URI in outgoing traffic via the `http_uri` flag, or the `http_cookie` flag for Cookie header information, then the coverage increases significantly. Among the Community Ruleset, 1,776 rules include one of the `http_*` fields without listing `http_client_body`. These are 44% of all rules and 88% of all HTTP-related rules.

The solution still comes with some inconveniences, though. First of all, one carefully needs to evaluate if revealing the HTTP information to Snort is admissible. Secondly, scanning the content is still not possible. Third, HTTPS currently does not differentiate between confidentiality levels for the HTTP parts and one would thus need to change the protocol in order to accommodate the specification of different confidentiality levels for data.

8 Conclusion

Our results show that, with some extra effort, existing cryptographic mechanisms can be enhanced to enable further features. As for the overhead, we note that we did some initial experiments for the stealth key exchange on commodity hardware. The computational costs in our experiments went up by roughly a factor 2.5 compared to the plain TLS 1.3 handshake protocol. This matches the expected overhead from theory, since one runs roughly two TLS key exchanges, plus Elligator needs two attempts to find a suitable point on the average, plus inversion time for the embedding. Based on the results in [PST20] about using post-quantum primitives in TLS connections for various network settings, it is plausible that the common network latency will also level out the slowdown due to our stealth computations.

We stress again that the changes to achieve stealthiness in TLS 1.3 require protocol modifications at the end points but not on the network layer. That is, the protocol is fully compatible with common TLS 1.3 network traffic. Still, integrating the sanitizable channel to enable HTTPS scanning as explained in Section 7 asks for modifications on the application-channel interface, for both the HTTPS part—semantically labeling header and body data— as well as on the TLS channel side, processing the different inputs parts accordingly. This certainly poses further engineering challenges. It is, however, beyond our cryptographic treatment here, showing that a graceful access, being fully under control of the sending party, is cryptographically possible.

An interesting prospect in light of stealth channels is the planned deployment of TLS hybrid key exchange protocols, as discussed for example by the IETF [SFG23]. In such a hybrid solution one runs a classical key exchange protocol, e.g., based on Diffie-Hellman, together with a quantum-resistant one, e.g., based on Kyber as suggested in [SFG23]. In this case the design would already generate two keys, and for the common cases one could now implement a stealth version within the given system. For this one either uses the Diffie-Hellman part, either generating a known secret key x by sending g^x, or refraining from doing so by sending $\mathsf{Embd}^{-1}(\mathsf{nonce})$ instead. Alternatively, one could also use the post-quantum part for the same purpose. The latter is possible since Kyber provides strong pseudorandomness under chosen-ciphertext attacks (SPR-CCA) [MX23], meaning that outsiders cannot distinguish actual ciphertexts from random strings.[4] In other words, a sender could deny to be able to compute the shared key in the classical or the quantum-secure part of the protocol. However, as opposed to our solution here which *preserves* security of the original protocol, the sketched hybrid solution would actually degrade security, although starting from a higher level. That is, the resulting scheme would only be classically secure or be post-quantum secure when using only one key, but would fail to give fallback security—which is the original idea of using hybrid schemes.

Acknowledgments

We thank the anonymous reviewers and Felix Günther for valuable comments. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297 and by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

[4] Another interesting feature in the Kyber case is that the receiver may actually become aware of the sender's choice by trying to decrypt.

References

[ACdMT05] Giuseppe Ateniese, Daniel H. Chou, Breno de Medeiros, and Gene Tsudik. Sanitizable signatures. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *ESORICS 2005*, volume 3679 of *LNCS*, pages 159–177. Springer, Heidelberg, September 2005.

[AFQ+14] Diego F. Aranha, Pierre-Alain Fouque, Chen Qian, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. Binary elligator squared. In Antoine Joux and Amr M. Youssef, editors, *SAC 2014*, volume 8781 of *LNCS*, pages 20–37. Springer, Heidelberg, August 2014.

[AP98] Ross J. Anderson and Fabien A. P. Petitcolas. On the limits of steganography. *IEEE J. Sel. Areas Commun.*, 16(4):474–481, 1998.

[BBD+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552. IEEE Computer Society Press, May 2015.

[BC05] Michael Backes and Christian Cachin. Public-key steganography with active attacks. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 210–226. Springer, Heidelberg, February 2005.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006.

[BFK16] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. mitls: Verifying protocol implementations against real-world attacks. *IEEE Secur. Priv.*, 14(6):18–25, 2016.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, November 2013.

[BKN04] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and provably repairing the SSH authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Trans. Inf. Syst. Secur.*, 7(2):206–241, 2004.

[BL18] Sebastian Berndt and Maciej Liskiewicz. On the gold standard for security of universal steganography. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 29–60. Springer, Heidelberg, April / May 2018.

[BR94] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Heidelberg, August 1994.

[CHH+17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1773–1788. ACM Press, October / November 2017.

[CHSv16] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy*, pages 470–485. IEEE Computer Society Press, May 2016.

[Cra98] Scott Craver. On public-key steganography in the presence of an active warden. In David Aucsmith, editor, *Information Hiding, Second International Workshop, Portland, Oregon, USA, April 14-17, 1998, Proceedings*, volume 1525 of *Lecture Notes in Computer Science*, pages 355–368. Springer, 1998.

[CSFP20] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020*, pages 1416–1432. IEEE, 2020.

[dCdCM16] Xavier de Carné de Carnavalet and Mohammad Mannan. Killed by proxy: Analyzing client-end TLS interce. In *23nd Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2016.

[dCdCvO20] Xavier de Carné de Carnavalet and Paul C. van Oorschot. A survey and analysis of TLS interception mechanisms and motivations. *CoRR*, abs/2010.16388, 2020.

[DDGJ22] Hannah Davis, Denis Diemert, Felix Günther, and Tibor Jager. On the concrete security of TLS 1.3 PSK mode. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 876–906. Springer, Heidelberg, May / June 2022.

[DFGS15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1197–1210. ACM Press, October 2015.

[DFGS21] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology*, 34(4):37, October 2021.

[DFK$^+$17] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy*, pages 463–482. IEEE Computer Society Press, May 2017.

[DG21] Hannah Davis and Felix Günther. Tighter proofs for the SIGMA and TLS 1.3 key exchange protocols. In Kazue Sako and Nils Ole Tippenhauer, editors, *Applied Cryptography and Network Security (ACNS), 2021*, volume 12727 of *Lecture Notes in Computer Science*, pages 448–479. Springer, 2021.

[DHO16] Ivan Damgård, Helene Haagh, and Claudio Orlandi. Access control encryption: Enforcing information flow with cryptography. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 547–576. Springer, Heidelberg, October / November 2016.

[DIRR09] Nenad Dedic, Gene Itkis, Leonid Reyzin, and Scott Russell. Upper and lower bounds on black-box steganography. *Journal of Cryptology*, 22(3):365–394, July 2009.

[DJ21] Denis Diemert and Tibor Jager. On the tight security of TLS 1.3: Theoretically sound cryptographic parameters for real-world deployments. *Journal of Cryptology*, 34(3):30, July 2021.

[FF15] Victoria Fehr and Marc Fischlin. Sanitizable signcryption: Sanitization over encrypted data (full version). Cryptology ePrint Archive, Report 2015/765, 2015. `https://eprint.iacr.org/2015/765`.

[FG14] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1193–1204. ACM Press, November 2014.

[FGKO17] Georg Fuchsbauer, Romain Gay, Lucas Kowalczyk, and Claudio Orlandi. Access control encryption for equality, comparison, and more. In Serge Fehr, editor, *PKC 2017, Part II*, volume 10175 of *LNCS*, pages 88–118. Springer, Heidelberg, March 2017.

[FJT13] Pierre-Alain Fouque, Antoine Joux, and Mehdi Tibouchi. Injective encodings to elliptic curves. In Colin Boyd and Leonie Simpson, editors, *ACISP 13*, volume 7959 of *LNCS*, pages 203–218. Springer, Heidelberg, July 2013.

[GAZ$^+$21] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. *IACR Cryptol. ePrint Arch.*, page 1022, 2021.

[GDH$^+$17] Matthew Green, Ralph Droms, Russ Housley, Paul Turner, and Steve Fenter. Data Center use of Static Diffie-Hellman in TLS 1.3. Internet-Draft draft-green-tls-static-dh-in-tls13-01, Internet Engineering Task Force, July 2017. Work in Progress.

[HNCB11] Amir Houmansadr, Giang T. K. Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: circumvention infrastructure using router redirection with plausible deniability. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 187–200. ACM Press, October 2011.

[Hop05] Nicholas Hopper. On steganographic chosen covertext security. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 311–323. Springer, Heidelberg, July 2005.

[KEJ$^+$11] Josh Karlin, Daniel Ellard, Alden W. Jackson, Christine E. Jones, Greg Lauer, David Mankins, and W. Timothy Strayer. Decoy routing: Toward unblockable internet communication. In Nick Feamster and Wenke Lee, editors, *USENIX Workshop on Free and Open Communications on the Internet, FOCI '11, San Francisco, CA, USA, August 8, 2011*. USENIX Association, 2011.

[KMO$^+$15] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. (De-)constructing TLS 1.3. In Alex Biryukov and Vipul Goyal, editors, *INDOCRYPT 2015*, volume 9462 of *LNCS*, pages 85–102. Springer, Heidelberg, December 2015.

[KPP$^+$23] Miroslaw Kutylowski, Giuseppe Persiano, Duong Hieu Phan, Moti Yung, and Marcin Zawada. Anamorphic signatures: Secrecy from a dictator who only permits authentication!, 2023.

[KW16] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 81–96. IEEE, 2016.

[KW17] Sam Kim and David J. Wu. Access control encryption for general policies from standard assumptions. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 471–501. Springer, Heidelberg, December 2017.

[LK06] Tri Van Le and Kaoru Kurosawa. Bandwidth optimal steganography secure against adaptive chosen stegotext attacks. In Jan Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee, editors, *Information Hiding, 8th International Workshop, IH 2006, Alexandria, VA, USA, July 10-12, 2006. Revised Selcted Papers*, volume 4437 of *Lecture Notes in Computer Science*, pages 297–313. Springer, 2006.

[LSL$^+$19] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted Taekyoung Kwon. matls: How to make TLS middlebox-aware? In *NDSS*. The Internet Society, 2019.

[Möl04] Bodo Möller. A public-key encryption scheme with pseudo-random ciphertexts. In Pierangela Samarati, Peter Y. A. Ryan, Dieter Gollmann, and Refik Molva, editors, *ESORICS 2004*, volume 3193 of *LNCS*, pages 335–351. Springer, Heidelberg, September 2004.

[MV04] David A. McGrew and John Viega. The security and performance of the Galois/counter mode (GCM) of operation. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT 2004*, volume 3348 of *LNCS*, pages 343–355. Springer, Heidelberg, December 2004.

[MX23] Varun Maram and Keita Xagawa. Post-quantum anonymity of Kyber. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 3–35. Springer, Heidelberg, May 2023.

[NSV$^+$15] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (mctls): Enabling secure in-network functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015*, pages 199–212. ACM, 2015.

[PM22] Ripon Patgiri and Naresh Babu Muppalaneni. Stealth: A highly secured end-to-end symmetric communication protocol. In *International Symposium on Networks, Computers and Communications, ISNCC 2022, Shenzhen, China, July 19-22, 2022*, pages 1–8. IEEE, 2022.

[PPY22] Giuseppe Persiano, Duong Hieu Phan, and Moti Yung. Anamorphic encryption: Private communication against a dictator. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 34–63. Springer, Heidelberg, May / June 2022.

[Pro14] Gordon Procter. A security analysis of the composition of ChaCha20 and Poly1305. Cryptology ePrint Archive, Report 2014/613, 2014. `https://eprint.iacr.org/2014/613`.

[PST20] Christian Paquin, Douglas Stebila, and Goutam Tamvada. Benchmarking post-quantum cryptography in TLS. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 72–91. Springer, Heidelberg, 2020.

[Raf19] Khan Farhan Rafat. *A Stealth Key Exchange Protocol*, pages 675–695. 07 2019.

[Res18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 98–107. ACM Press, November 2002.

[SFG23] Douglas Stebila, Scott Fluhrer, and Shay Gueron. Hybrid key exchange in TLS 1.3. Internet-Draft draft-ietf-tls-hybrid-design-08, Internet Engineering Task Force, August 2023. Work in Progress.

[Shr04] Tom Shrimpton. A characterization of authenticated-encryption as a form of chosen-ciphertext security. Cryptology ePrint Archive, Report 2004/272, 2004. `https://eprint.iacr.org/2004/272`.

[Sim83] Gustavus J. Simmons. The prisoners' problem and the subliminal channel. In David Chaum, editor, *CRYPTO'83*, pages 51–67. Plenum Press, New York, USA, 1983.

[SLPR15] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015*, pages 213–226. ACM, 2015.

[Tib14] Mehdi Tibouchi. Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 139–156. Springer, Heidelberg, March 2014.

[vH04] Luis von Ahn and Nicholas J. Hopper. Public-key steganography. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 323–341. Springer, Heidelberg, May 2004.

[WC21] Xiuhua Wang and Sherman S. M. Chow. Cross-domain access control encryption: Arbitrary-policy, constant-size, efficient. In *2021 IEEE Symposium on Security and Privacy*, pages 748–761. IEEE Computer Society Press, May 2021.

[WSH14] Eric Wustrow, Colleen Swanson, and J. Alex Halderman. TapDance: End-to-middle anti-censorship without flow blocking. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 159–174. USENIX Association, August 2014.

[WWGH11] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *USENIX Security 2011*. USENIX Association, August 2011.

[WWY$^+$12] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: a camouflage proxy for the Tor anonymity system. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 109–120. ACM Press, October 2012.

A Integration into the TLS 1.3 Record Protocol

In this section we describe how one can use the stealth key exchange to derive a sanitizable channel. The full description of the construction of the sanitizable channel and the security proofs can be found in Section 6. In this overview we only describe a sanitizable version of the TLS 1.3 record layer in which the sanitizer has partly access to designated parts of the record protocol data.

Key Establishment. We assume that the sender and the receiver have executed the TLS 1.3 key exchange protocol. The two parties have used the stealth mode to generate a stealth key stkey in addition to the session key chkey. This is done in such a way that the sanitizer also knows this key chkey (but the sanitizer remains oblivious about the stealth key). One option was to let the receiver securely pass the session key to the sanitizer upon establishment, albeit this appears to be very inconvenient in the firewall setting. An alternative is to let the sanitizer provide the ephemeral secret of the receiver in the key exchange step, being able to compute chkey from the transcript of communication. This requires the sanitizer to either communicate with the receiver while the key exchange protocol runs, or by sharing a local key with the receiver from which the ephemeral secret is derived. Alternatively, the receiver may re-use a sanitizer-provided ephemeral secret in multiple executions. In fact, this corresponds to the static Diffie-Hellman share solution for TLS 1.3 [GDH+17]. The disadvantage in the latter case is that this solution infringes with forward secrecy (yet, forward secrecy in the stealth part of the connection is still preserved).

TLS 1.3 Record Protocol. We note that the key in TLS 1.3 consists of the actual encryption and decryption key and a random offset, called `client_write_iv` resp. `server_write_iv` in TLS, depending on the direction of communication. In this sense it is understood that our derived keys chkey and stkey both contain such a random offset.

The key and the offset are used to encrypt the payload message m in the TLS 1.3 record protocol via a scheme for authenticated encryption with associated data (AEAD) [Rog02] as $c \leftarrow \mathsf{AEEnc}(\mathsf{key}, \mathsf{offset} \oplus \mathsf{st}_S, \mathsf{AD}, m)$. Here, $\mathsf{offset} \oplus \mathsf{st}_S$ is used as a nonce for the AEAD scheme and st_S is a counter (the state of the sender), incremented with each sent ciphertext. The associated data in TLS 1.3 are given by the constant `ContentType opaque_type = application_data` (which equals 23), followed by the constant `ProtocolVersion legacy_record_version = 0x0303`, followed by the (expected) length of the ciphertext in bytes. The latter can be derived from the length of the input message m for the suggested AEAD schemes. To decrypt the receiver calls $m \leftarrow \mathsf{AEDec}(\mathsf{key}, \mathsf{offset} \oplus \mathsf{st}_R, \mathsf{AD}, m)$ where st_R is the current counter value of the receiver (incremented, too, after successful decryption) and AD is given by the constants and ciphertext length as for encryption.

Partly Accessible Channel. We can now proceed as follows to build the stealth channel. Recall that sender, receiver, and sanitizer all share the session key chkey (including the offset $\mathsf{choffset}$), but only sender and receiver know the stealth key stkey (with its own offset $\mathsf{stoffset}$). We assume that we have a message part m_{sec} which should be sent confidentially between sender and receiver, and a part m_{plain} which should only be accessible by the sanitizer (but not to outsiders). We now use a nested encryption, encrypting the m_{sec}-part under the stealth key and then the derived ciphertext together with m_{plain} under the channel key:

$$\begin{aligned} c_{\mathsf{sec}} &\leftarrow \mathsf{AEEnc}(\mathsf{stkey}, \mathsf{stoffset} \oplus \mathsf{st}_S, \mathsf{AD}, m_{\mathsf{sec}}) \\ c &\leftarrow \mathsf{AEEnc}(\mathsf{chkey}, \mathsf{choffset} \oplus \mathsf{st}_S, \mathsf{AD}', (c_{\mathsf{sec}}, m_{\mathsf{plain}})) \end{aligned}$$

Here AD and AD' are the corresponding associated data.

We note that, with this construction, the sanitizer may alter the m_{plain}-part. If we want to give read-only access to the message part, then we put m_{plain} into the associated data $(\mathsf{AD}, m_{\mathsf{plain}})$ in the inner encryption. Since the associated data are authenticated via the stealth key stkey, the sanitizer cannot modify m_{plain} without the receiver detecting modifications of m_{plain}. In fact, this means we rather put the accessible part in m_{auth} and leave m_{plain} empty, according to the terminology of message parts in sanitizable channels in Appendix 6. From now on we will hence use the term m_{auth} for the read-only accessible part.

The sender then transmits c. The receiver and the sanitizer can individually recover $(c_{\mathsf{sec}}, m_{\mathsf{auth}})$ with the help of chkey. The sanitizer can check the information in m_{auth}, but only the receiver is able to also recover the message m_{sec} from c_{sec} with the help of stkey. If m_{auth} is part of the associated data in c_{sec}, then the receiver can also check its integrity. We note that outsiders, which do not know chkey, cannot access either of the two parts.

There are two things to consider regarding the length of the nested ciphertext c. First note that, compared to subliminal communication, an outsider can observe that ciphertexts in this version are longer than when using the original record protocol. As explained in the introducton, we do not aim to hide this fact. Secondly, TLS 1.3 sets an upper bound of $2^{14} + 256$ bytes for the length of ciphertexts, requiring that input messages are of at most 2^{14} bytes (or else need to be fragmented) [Res18]. This needs to be taken into account with the ciphertext expansion due to the double encryption here. Indeed, we need to make sure that the combined length of $(c_{\mathsf{sec}}, m_{\mathsf{auth}})$ is at most 2^{14} bytes, resulting in an overall bound of $2^{14} - 256$ for m_{sec} and m_{auth} and possibly further fragmentations. Let us stress once more that our goal is not to hide the fact that we are using the stealth channel. If this is obeyed, then c is a perfectly legitimate TLS 1.3 record protocol ciphertext which supports read-only access for the sanitizer.