Proxying is Enough: Security of Proxying in TLS Oracles and AEAD Context Unforgeability

Zhongtang Luo
Purdue University
luo401@purdue.edu

Yanxue Jia
Purdue University
jia168@purdue.edu

Yaobin Shen
Xiamen University
yaobin.shen@xmu.edu.cn

Aniket Kate
Purdue University / Supra Research
aniket@purdue.edu

***Abstract*—TLS oracles allow a TLS client to offer selective data provenance to an external (oracle) node such that the oracle node is ensured that the data is indeed coming from a pre-defined TLS server. Typically, the client/user supplies their credentials to the server and reveals selective data using zero-knowledge proofs to demonstrate certain server-offered information to oracles while ensuring the secrecy of the rest of the TLS transcript. Conceptually, this is a standard three-party secure computation between the TLS server, TLS client (prover), and the oracle (verifier) node; however, the key practical requirement for TLS oracles to ensure that data provenance process remains transparent to the TLS server. Recent TLS oracle protocols such as DECO enforce the communication pattern of server-client-verifier and utilize a novel three-party handshake process during TLS to ensure data integrity against potential tempering by the client. However, this approach introduces a significant performance penalty on the client/prover and the verifier. This raises the question of whether it is possible to reduce the overhead by putting the verifier (as a proxy) between the server and the client such that the correct TLS transcript is available to the verifier.**

This work offers both positive and negative answers to this oracle proxy question: We first formalize the oracle proxy notion that allows the verifier to directly proxy client-server TLS communication, without entering a three-party handshake or interfering with the connection in any way. We then show that for common TLS-based higher-level protocols such as HTTPS, data integrity to the verifier proxy is ensured by the variable padding built into the HTTP protocol semantics. On the other hand, if a TLS-based protocol comes without variable padding, we demonstrate that data integrity cannot be guaranteed. In this context, we then study the case where the TLS response is pre-determined and cannot be tampered with during the connection. We propose the concept of context unforgeability and show allows overcoming the impossibility. We further show that ChaCha20-Poly1305 satisfies the concept while AES-GCM does not under the standard model.

1. Introduction

Transport Layer Security (TLS) [1], [2] has been widely used to establish a secure communication channel between a client and a server, enabling secure data access (we refer to such data as TLS-protected data). However, TLS relies on symmetric encryption and both the client and the server can obtain the key. This enables the client to build a ciphertext for any data and falsely claim that it comes from the server. Thus, TLS does not allow the client to prove the *provenance* and *integrity* of the data to third parties, limiting the propagation of data. In particular, many decentralized applications on the blockchain need to access the TLS-protected data as third parties. Due to the TLS limitation, we cannot rely on each client to feed data into the blockchain. Therefore, a type of service called *Oracle* [3], [4] was proposed to feed TLS-protected data into blockchains while guaranteeing the provenance and integrity of data.

A straightforward way to implement the service is to allow a verifier to act as the client to access data from the server. Unfortunately, this approach compromises the client's privacy, as the verifier can obtain additional private information beyond the data. Therefore, a desirable way is to allow the verifier to participate in the communication between the client and the server to prevent the client from modifying the data, but *without* obtaining additional private information. In addition, some applications may also not necessarily require the complete data but only need to verify if the data meets a predetermined predicate. In such cases, to maximize privacy protection, the client can use zero-knowledge proofs to prove that the data satisfies the predicate without disclosing the data to the verifier.

In Figure 1, we give a use case to explain how the oracle works: There is an application on the blockchain that activates a rental contract when 'Alice's balance in bank A exceeds \$50.' Alice uses her password to request the balance \$100 from bank A, through a TLS connection. The verifier participates in the TLS communication without learning request and response messages but can check if Alice's balance in bank A exceeds \$50. If so, the verifier submits the confirmation information to the blockchain, and then the rental contract is activated.

As bringing clients' private data securely and selectively to blockchains is a problem of immense interest, a few academic [5], [6], [7], [8], [9], [10] and industrial [11], [12] efforts in this direction are already available. Earlier works typically involve modifying the TLS server-side code [7] or utilizing trusted hardware [6]. While the approaches remain an interesting theoretical probability, typical servers

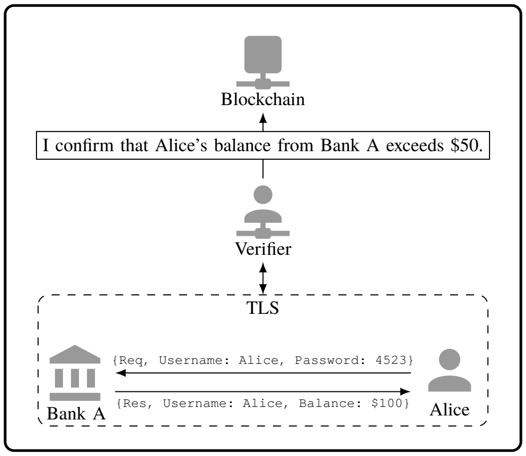

Blockchain
I confirm that Alice's balance from Bank A exceeds $50.
Verifier
TLS
{Req, Username: Alice, Password: 4523}
{Res, Username: Alice, Balance: $100}
Bank A
Alice

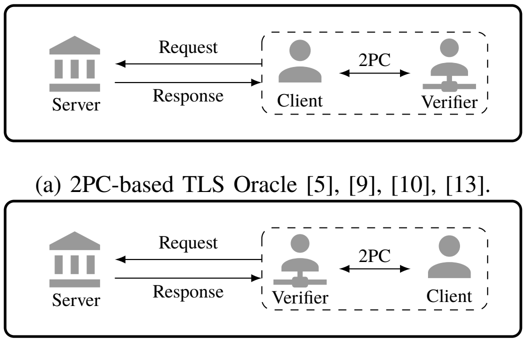
Figure 1: A use-case example. Alice queries Bank A about her balance via TLS. The verifier/oracle attests to the blockchain about Alice's balance in bank A, by intervening in the TLS communication.

are reluctant to use modified TLS, and the usage of trusted hardware introduces additional trusted entities. Hence both approaches see limited use.

(Proxy-but-)2PC-based TLS Oracle. In a seminal work, Zhang et al. [5] proposed DECO, which overcomes this practicality barrier without making any changes to the TLS server-side code and using any trusted hardware. The core reason why the client in TLS can forge the TLS data is that the symmetric encryption and authentication key generated in the TLS session is shared by the client and the server (please see Section 2 for more details). DECO [5], essentially splits the role of the client in typical TLS into two parties, as shown in Figure 2a, so that the client cannot learn the entire symmetric encryption and authentication key. In particular, the client and the verifier collaboratively communicate with the server using secure two-party computation (2PC). Moreover, while the client and the verifier collaborate, it is only the client that communicates with the server as in TLS. We call this design as '2PC-based TLS oracle' protocol. Later, the subsequent works [9], [10], [13] also follow the collaboration model.

In addition, Zhang et al. [5] also discussed another design framework, where the verifier acts as a proxy to communicate with the sender but the verifier still needs to perform 2PC with the client, as shown in Figure 2b. We call this design as 'proxy-but-2PC based TLS oracle' protocol. Lauinger et al. [8] proposed a solution in this model that utilizes a garble-then-prove scheme to improve efficiency.

We can see that the previous works all rely on heavy 2PC between the client and the verifier. A natural question is *'Is there a secure TLS oracle solution that can totally avoid heavy 2PC between the client and the verifier?'*



(a) 2PC-based TLS Oracle [5], [9], [10], [13].

(b) Proxy-but-2PC based TLS Oracle [5], [8].

Figure 2: Design frameworks in the previous works. No matter who sends the ciphertext to the server, a 2PC protocol is used to prevent the client from forging a message.

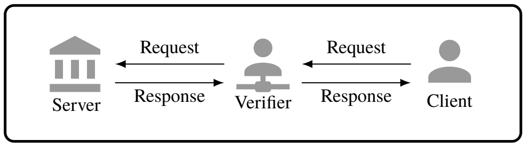


Figure 3: Proxy-based TLS Oracle (this work). The client and the server interact with each other as in TLS, but the messages between them are forwarded by the verifier.

Proxy-based TLS Oracle. To answer this question, in this work, we consider a pure proxy-based TLS oracle as shown in Figure 3, where the verifier only needs to forward the messages generated by the client and the server, without additionally introducing any heavy computation, which directly implies *better performance*. In addition, compared with the (proxy-but-)2PC-based TLS oracles, the proxy-based oracle enables *better compatibility* across different TLS versions, as the verifier only needs to forward messages. Moreover, the proxy-based oracle not only does not modify server-side code but also achieves *client-side non-modification*.

Given that Authenticated Encryption with Associated Data (AEAD) is the core building block of TLS, we investigate the security of two AEAD schemes, AES-GCM and ChaCha20-Poly1305, which are overwhelmingly used in TLS 1.2/1.3 with an adoption rate of over 99% [14]. Based on the security of AEAD, we systematically conduct an investigation that outlines the boundary between security and insecurity of the proxy-based TLS oracle protocol. Surprisingly, we find that the proxy-based TLS oracle can be securely used in two prevalent situations, HTTPS (see Section 6) and prefixed relevant data (see Section 7).

1.1. Contributions

Formalization. In Section 4, we provide an oracle ideal functionality and formalize the notion of the above proxy-based TLS oracle protocol that allows the verifier to directly proxy client-server TLS communication. The definitions allow us to analyze its security in this paper but can also be applied to different versions of TLS and a comparison between oracle protocols. We consider them to be of independent interest.

Impossibility. Based on the above definitions, we also prove that if the underlying AEAD scheme satisfies key commitment, the proxy-based TLS oracle protocol can securely realize the oracle ideal functionality in Section 4. On the other hand, in Section 5, we find that without the key commitment, we can construct a server that allows an adversarial client to forge a message to the verifier.

Therefore, we know that the key commitment is *sufficient* and *necessary* to guarantee the security of the proxy-based TLS oracle protocol. However, the AEAD schemes currently used in TLS do not satisfy the key commitment, according to previous research [15], [16], which means that the proxy-based TLS oracle protocol is not secure without specific constraints.

Possibility. Fortunately, we observe that the real-world application scenarios provide some practical constraints such that the proxy-based TLS oracle can be used securely.

HTTPS. In Section 6, we define a new notion called 'variable padding' in Definition 6.1, and prove that when the plaintext contains a sufficiently long variable padding, AES-GCM and ChaCha20-Poly1305 satisfy the key commitment, which means that the proxy-based TLS oracle is secure in this case. Notably, we show that the TLS-based HTTPS protocol, which is used in over 85% of all web pages [17], contains a sufficiently long variable padding, and thus the proxy-based TLS oracle protocol can be securely used on it.

Prefixed relevant data. In Section 7, we focus on the constraint where the response plaintext is prefixed and not changeable by the client once the connection is established. For example, in practice, the client cannot arbitrarily modify his age, salary, social security number, etc. To analyze the security in this case, we propose a new cryptographic property, *context unforgeability*, and prove that AES-GCM is not secure yet, ChaCha20-Poly1305 is secure with respect to context unforgeability. In addition, we prove that as long as the TLS uses a context-unforgeable AEAD scheme, the proxy-based TLS oracle is secure in this case.

New AEAD Security Property. Besides being used for analyzing the security of the proxy-based TLS oracle, the newly proposed context unforgeability precisely bridges the gap in the security analysis of AEAD, and thus may be of independent interest.

2. Technical Overview

In practice, a client $\mathcal{P}$ can retrieve a message m from a server $\mathcal{W}$ via TLS. An Oracle protocol aims to allow the client $\mathcal{P}$ to prove to a verifier $\mathcal{V}$ that the message m is indeed obtained from the server $\mathcal{W}$ and satisfies a predetermined requirement (e.g., age is greater than 18). In this work, we assume that server $\mathcal{W}$ is always honest, while client $\mathcal{P}$ and verifier $\mathcal{V}$ can be malicious. The key idea of our work is to use verifier $\mathcal{V}$ as a proxy to prevent client $\mathcal{P}$ from tampering with the message m. Therefore, we also assume that verifier $\mathcal{V}$ reliably connects to server $\mathcal{W}$, i.e., verifier $\mathcal{V}$ can ensure that she indeed connects with server $\mathcal{W}$ and the communication messages cannot be tampered with by others (including client $\mathcal{P}$). The assumption has been accepted by other proxy-setting systems [18]. Based on the above assumptions, we design a proxy-based oracle protocol as illustrated in Figure 4 and obtain a series of theoretical results summarized in Figure 5. Next, we will detail how we obtain these results.

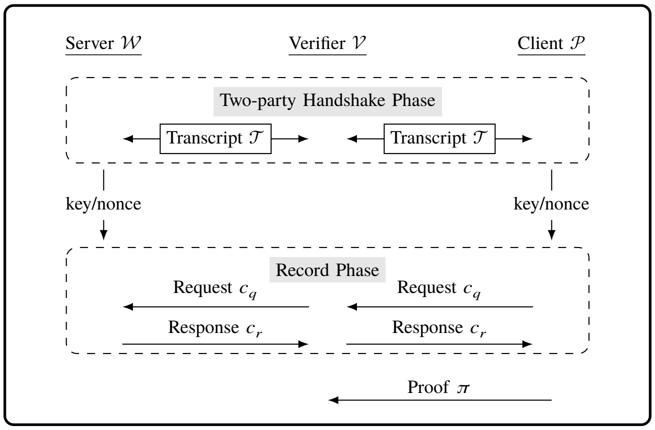


Figure 4: Our Proxy-based Oracle Protocol. The server and the client perform as in TLS, but the messages are forwarded by the verifier. Finally, the client sends a proof π to the verifier to convince her that there is a message m that is decrypted from c_r and satisfies a predetermined requirement. The detailed specification is defined in Π_{Proxy} (see Figure 9).

TLS relies on authenticated encryption with associated data (AEAD), a symmetric-key primitive, to achieve integrity and confidentiality of exchanged messages. As shown in Figure 4 (ignoring the verifier and the proof π), TLS consists of a two-party handshake phase and a record phase. After the two-party handshake phase, the server $\mathcal{W}$ and the client $\mathcal{P}$ both obtain a key/nonce pair [1] of AEAD to encrypt and decrypt the server's messages. Then, in the record phase, the server $\mathcal{W}$ and the client $\mathcal{P}$ communicate with each other using the key/nonce pair generated by the handshake phase. We can see that the client $\mathcal{P}$ also holds the key/nonce pair of AEAD. Therefore, the client $\mathcal{P}$ can use the key/nonce pair to encrypt another distinct message m' to generate a ciphertext c'_r, and falsely claim that c'_r is retrieved from server $\mathcal{W}$. Note that here we discuss the original TLS without the verifier as a proxy.

To address the problem, previous works [5], [8], [9], [10], [13] prevent client $\mathcal{P}$ from obtaining the whole key by splitting the key into two parts, each of which is obtained by client $\mathcal{P}$ and verifier $\mathcal{V}$ respectively. Specifically, client $\mathcal{P}$ and verifier $\mathcal{V}$ collectively fulfill the role of the client in TLS through a secure two-party computation (2PC) protocol. However, compared with the original TLS protocol, introducing a 2PC protocol incurs significant extra costs, which raises a question: *Is it possible to avoid 2PC protocols?*

[1] Here, we ignore the other key/nonce pair for encrypting and decrypting the client's messages, as this work mainly focuses on preventing the client from forging the server's messages.

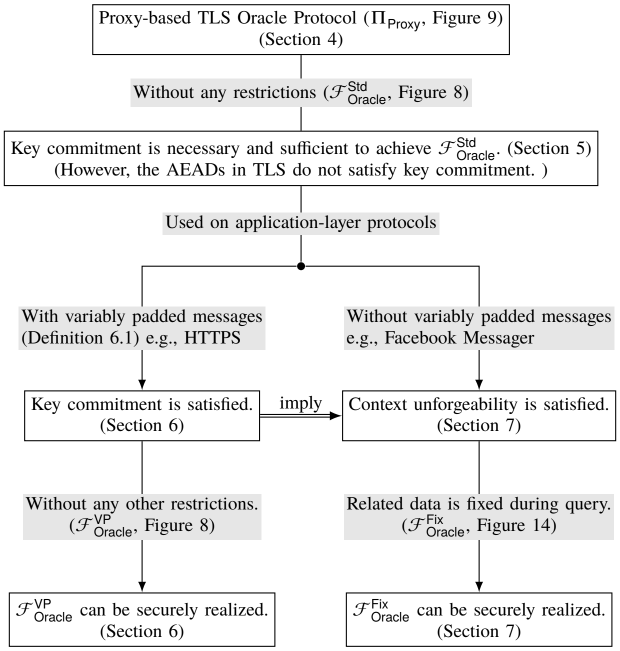

Proxy-based TLS Oracle Protocol (Π_{Proxy}, Figure 9)
(Section 4)
Without any restrictions ($\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, Figure 8)
Key commitment is necessary and sufficient to achieve $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$. (Section 5)
(However, the AEADs in TLS do not satisfy key commitment.)
Used on application-layer protocols
With variably padded messages
(Definition 6.1) e.g., HTTPS
Without variably padded messages
e.g., Facebook Messager
Key commitment is satisfied.
(Section 6)
imply
Context unforgeability is satisfied.
(Section 7)
Without any other restrictions.
($\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$, Figure 8)
Related data is fixed during query.
($\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$, Figure 14)
$\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$ can be securely realized.
(Section 6)
$\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ can be securely realized.
(Section 7)

Figure 5: Main contributions. We first give a proxy-based TLS oracle protocol Π_{Proxy} and a functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ without any restrictions. Then, we observe that key commitment is necessary and sufficient to achieve $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$; however, the AEADs used in TLS are not key committing in general. Fortunately, real-world applications come with different restrictions on the plaintext response: 1) Plaintexts are variably padded (e.g., HTTPS), and we denote the functionality as $\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$. 2) Relevant data is immutable during the query (e.g., age), and we denote the functionality as $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$. When applying the AEADs in TLS to variably padded messages when the padding is known by the verifier in advance, we show that key commitment is satisfied, and thus $\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$ can be achieved. In addition, we prove that regardless of whether the plaintexts are variably padded or not, ChaCha20-Poly1305 in TLS satisfy a newly proposed property, *context unforgeability*, which is sufficient to achieve $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$.

Preliminary Attempt. We observe that in the attack above, client $\mathcal{P}$ uses the AEAD key/nonce pair to generate a new ciphertext $c_r' \neq c_r$, where c_r is the original ciphertext from server $\mathcal{W}$. The essence of previous works [5], [8], [9], [10], [13] is to ensure that client $\mathcal{P}$ cannot obtain the complete AEAD key before he commits the message. Our preliminary attempt is to ensure that verifier $\mathcal{V}$ can directly obtain the ciphertext c_r from server $\mathcal{W}$, such that verifier $\mathcal{V}$ can recognize any modifications on the ciphertext c_r. To this end, we treat verifier $\mathcal{V}$ as a proxy that forwards and records the messages between server $\mathcal{W}$ and client $\mathcal{P}$, as shown in Figure 4. In this way, the ciphertext c_r cannot be modified by client $\mathcal{P}$. Moreover, verifier $\mathcal{V}$ records the transcript $\mathcal{T}$ generated during the handshake phase. Therefore, client $\mathcal{P}$ can prove the following three statements:

1. He holds a key/nonce pair (k, n) that can decrypt c_r to a message m;
2. The message m satisfies a preset requirement;
3. The key/nonce pair (k, n) is derived from the transcript $\mathcal{T}$.

If the message m does not involve privacy, client $\mathcal{P}$ can directly open the key/nonce pair (k, n) and the message m to verifier $\mathcal{V}$. However, if client $\mathcal{P}$ intends to hide the message m, he needs to leverage a zero-knowledge proof scheme to complete the proof, which still also results in high costs.

Obviously, proving the first two statements is *necessary*. However, an oracle task itself does not include proving the third statement. Moreover, the hash function used in TLS 1.2/1.3 to derive key/nonce is SHA256, which is not SNARK (Succinct Non-interactive Argument Knowledge)-friendly [19]. Therefore, to further reduce extra overhead, we delve deeper into the potential of *eliminating the proof that the key/nonce pair is derived from the transcript $\mathcal{T}$ (i.e., the above statement 3).*

Eliminating the Proof of Key Origin. As shown by the previous research [15], [16], the AEAD schemes used by TLS do not satisfy the key commitment property (see Lemma 3.1); an adversary can efficiently construct a ciphertext c and two distinct key/nonce pairs (k_1, n_1) and (k_2, n_2), such that c can be decrypted to two distinct messages m_1 and m_2 by using (k_1, n_1) and (k_2, n_2) respectively. At first glance, in our proxy-based oracle protocol, client $\mathcal{P}$ cannot launch the above attack, since client $\mathcal{P}$ seems unable to construct the ciphertext sent by server $\mathcal{W}$. However, we observe this is not true.

More specifically, client $\mathcal{P}$ can obtain the key/nonce pair (k, n) once completing the handshake phase. Then, client $\mathcal{P}$ can construct a ciphertext c such that it can be decrypted into m and m' under key/nonce pair (k, n) and another key/nonce pair (k', n') respectively. Moreover, m' meets the predetermined requirement, while m does not. Subsequently, the client can change the record in server $\mathcal{W}$ to m through external interactions, so that server $\mathcal{W}$ produces and sends the ciphertext c. Finally, client $\mathcal{P}$ can use (k', n') to convince verifier $\mathcal{V}$ that m' satisfying the preset requirement is the corresponding plaintext. For example, if the message is about bank balance, client $\mathcal{P}$ can adjust his balance to $m = 10$, and then prove that balance $m' = 1000$ by using (k', n'). Please see Section 5 for more details.

With Variable Padding. According to the above analysis, we know that key commitment is necessary for a secure proxy-based oracle protocol without proving that the key is derived from a given transcript. Fortunately, we observe that if the plaintext is well-formatted as seen in HTTPS (or more formally, has a variable padding), the adversary cannot break the key commitment. Intuitively, even if the adversary can generate (c, k, n, m) and (c, k', n', m'), the probability that m and m' are both well-formatted is negligible (please see Section 6 for more details).

In practice, HTTPS (using TLS on the application layer protocol HTTP) satisfies the above condition, since HTTP headers are variably padded [20]. Therefore, our proxy-based oracle protocol without *proving the key origin* and

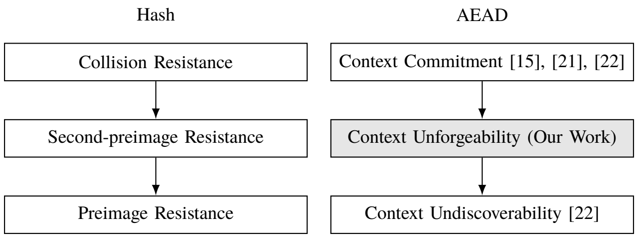

Hash
AEAD
Collision Resistance
Context Commitment [15], [21], [22]
Second-preimage Resistance
Context Unforgeability (Our Work)
Preimage Resistance
Context Undiscoverability [22]

Figure 6: Hierarchical Security of AEAD. Both hash and AEAD can be abstracted as a map: $x \rightarrow y$; in AEAD, x includes key k, nonce n, and associated data A, and y refers to ciphertext c. The first level (collision resistance and context commitment) refers to the fact that an adversary cannot efficiently compute y and two different x and x' such that x and x' map to y. The second level (second-preimage resistance and context unforgeability) refers to the fact that given x and y where $x \rightarrow y$, an adversary cannot efficiently compute $x' \neq x$ such that x' also maps to y. The third level (preimage resistance and context undiscoverability) refers to that given y, an adversary cannot efficiently compute x such that x maps to y.

2PC protocols is secure with HTTPS. While HTTPS is one of the most popular protocols and thus the current results are already sufficient to address a wide range of application scenarios, we also discuss if our proxy-based oracle protocol can be securely used for other protocols whose messages are not variably padded.

Without Variable Padding. In the above attack, we assume that client $\mathcal{P}$ can arbitrarily modify the record in server $\mathcal{W}$. However, in some scenarios (e.g., ages, stock prices and insurance numbers), client $\mathcal{P}$ is not able to perform modifications. In these cases, the AEAD used in TLS only needs to satisfy a weaker security property: given a ciphertext c, a key/nonce pair (k, n) and the corresponding message m, the adversary cannot construct another key/nonce pair (k', n') such that c can be decrypted to another message m'. In Section 7, we formally define this security property as *context unforgeability (CFY-security)* and systematically investigate if the AEADs (including AES-GCM and ChaCha20-Poly1305) used in TLS satisfy this new security property. Specifically, we rigorously prove that AES-GCM does not satisfy the context unforgeability, whereas ChaCha20-Poly1305 does.

Hierarchical Security of AEAD. Besides designing an efficient proxy-based oracle protocol, our newly proposed security property, context unforgeability, bridges the gap in the security analysis of AEAD, as shown in Figure 6. Interestingly, combined with the previous properties, context commitment [15], [21], [22] [2] and context undiscoverability [22], the hierarchical security of AEAD precisely corresponds to the hierarchical security of hash functions.

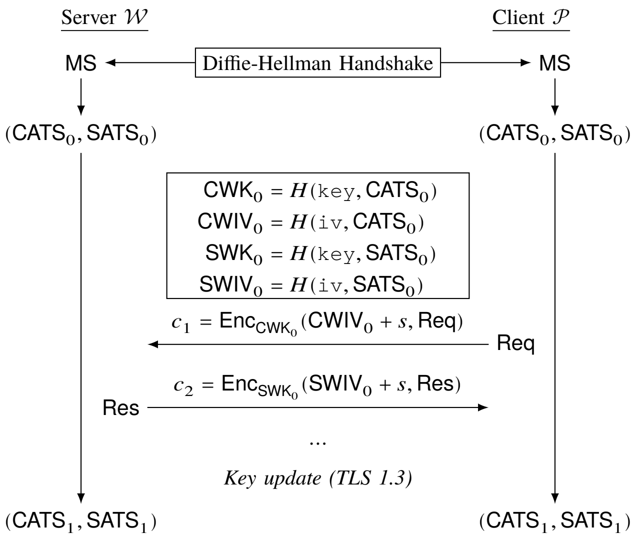

Server $\mathcal{W}$
Client $\mathcal{P}$
MS
Diffie-Hellman Handshake
MS
$(\mathsf{CATS}_0, \mathsf{SATS}_0)$
$(\mathsf{CATS}_0, \mathsf{SATS}_0)$
$\mathsf{CWK}_0 = H(\mathtt{key}, \mathsf{CATS}_0)$
$\mathsf{CWIV}_0 = H(\mathtt{iv}, \mathsf{CATS}_0)$
$\mathsf{SWK}_0 = H(\mathtt{key}, \mathsf{SATS}_0)$
$\mathsf{SWIV}_0 = H(\mathtt{iv}, \mathsf{SATS}_0)$
$c_1 = \mathsf{Enc}_{\mathsf{CWK}_0}(\mathsf{CWIV}_0 + s, \mathsf{Req})$
Req
$c_2 = \mathsf{Enc}_{\mathsf{SWK}_0}(\mathsf{SWIV}_0 + s, \mathsf{Res})$
Res
...
Key update (TLS 1.3)
$(\mathsf{CATS}_1, \mathsf{SATS}_1)$
$(\mathsf{CATS}_1, \mathsf{SATS}_1)$

Figure 7: An overview of a TLS session between a client and a server. After establishing a master secret MS with a Diffie-Hellman handshake in the handshake protocol, the parties derive the 0-th client/server application secret $(\mathsf{CATS}_0, \mathsf{SATS}_0)$. They then derive the client write key CWK, the client write IV [3]CWIV, the server write key SWK, and the server write IV SWIV. The values are used in the communication, together with the sequence number s to ensure the uniqueness of each IV. Additionally, in TLS 1.3, each party can also trigger a key update event that refreshes the application secret.

3. Preliminary

We briefly outline the relevant background knowledge on TLS, the definition and security properties of Authenticated Encryption with Associated Data (AEAD), and Non-interactive Zero-Knowledge Proof (NIZK).

Notations. We use $|m|$ to denote the bit-length of a string m, and $m[i : j]$ to represent its substring starting at i (0-based) with length $(j - i)$, and $m||n$ to denote the concatenation of strings m and n. We represent the set size of a set S as $|S|$.

3.1. TLS

Transport Layer Security (TLS) is a family of communication protocols designed to provide end-to-end security over a computer network. Its most prominent use remains to be HTTPS, the web-browsing protocol that sees day-to-day use. TLS 1.3 is the latest protocol in the TLS family, defined in August 2018 [1].

There are two main protocols in TLS. The handshake protocol negotiates a symmetric key to be used in the record protocol. The record protocol manages the transmission of messages, using an authenticated encryption with associated data (AEAD) cipher suite to ensure confidentiality and integrity. An overview of TLS is available in Figure 7.

[2] The works [15], [21] first proposed the key commitment problem. Then Bellare et al. [23] extended to committing nonce and associated data, and Menda et al. [22] summarized them as context commitment.

Authenticated Encryption with Associated Data. Nonce-based authenticated encryption with associated data (AEAD) schemes [24] are employed in TLS to ensure data confidentiality and integrity. An AEAD scheme consists of the following four algorithms:

- $\mathsf{AEAD.Setup}(1^\lambda)$: takes a security parameter λ, and outputs a public parameter pp indicating the key space $\mathcal{K}$, the nonce space $\mathcal{N}$, the associated-data space $\mathcal{AD}$, the plaintext space $\mathcal{M}$ and the ciphertext space $\mathcal{C}$;
- $\mathsf{AEAD.Gen}(pp)$: takes a public parameter pp, and outputs a secret key k sampled from $\mathcal{K}$;
- $\mathsf{AEAD.Enc}_k(n, a, m)$: takes a secret key k, a nonce $n \in \mathcal{N}$, an associated data $a \in \mathcal{AD}$ and a message $m \in \mathcal{M}$, and outputs a ciphertext $c \in \mathcal{C}$;
- $\mathsf{AEAD.Dec}_k(n, a, c)$: takes a secret key k, a nonce $n \in \mathcal{N}$, an associated data $a \in \mathcal{AD}$ and a ciphertext $c \in \mathcal{C}$, and outputs a message $m \in \mathcal{M}$ or $\perp$ to indicate an invalid ciphertext.

Cipher Suites in TLS. TLS supports a handful number of cipher suites [1], with AES-GCM and ChaCha20-Poly1305 being the most popular and enabled in OpenSSL by default [25]. Notably, all cipher suites in TLS adopt a block-based construction — the plaintext and the associated data are packed into a series of fixed-size blocks $(p_1, p_2, \dots, p_n)$ and $(a_1, a_2, \dots, a_m)$, which is then processed by the algorithm to produce the ciphertext.

AES-GCM. AES-GCM is the most widely used cipher suite in TLS and the only cipher suite that must be implemented by every application under the specification. In AES-GCM, all the computation is done over the field $GF(2^{128})$. The ciphertext is obtained by

$$c_i = p_i + E_k(n + i), \tag{1}$$

where E is the AES block cipher.

AES-GCM also ensures authenticity by using an authentication tag. The tag is computed by

$$t = E_k(n) + \sum_{i=1}^{m+n+1} s_i E_k(0)^{m+n+2-i}. \tag{2}$$

where $s = (a, c, m\|n)$ is a concatenation of a, c and their length. For further information, we refer the reader to the specification for clarity [26].

ChaCha20-Poly1305. ChaCha20-Poly1305 is an alternative to AES-GCM. In ChaCha20-Poly1305, encryption is done similarly to AES-GCM:

$$c_i = p_i + H_k(n + i), \tag{3}$$

where H is the ChaCha20 stream cipher.

However, the authentication tag computation in ChaCha20 is different from AES-GCM due to the usage of Poly1305. First, two 128-bit variables $(r, s) = H_k(n)[0 : 256]$ are sampled from the stream cipher. Then, the authentication tag is computed by

$$t = s + \sum_{i=1}^{m+n+1} s_i r^{m+n+2-i}, \tag{4}$$

where $s = (a, c, m\|n\|0^8)$ is the concatenation as in AES-GCM. The computation of t is done over $GF(2^{130} - 5)$, then truncated to 128 bits. For details, we refer the reader to the RFC specification [27].

Key Update. TLS 1.3, the latest TLS version, also supports an operation known as a key update. The operation allows any party to refresh the secret (i.e., CATS_i and/or SATS_i in Figure 7) used on either or both sides. The new secret is generated based on a hash of the old secret, and the new keys and IVs are derived from the new secrets based on the same rule as shown in Figure 7.

3.2. Context Attacks of AEAD

In the proxying oracle protocol, we need to discuss the possibility that an adversarial prover deceives the verifier by providing a symmetric key that does not correspond to the one in the handshake yet still decrypts the ciphertext. This corresponds to the concept of context discovery and commitment attacks, first summarized by Menda et al. [22]. Here we outline a couple of specific definitions that will be useful in our cases. We use † to denote a specification of the more general case discussed by Menda et al.

Context Discovery Attack. A context discovery (CDY) attack refers to the adversary's capability to come up with some part of the context (i.e. a key and/or a nonce) that decrypts the given ciphertext without error, although not necessarily to the original plaintext. Formally:

Definition 3.1 (Context Discovery)**.** *Fix some AEAD parameter pp and a corresponding AEAD oracle* Π. *The game* $\mathsf{CDY}^\dagger$ *is defined as:*

1) *The challenger samples a random ciphertext c from some ciphertext space and its corresponding decryption context* (k, n, a).
2) *The challenger sends* (c, a) *to some adversary* $\mathcal{A}$.
3) *The adversary wins if it outputs a valid context* (k', n', a) *that decrypts c successfully.*

The adversary's q-advantage $\Delta^{\mathcal{A}}_{\mathsf{CDY}^\dagger}$ *is defined as the probability it wins under q queries to the AEAD oracle* Π.

Context Commitment Attack. A context commitment (CMT) attack refers to the adversary's capability to come up with some context (e.g. ciphertext) and provide two interpretations of it. Formally, the game is defined as:

Definition 3.2 (Context Commitment)**.** *Fix some AEAD parameter pp and a corresponding AEAD oracle* Π. *The game* $\mathsf{CMT}^\dagger$ *is defined as:*

1) *The challenger samples and sends* (k, n) *to some adversary* $\mathcal{A}$.

[3] We note that there is a difference between the IV in the TLS protocol and the nonce in the AEAD scheme. In both AES-GCM and ChaCha20-Poly1305 of the RFC specification [24], IV is limited to 96 bits, while the nonce used in the encryption is derived from IV by padding and can be longer.

2) The adversary wins if it outputs a ciphertext c and two valid contexts (k,n,a) and (k',n',a) with $(k,n) \neq (k',n')$ that decrypts c successfully.

The adversary's q-advantage $\Delta^{\mathcal{A}}_{\mathsf{CMT}^{\dagger}}$ is defined as the probability it wins under q queries to the AEAD oracle Π.

Rationale for the Definition. As Menda et al. [22] have pointed out, there exist many variations of commitment attacks with different limitations on the key, nonce and associated data. Here we observe that in TLS the verifier cannot access the key and the nonce but can access the associated data. Therefore, the adversary (i.e., client $\mathcal{P}$) can only manipulate the key and the nonce. Based on the fact, we pick the above definition that matches our scenario. Nevertheless, we observe that our definition of CDY and CMT is a specification of the granular security framework proposed by Menda et al., and we defer the discussion of the general abstraction to Appendix A.

Relationship between CMT and CDY. During the discussion of the relationship between CMT and CDY, Menda et al. proved that CMT security implies CDY security, assuming that there is a non-negligible probability that a ciphertext can be decrypted under two different contexts (known as context compression). They also make an analogy that CDY is to CMT as a preimage attack is to a collision attack on a hash function. For a full discussion, we refer the reader to their paper for details.

The required security for AEAD in proxying is a little more lenient than CDY but a little more strict than CMT. We will define an in-between game, the context forgery (CFY) attack, and provide a discussion of the relation between the three games in Section 7.2.

Key Commitment Attacks. Notably, the two major cipher suites in TLS, AES-GCM and ChaCha20-Poly1305, are not key committing under this definition.

Lemma 3.1 (Key Commitment Attacks)**.** *For $\mathsf{AEAD} \in \{E\text{-}GCM, H\text{-}Poly1305\}$, where E is an ideal cipher modeling AES and H is a random function modeling ChaCha20, there exists an adversary that queries the oracle at most q times and wins $\mathsf{CMT}^{\dagger}$ with probability at least $2^{-32}q$.*

The concrete attack is well-studied and presented in multiple works [15], [16], [22], which we invite the readers to explore for more context.

3.3. Non-Interactive Zero-Knowledge Proof

Non-interactive zero-knowledge (NIZK) proof allows a prover to demonstrate that something is true to a verifier without him knowing any additional information. We follow the formal definition of NIZK defined by Groth et al. [28] as Functionality $\mathcal{F}_{\mathsf{NIZK}}$. For a formal definition, we refer the readers to Appendix B.

Functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$

This functionality interacts with a server $\mathcal{W}$, a client $\mathcal{P}$ (namely, prover), a verifier $\mathcal{V}$ and an adversary $\mathcal{S}$. There is a public predicate $\mathsf{P}(\cdot)$.

Functionality:

Handshake phase:

- Upon receiving $\langle \textsc{RequestHS}, sid\rangle$ from client $\mathcal{P}$, send $\langle \textsc{RequestHS}, sid\rangle$ to adversary $\mathcal{S}$ and verifier $\mathcal{V}$;
- Upon receiving OK from adversary $\mathcal{S}$ and verifier $\mathcal{V}$, send $\langle \textsc{RequestHS}, sid\rangle$ to server $\mathcal{W}$;
- Upon receiving $\langle \textsc{ResponseHS}, sid, \mathrm{OK}\rangle$ from server $\mathcal{W}$, send $\langle \textsc{ResponseHS}, sid, \mathrm{OK}\rangle$ to adversary $\mathcal{S}$ and verifier $\mathcal{V}$;
- Upon receiving OK from adversary $\mathcal{S}$ and verifier $\mathcal{V}$, send $\langle \textsc{ResponseHS}, sid, \mathrm{OK}\rangle$ to client $\mathcal{P}$;

Request phase:

- Upon receiving $\langle \textsc{Request}, sid, Q\rangle$ from client $\mathcal{P}$, send $\langle sid, |Q|\rangle$ to adversary $\mathcal{S}$, where $|Q|$ is the bit-length of the query Q;
- Upon receiving OK from adversary $\mathcal{S}$ and verifier $\mathcal{V}$, send $\langle sid, Q\rangle$ to server $\mathcal{W}$;

Response phase:

- Upon receiving $\langle \textsc{Response}, sid, R\rangle$ from server $\mathcal{W}$, send $\langle sid, |R|\rangle$ to adversary $\mathcal{S}$ and verifier $\mathcal{V}$, where $|R|$ is the bit-length of the response R;
- Upon receiving OK from adversary $\mathcal{S}$ and verifier $\mathcal{V}$, send $\langle sid, R\rangle$ to client $\mathcal{P}$;

Prove phase:

- Upon receiving $\langle \textsc{Prove}, sid\rangle$ from client $\mathcal{P}$, send $\langle sid\rangle$ to adversary $\mathcal{S}$;
- Upon receiving OK from adversary $\mathcal{S}$, send $\langle sid, \mathsf{P}(R)\rangle$ to verifier $\mathcal{V}$.

Figure 8: Standard Oracle Functionality. The functionality $\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$ is the same except that the server's response is variable-padding (see Definition 6.1).

4. Proxy-based Oracle Protocol

4.1. Oracle Functionality

We define the oracle functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ in Figure 8. The functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ consists of four phases. In the handshake phase, client $\mathcal{P}$ and server $\mathcal{W}$ will reach an agreement on whether to initiate the connection. At the same time, the connection is only established successfully when the verifier also agrees to establish it, which reflects that the verifier acts as a proxy to relay messages between client $\mathcal{P}$ and server $\mathcal{W}$. This fact also applies to subsequent processes. In the request phase, the functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ receives the query Q from client $\mathcal{P}$ and sends it to server $\mathcal{W}$. In the response phase, server $\mathcal{W}$ obtains the response R from the environment $\mathcal{E}$, which reflects that the related data can be modified. In the prove phase, the functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ sends $\mathsf{P}(R)$ to the verifier $\mathcal{V}$.

Comparison with other works. Comparing our definition of oracle functionality with other works [5], [8], we observe that the definition given by Zhang et al. allows the verifier to assert the request of the client in addition to the response. We consider this functionality redundant, since in most HTTPS applications the server responds with a well-formatted JSON string that allows the verifier to verify the content without the request. For example, all of the 57 examples provided by reclaim, an industrial implementation of the TLS oracle protocol [12], show that responses include self-explanatory JSON strings that contain some form of username/ID so that the verifier does not need the content of the request to assert the validity of the proof.

Nevertheless, we recognize the possibility that especially in home-brew protocols, there may exist a requirement to assert the request of the client. We observe that it is possible to achieve this assertion in TLS 1.3 with key update and include a discussion on various methods in Appendix C.

4.2. Detailed Protocol

In Figure 9, we show the details of our proxy-based oracle protocol Π_{Proxy}, which is parameterized by a predicate $\mathsf{P}(\cdot)$. There are a server $\mathcal{W}$, a client $\mathcal{P}$ and a verifier $\mathcal{V}$. The client $\mathcal{P}$ aims to prove that $\mathsf{P}(R) = 1$ where R is from server $\mathcal{W}$, to verifier $\mathcal{V}$. To prevent client $\mathcal{P}$ from altering the data received from server $\mathcal{W}$, we use verifier $\mathcal{V}$ as a proxy to relay the messages between server $\mathcal{W}$ and client $\mathcal{P}$. Our protocol Π_{Proxy} consists of four phases: (1) handshake phase, (2) request phase, (3) response phase, and (4) prove phase. Next, we detail the four phases, respectively.

In the handshake phase, client $\mathcal{P}$ and server $\mathcal{W}$ perform the TLS handshake protocol, and the communication messages are relayed by verifier $\mathcal{V}$. Then, the client and the server can obtain the pair of initial client/server application key $(\mathsf{CATS}_0, \mathsf{SATS}_0)$, and derive the client write key/IV pair $(\mathsf{CWK}_0, \mathsf{CWIV}_0)$ and the server write key/IV pair $(\mathsf{SWK}_0, \mathsf{SWIV}_0)$ (see Section 3.1 for more details). Later, $(\mathsf{CWK}_0, \mathsf{CWIV}_0)$ and $(\mathsf{SWK}_0, \mathsf{SWIV}_0)$ would be updated, so we use $(\mathsf{CWK}_i, \mathsf{CWIV}_i)$ and $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ to denote the currently used key/IV pairs.

In the request and response phases, client $\mathcal{P}$ and server $\mathcal{W}$ still follow the specification of TLS to generate the request ciphertext c under $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ and the response ciphertext c' under $(\mathsf{CWK}_i, \mathsf{CWIV}_i)$. The verifier $\mathcal{V}$ is responsible for relaying the two ciphertexts.

In the prove phase, the client $\mathcal{P}$ invokes $\mathcal{F}_{\mathsf{NIZK}}$ to generate a proof π' proving that he holds a client write key/IV pair $(\mathsf{CWK}_i, \mathsf{CWIV}_i)$ that can decrypt the response ciphertext c' to m and $\mathsf{P}(\tilde{m}) = 1$ where $\tilde{m}$ is the pertinent message extracted from m. If the proof π' is valid, the verifier $\mathcal{V}$ outputs 1, otherwise, outputs 0.

Protocol Π_{Proxy}

There are a server $\mathcal{W}$, a client $\mathcal{P}$ (i.e., prover), and a verifier $\mathcal{V}$, and server $\mathcal{W}$ behaves the same as in TLS. A predicate $\mathsf{P}(\cdot)$ is to decide if the response from $\mathcal{W}$ satisfies some conditions (e.g., age is greater than 18).

Handshake phase:

- The client $\mathcal{P}$ and server $\mathcal{W}$ run TLS handshake protocol (see Section 3.1 for more details) via verifier $\mathcal{V}$ (i.e., the messages generated during handshake protocol execution are forwarded by $\mathcal{V}$) to obtain the pair of initial client and server application keys $(\mathsf{CATS}_0, \mathsf{SATS}_0)$;
- The client $\mathcal{P}$ and server $\mathcal{W}$ both compute $(\mathsf{CWK}_0, \mathsf{CWIV}_0) \leftarrow \mathsf{TLS.Derive}(\mathsf{client}, \mathsf{CATS}_0)$ and $(\mathsf{SWK}_0, \mathsf{SWIV}_0) \leftarrow \mathsf{TLS.Derive}(\mathsf{server}, \mathsf{SATS}_0)$;

Request phase:

- The client $\mathcal{P}$ computes $c \leftarrow \mathsf{AEAD.Enc}_{\mathsf{SWK}_i}(\mathsf{SWIV}_i, a, Q)$, where i is initialized as 0 and increases by 1 with each key update (described below), and a is the associated data in $\mathsf{TLS\ 1.3}$;
- The verifier $\mathcal{V}$ forwards (c, a) to server $\mathcal{W}$;

Response phase:

- Get the query $Q = \mathsf{AEAD.Dec}_{\mathsf{CWK}_i}(\mathsf{CWIV}_i, a, c)$, and obtains the response R according to the current dataset;
- Generate $c' = \mathsf{AEAD.Enc}_{\mathsf{CWK}_i}(\mathsf{CWIV}_i, a, R)$ and sends (c', a') to verifier $\mathcal{V}$;
- After receiving the response ciphertext (c', a') from server $\mathcal{W}$, verifier $\mathcal{V}$ forwards (c', a') to client $\mathcal{P}$;
- The client $\mathcal{P}$ computes $m = \mathsf{AEAD.Dec}_{\mathsf{CWK}_i}(\mathsf{CWIV}_i, a', c')$, if $m \neq \perp$, the process continues, otherwise it aborts;

Prove phase:

- The client $\mathcal{P}$ proves that the response satisfies the conditions defined by the predicate $\mathsf{P}(\cdot)$ as follows:
  - The client $\mathcal{P}$ generates a proof π' by invoking $\mathcal{F}_{\mathsf{NIZK}}$ (see Figure 15) to prove that he knows m, $\tilde{m}$ and $(\mathsf{CWK}_i, \mathsf{CWIV}_i)$ such that $m = \mathsf{AEAD.Dec}_{\mathsf{CWK}_i}(\mathsf{CWIV}_i, a', c')$ and $\tilde{m}$ is extracted from m, and sends π' to verifier $\mathcal{V}$;
  - If π' is verified successfully, verifier $\mathcal{V}$ outputs 1, otherwise 0.

Figure 9: Proxy-based Oracle Protocol

5. Vulnerability in Unrestricted Setting

While it is tempting to reason the protocol's security on general TLS connections, unfortunately, as we confirm the intuition of previous works [5], [8], the proxy-based oracle protocol is not secure if we only consider the oracle with unrestricted setting $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$. We present our findings below.

Theorem 5.1. *When the underlying AEAD is vulnerable to key-committing attacks (see Lemma 3.1), there exists some server $\mathcal{W}$ and predicate $\mathsf{P}(m)$ such that the protocol Π_{Proxy}*

does not realize the functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$.

Proof. Recall that in key-committing attack (see Definition 3.2), the adversary (i.e., the client $\mathcal{P}$) can generate a ciphertext c and two contexts (k, n, a) and (k', n', a), where $(k, n) \neq (k', n')$, such that the ciphertext c can be successfully decrypted under the two contexts. At first glance, the key-committing attack is not applicable to our scenario, as it is the server that generates the response ciphertext c. However, since the client $\mathcal{P}$ can obtain (k, n) after the handshake phase and the associated data a can be known beforehand according to the specification of TLS [1], [2], the client $\mathcal{P}$ can manipulate the data stored by the server through a side channel to build a ciphertext c. For instance, the client's balance in some bank can be manipulated by withdrawing or saving funds. Specifically, we then construct an attacker $\mathcal{A}$ that does the following:

1) $\mathcal{A}$ starts the protocol. It starts handshaking with the server $\mathcal{W}$ through verifier $\mathcal{V}$ to obtain (k, n).
2) $\mathcal{A}$ builds a ciphertext c and another context $(k', n') \neq (k, n)$, such that $m = \mathsf{Dec}_k(n, a, c)$ and $m' = \mathsf{Dec}_{k'}(n', a, c)$ through traditional key commitment attacks (see Lemma 3.1).
3) $\mathcal{A}$ modifies the server-side data to m via side-channel methods, such that the verifier $\mathcal{V}$ receives and records the response ciphertext c.
4) $\mathcal{A}$ uses (k', n') to convince the verifier $\mathcal{V}$ that m' is the corresponding plaintext. □

A real-life attack scenario can also be found in the famous Facebook attack [15]. In the Facebook attack, the attacker constructs a specific ciphertext *after* the handshake, when it knows the symmetric key that will be used in the communication. It has the server store this ciphertext, which it will be decrypted later using a different key other than the one in the handshake.

Nevertheless, not all hope is lost. We observe that the attack in the theorem exploited the fact that the AEAD scheme in TLS is not key-committing. In fact, as we will see in the theorem below, Π_{Proxy} realizes $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ if and only if the underlying AEAD is key-committing.

Theorem 5.2. *Given a secure AEAD with key commitment security (see Lemma 3.1), protocol Π_{Proxy} shown in Figure 9 can securely realize the oracle functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ defined in Figure 8 in the $\mathcal{F}_{\mathsf{NIZK}}$-hybrid model, against a static active adversary who can corrupt client $\mathcal{P}$ or verifier $\mathcal{V}$.*

Proof. We will show that for any adversary $\mathcal{A}$, we can construct a simulator $\mathcal{S}$ that can simulate the view of the corrupted client $\mathcal{P}$ and the corrupted verifier $\mathcal{V}$, such that any PPT environment $\mathcal{E}$ cannot distinguish the execution in the ideal world from that in the real world.

Corrupted client $\mathcal{P}$: We construct the simulator $\mathcal{S}_C$ for the corrupted client in Figure 10. Next, we prove that the simulated execution in the ideal world is indistinguishable from that in the real world.

- Hyb_0: The execution in the real world.

Simulator $\mathcal{S}_C$

Handshake phase:
- Upon receiving the handshake request from $\mathcal{A}$, sends $\langle \textsc{RequestHS}, sid \rangle$ to $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, and sends OK later;
- Upon receiving $\langle \textsc{ResponseHS}, sid, \mathrm{OK} \rangle$, simulates server $\mathcal{W}$ to generate the handshake transcript while simulating verifier $\mathcal{V}$ to forward the transcript;
- Derive $(\mathsf{CWK}_0, \mathsf{CWIV}_0)$ and $(\mathsf{SWK}_0, \mathsf{SWIV}_0)$, as the simulator simulates the secret information of server $\mathcal{W}$;

Request phase:
- Once receiving the request ciphertext (c, a) from $\mathcal{A}$, compute $Q = \mathsf{AEAD.Dec}_{\mathsf{SWK}_i}(\mathsf{SWIV}_i, a, c)$;
- Send $\langle \textsc{Response}, sid, Q \rangle$ to $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ and subsequent responds OK to $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$;

Response phase:
- Once receiving $\langle sid, R \rangle$ from $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, compute $c' = \mathsf{AEAD.Enc}_{\mathsf{CWK}_i}(\mathsf{CWIV}_i, a', R)$ and sends (c', a') to adversary $\mathcal{A}$;

Prove phase:
- If receiving π' from adversary $\mathcal{A}$, send OK to $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$.

Figure 10: Simulation for Corrupted Client.

- Hyb_1: The same as Hyb_0 except that the handshake transcript is generated by the simulator $\mathcal{S}$ rather than by server $\mathcal{W}$. This hybrid is perfectly indistinguishable from Hyb_0, as server $\mathcal{W}$ in Hyb_0 uses a uniformly random secret to generate the handshake transcript. Note that since the handshake transcript is changed, the corresponding $(\mathsf{CWK}_i, \mathsf{CWIV}_i, \mathsf{SWK}_i, \mathsf{SWIV}_i)$ is also changed.
- Hyb_2: The same as Hyb_1 except that the simulator $\mathcal{S}$ obtains the query Q from the ciphertext c and then uses Q to obtain the response R from $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$. This hybrid is perfectly indistinguishable from Hyb_1 due to the correctness of AEAD.
- Hyb_3: The same as Hyb_2 except that the output of verifier is changed to the output from $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$. This hybrid is computationally indistinguishable from Hyb_2 due to the key commitment of AEAD.

The indistinguishability between $\mathsf{Hyb}_0 \sim \mathsf{Hyb}_2$ is natural. Next, we detail why Hyb_3 and Hyb_2 are indistinguishable. Specifically, we show that if there is an adversary $\mathcal{A}$ such that environment $\mathcal{E}$ can distinguish Hyb_3 and Hyb_2, we can construct an environment $\mathcal{E}'$ to break the key commitment of AEAD.

Assuming that the environment $\mathcal{E}$ chooses a response R such that $\mathsf{P}(R) = 0$ for server $\mathcal{W}$, verifier $\mathcal{V}$ in Hyb_3 will output 0 with probability 1. Therefore, when $\mathcal{E}$ obtains 1 from verifier $\mathcal{V}$, the environment $\mathcal{E}$ learns that the execution is in Hyb_2. The environment $\mathcal{E}'$ runs a copy of $\mathcal{E}$ and

Simulator $\mathcal{S}_V$

Handshake phase:

- Once receiving $\langle \text{RequestHS}, sid \rangle$ and $\langle \text{ResponseHS}, sid, \text{OK} \rangle$ from $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, simulate the handshake transcript generated by client $\mathcal{P}$ server $\mathcal{W}$;
- Derive $(\mathsf{CWK}_0, \mathsf{CWIV}_0)$ and $(\mathsf{SWK}_0, \mathsf{SWIV}_0)$, as the simulator $\mathcal{S}$ simulates the secret information of server $\mathcal{W}$ and client $\mathcal{P}$;

Request phase:

- Once receiving $\langle sid, |Q| \rangle$ from $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, generates a query Q of length $|Q|$, and encrypts it to ciphertext c under $(\mathsf{CWK}_i, \mathsf{CWIV}_i)$, and then sends c to $\mathcal{A}$;
- Once reading (c, a) from the output tape of $\mathcal{A}$, sends OK to $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$;

Response and prove phase:

- Once receiving $\langle sid, |R| \rangle$ from $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$, sends OK to $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$;
- Generate a ciphertext c' according to $|R|$ and send it to $\mathcal{A}$, if read c' from the output tape of $\mathcal{A}$, sends OK to $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ in place of $\mathcal{A}$;
- Once receiving $\langle sid \rangle$ from $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, send OK to obtain $\mathsf{P}(R)$;
- Rewind $\mathcal{A}$ to the second step of this phase, and then perform the following:
  - Generates a message $\tilde{m}$ such that $\mathsf{P}(\tilde{m}) = \mathsf{P}(R)$, and then pads $\tilde{m}$ to m of length $|R|$;
  - Encrypts m to ciphertext c' under $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ and sends c' to $\mathcal{A}$ to simulate the response of server $\mathcal{W}$;
  - Use $((\mathsf{SWK}_i, \mathsf{SWIV}_i), m)$ as witness to generate a proof π' to simulate client $\mathcal{P}$;

Figure 11: Simulation for Corrupted Verifier.

simulates the simulator $\mathcal{S}$ (internally running a copy of $\mathcal{A}$) and $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ for $\mathcal{E}$. More specifically, when $\mathcal{A}$ queries a random oracle (used in handshake), $\mathcal{E}'$ simulates $\mathcal{S}$ to output randomnesses to simulate $(\mathsf{CWK}_i, \mathsf{CWIV}_i, \mathsf{SWK}_i, \mathsf{SWIV}_i)$. Then $\mathcal{E}'$ simulates $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ to send $\langle sid, Q \rangle$ to $\mathcal{E}$, and obtains a response R where $\mathsf{P}(R) = 0$. Then, $\mathcal{E}'$ simulates $\mathcal{S}$ to send (c', a') to $\mathcal{A}$. The environment $\mathcal{E}$' also needs to simulate $\mathcal{F}_{\mathsf{NIZK}}$ and receive the proving request including the witness $(c', R', (\mathsf{SWK}'_i, \mathsf{SWIV}'_i))$, from $\mathcal{A}$. If $R \neq R'$ and $(\mathsf{SWK}_i, \mathsf{SWIV}_i) \neq (\mathsf{SWK}'_i, \mathsf{SWIV}'_i)$, $\mathcal{E}'$ uses c', $(R', (\mathsf{SWK}'_i, \mathsf{SWIV}'_i))$ and $(R, (\mathsf{SWK}_i, \mathsf{SWIV}_i))$ to break key commitment of AEAD.

Corrupted verifier: We construct the simulator $\mathcal{S}_V$ for the corrupted verifier in Figure 11. Next, we prove that the simulated execution in the ideal world is indistinguishable from that in the real world.

- Hyb_0: The execution in the real world.
- Hyb_1: The same as Hyb_0 except that the handshake transcript is generated by the simulator $\mathcal{S}$ rather than by server $\mathcal{W}$ and client $\mathcal{P}$. This hybrid is perfectly indistinguishable from Hyb_0, as server $\mathcal{W}$ and client $\mathcal{P}$ in Hyb_0 use uniformly random secrets to generate the handshake transcript. Note that since the handshake transcript is changed, the corresponding $(\mathsf{CWK}_i, \mathsf{CWIV}_i, \mathsf{SWK}_i, \mathsf{SWIV}_i)$ is also changed.
- Hyb_2: The same as Hyb_1 except that the query Q changes to the one chosen by the simulator. This hybrid is computationally indistinguishable from Hyb_1 due to the CPA-security of AEAD.
- Hyb_3: This hybrid is the same as Hyb_2 except that the messages m and $\tilde{m}$ are generated by the simulator $\mathcal{S}$ according to $\mathsf{P}(R)$. The hybrid is computationally indistinguishable from Hyb_2 due to the CPA-security of AEAD, and the completeness and zero-knowledge property of $\mathcal{F}_{\mathsf{NIZK}}$.

□

```
HTTP/1.1 200 OK
Date: Sat, 30 Dec 2023 18:52:39 GMT
Server: Apache/2.4.52 (Ubuntu)
Last-Modified: Mon, 11 Dec 2023 01:10:39 GMT
ETag: "25c9-60c319a24a0d5-gzip"
Accept-Ranges: bytes
Vary: Accept-Encoding
Access-Control-Allow-Origin: *
Content-Length: 9673
Connection: close
Content-Type: text/html
```

Figure 12: An HTTP response header from an Apache server. The bold part (status code and date) can be used as a generic variable padding for all HTTP responses.

6. Variable Padding and HTTPS Server

The previous section shows that key commitment is necessary and sufficient for a secure oracle protocol. However, with all that being said, key committing is simply *not* a property present in the current TLS protocol. Fortunately, we find that all HTTPS responses contain an HTTP header at the start of the plaintext, which can be used to prevent key-committing attacks. In this section, we define a notion called *variable padding* to capture the padding-like structure existing in HTTPS responses. Furthermore, we prove that when considering plaintexts with variable padding, both AES-GCM and ChaCha20-Poly1305 are secure against key-committing attacks. Next, we first give more details about variable padding.

6.1. Variable Padding

Figure 12 contains a typical response header from an Apache server. The sample header has 308 bytes of text — enough for 19 blocks of AES. Albertini et al. [16] have shown that AEAD schemes can be made key-committing with 4 blocks of fixed padding. However, we observe that an adversarial prover in our game has some wiggle room for the header, because it may be able to determine parameters such as HTTP status code and response date. Nevertheless, we find its capability highly constrained, since all of these parameters have limited ranges of value. For instance, the HTTP status code has only 63 variations. If we take a rather generous time window of one hour between the response of the server and the receival of the verifier, then the response date has no more than 3600 different variations. This motivates us to define the concept of variable padding: padding with limited variations controlled by the adversary.

Definition 6.1 (Variable padding)**.** *Consider a set S that consists of only λ-bit strings. We say that a string m is variably padded by S, if the λ-bit prefix of m is in S.*

For instance, consider the sample header in Figure 12. We can easily verify that the first 54 bytes consist of only the status code and the date as the variable parameters. Therefore, we can assert that there exists a 54-byte string set S with $|S| = 63 \times 3600$, such that all response plaintext produced by this particular server is variably padded by S.

Application to Generic HTTPS Servers. While to achieve the tightest security bound, the specific padding and length should be analyzed on a per-server basis, we can nevertheless demonstrate a generic bound based on the HTTP status code and date header.

Corollary 6.1. *Every valid HTTP response is variably padded by a* 54*-byte string set S with $|S| = 63t$ if it follows good practice and is received within t seconds.*

This corollary is verified by noticing that

1) RFC 7231 requires any server with a reasonable clock to send the date header, and states 'it is good practice to send header fields that contain control data first, such as… Date on responses.' Notably, both Apache and Nginx, the two popular HTTP server implementations, have Server and Date as the first two response headers.
2) The HTTP header with a date takes at least 54 bytes.

6.2. Key Commitment with Variable Padding

Next, we consider an adversarial prover's advantage with a variably padded plaintext, in AES-GCM and ChaCha20-Poly1305. Since the verifier has access to the ciphertext transcript, the attacker at a bare minimum needs to come up with some ciphertext c and two pairs of key and nonce $((k, n), (k', n'))$ such that both $\mathsf{Dec}_k(n, a, c)$ and $\mathsf{Dec}_{k'}(n', a, c)$ give a plaintext variably padded by some S.

Analysis on AES-GCM. We first formalize the adversary's advantage in AES-GCM.

Theorem 6.1 (AES-GCM Key commitment with variable padding)**.** *Assume E is an ideal block cipher with l-bit block size and $(\mathsf{Enc}, \mathsf{Dec})$ is a GCM scheme defined on E. The adversary's advantage of constructing a ciphertext c, an associated data a, and two key/nonce pairs $((k, n), (k', n'))$ such that both $\mathsf{Dec}_k(n, a, c)$ and $\mathsf{Dec}_{k'}(n', a, c)$ give a plaintext variably padded by some λ-bit string set S within q queries to E is at most $\epsilon = \frac{q^2|S|^2}{2^{\lambda-2v+1}}\left(\frac{2^l}{2^l-b}\right)^{b+1}$, where $b = \lfloor \frac{\lambda}{l} \rfloor$ and v is the bit-length of IV.*

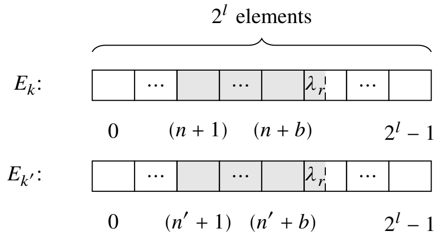


Figure 13: A representation of E_k and $E_{k'}$ as permutations. The shaded part in $E_{k'}$ is uniquely determined from the shaded part in E_k.

Proof. We first fix a key/nonce pair (k, n) and two prefixes (s, s') and consider the bad event BAD where there exists another key/nonce pair (k', n') and a ciphertext c such that $\mathsf{Dec}_k(n, a, c)$ has prefix s and $\mathsf{Dec}_{k'}(n', a, c)$ has prefix s'. Let us upper-bound the probability $\Pr(\mathsf{BAD})$ as follows.

We assume that the λ-bit prefix spans across b blocks and λ_r extra bits, which means that $\lambda = l \cdot b + \lambda_r$. We know that for some ciphertext c, $\mathsf{Dec}_k(n, a, c)$ has prefix s and $\mathsf{Dec}_{k'}(n', a, c)$ has prefix s'. Therefore, according to the AES-GCM encryption process (see Equation 2), the following equation set holds:

$$\begin{aligned}
E_k(n+1) + s[0:l] &= c[0:l] \\
&= E_{k'}(n'+1) + s'[0:l], \\
E_k(n+2) + s[l:2l] &= c[l:2l] \\
&= E_{k'}(n'+1) + s'[l:2l], \\
&\vdots \\
E_k(n+b) + s[(b-1)l:bl] &= c[(b-1)l:bl] \\
&= E_{k'}(n'+b) + s'[(b-1)l:bl], \\
E_k(n+b+1) + s[bl:\lambda] &= c[bl:\lambda] \\
&= E_{k'}(n'+b+1) + s'[bl:\lambda].
\end{aligned}$$

Recall that we consider fixed (k, n) and prefixes s and s'. Therefore, the equation set above tells us that b blocks and λ_r extra bits of $E_{k'}$ are also fixed. Next, we calculate how many $E_{k'}$ can satisfy the requirement.

Because E is an ideal cipher and block size is l bits, both E_k and $E_{k'}$ are permutations on 2^l elements. In Figure 13, we show the results of the permutation on the 2^l elements. We assume that from $(n+1)$-th block to $(n+b)$-th block are fixed and the first λ_r bits in the $(n+b+1)$-th block are also fixed. We can see that the remaining $(l - \lambda_r)$ bits in the $(n+b+1)$-th block can be randomly chosen from $\{0,1\}^{l-\lambda_r}$. Once the remaining $(l - \lambda_r)$ bits in the $(n+b+1)$-th block are chosen, the $(b+1)$ blocks are fixed. Then, the remaining $(2^l - b - 1)$ blocks have a total of $(2^l - b - 1)!$ possible assignments. Therefore, the upper bound of the number of $E_{k'}$ satisfying the above requirement is given by

$$|E_{k'}| \le 2^{l-\lambda_r}(2^l - b - 1)!.$$

Without considering the above requirement, $E_{k'}$ could be randomly chosen from $(2^l)!$ permutations. Hence the probability of BAD happening is at most

$$\begin{aligned}\Pr(\mathsf{BAD}) &\le \frac{2^{l-\lambda_r}(2^l - b - 1)!}{(2^l)!} < \frac{2^{l-\lambda_r}}{(2^l - b)^{b+1}} \\ &= \frac{1}{2^\lambda}\left(\frac{2^l}{2^l - b}\right)^{b+1}.\end{aligned}$$

Now the total probability is bounded by the summation of all different keys the adversary tests (note that the adversary queries E at most q times) and all different nonces and prefixes:

$$\begin{aligned}\Delta^{\mathcal{A}} &< \underbrace{\binom{q}{2}}_{\text{keys}} \cdot \underbrace{2^{2v}}_{\text{nonces}} \cdot \underbrace{|S|^2}_{\text{prefixes}} \cdot \Pr(\mathsf{BAD}) \\ &< \frac{q^2|S|^2}{2^{\lambda-2v+1}}\left(\frac{2^l}{2^l - b}\right)^{b+1}. \qquad \square\end{aligned}$$

Since in TLS, the nonce n for AES-GCM is uniquely determinely by a 96-bit IV, we have $v = 96$. Plugging in the generic HTTP padding with $\lambda = 8 \times 56$ bits, $l = 128$ bits, $|S| = 63t$ and a generous $t = 3600$ seconds gives us $\epsilon \le 2^{-221}q^2$. Note that $\left(\frac{2^l}{2^l - b}\right)^{b+1} \approx 1$.

Therefore, the protocol is highly secure, even on HTTPS servers with minimal features.

Analysis on ChaCha20-Poly1305. Similarly, below we demonstrate the security of ChaCha20-Poly1305.

Theorem 6.2 (ChaCha20-Poly1305 Key commitment with variable padding)**.** *Assume $H : \mathbb{Z}^{|k|} \times \mathbb{Z}^{|n|} \to \mathbb{Z}^l$ is an ideal random function representing ChaCha20 and $(\mathsf{Enc}, \mathsf{Dec})$ is a Poly1305 scheme defined on H. The adversary's advantage of constructing a ciphertext c, an associated data a, and two key/nonce pairs $((k, n), (k', n'))$ such that both $\mathsf{Dec}_k(n, a, c)$ and $\mathsf{Dec}_{k'}(n', a, c)$ give a plaintext variably padded by some λ-bit string set S within q queries to H is at most $\epsilon = \frac{q^2|S|^2}{2^{\lambda-2v+1}}$, where v is the bit-length of IV.*

Proof. The only difference in the analysis between AES-GCM and ChaCha20-Poly1305 is that AES is a random permutation and ChaCha20 is a random function. Therefore, following a similar analysis, we can bound the probability of getting a bad random function to

$$\Pr(\mathsf{BAD}) \le \frac{2^{l-\lambda_r}(2^l)^{2^l-b-1}}{(2^l)^{2^l}} = \frac{2^{l-\lambda_r}}{(2^l)^{b+1}} = \frac{1}{2^\lambda}.$$

Therefore, the overall probability is bounded by

$$\Delta^{\mathcal{A}} < \binom{q}{2} \cdot 2^{2v} \cdot |S|^2 \cdot \Pr(\mathsf{BAD}) < \frac{q^2|S|^2}{2^{\lambda-2v+1}}. \qquad \square$$

Similarly, in TLS, the nonce n for ChaCha20-Poly1305 is uniquely determined by a 96-bit IV. Therefore, we have $v = 96$. Plugging in the generic HTTP padding with $\lambda = 8 \times 56$ bits, $l = 512$ bits, $|S| = 63t$ and $t = 3600$ seconds gives us $\epsilon \le 2^{-221}q^2$.

An application of Theorem 6.1 and Theorem 6.2 gives us the following desired security property.

Theorem 6.3. *Assume TLS uses either AES-GCM or ChaCha20-Poly1305. Fix some set variable padding S of polynomial size. Let $\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$ denote the same functionality as $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, except that the server's response is guaranteed to be variably padded by S. Protocol Π_{Proxy} shown in Figure 9 can securely realize the oracle functionality $\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$ defined in Figure 8 in the $\mathcal{F}_{\mathsf{NIZK}}$-hybrid model, against a static active adversary who can corrupt client $\mathcal{P}$ or verifier $\mathcal{V}$.*

The proof follows a direct application of the above analysis on AES-GCM and ChaCha20-Poly1305 to Theorem 5.2.

7. Beyond HTTPS: without Variable Padding

We have shown that proxying is secure over HTTPS thanks to the variable padding. Nonetheless, exploring the potential usage of proxying over home-bake protocols is still an interesting topic from a theoretical perspective. We have seen that proxying is not secure given any home-bake protocol: the Facebook attack [15] is a practical counter-example. Therefore, we must restrict the adversarial prover's power to some extent. In this section, we explore the scenario where the prover must fix the server's response before the protocol. This is common in many daily scenarios. For instance, an adversarial prover should not be able to change his age depending on the handshake secret.

We formalize the intuition as functionality $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ in Figure 14. Unlike the functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, the functionality $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ obtains the data related to client $\mathcal{P}$, denoted as $\mathsf{Data}_{\mathcal{P}}$, from the environment $\mathcal{E}$, before the handshake phase. Then, in the response phase, the functionality $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ applies the query Q to $\mathsf{Data}_{\mathcal{P}}$ to obtain the response R. This reflects that $\mathsf{Data}_{\mathcal{P}}$ is fixed before the handshake phase and cannot be modified later.

7.1. Context Forgery Attack

What we are considering is akin to a forgery attack: our adversarial attacker must come up with an additional explanation of the ciphertext provided by the server. We notice that the context discovery (CDY) attack on AEAD (see Definition 3.1) considered by Menda et al. [22] is closely related to our problem, but there are notable differences.

Specifically, the adversarial $\mathcal{P}$ considered by us has more knowledge than what is described under the CDY model in Menda et al. [22], since he knows the original context (i.e., key/nonce pair) of the ciphertext. Moreover, he also has a different goal in mind: he cannot just find any context that decrypts the ciphertext — he has to find one that is different from the original context. Based on these differences, we propose a new model named context forgeability (CFY) and relate this to the second-preimage attack on a hash function,

Functionality $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$

The predicate $\mathsf{P}(\cdot)$ and the boolean value Mode_p are the same as in $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ (see Figure 8). The data related to $\mathcal{P}$, denoted as $\mathsf{Data}_{\mathcal{P}}$.

Functionality:

- Send $\langle \textsc{RequestRecord}, sid, \mathcal{P}\rangle$ to server $\mathcal{W}$, if receiving $\langle \textsc{ResponseRecord}, sid, \mathsf{Data}_{\mathcal{P}}\rangle$ from server $\mathcal{W}$, store $\mathsf{Data}_{\mathcal{P}}$ and continue, otherwise, wait and ignore the subsequent messages;

Handshake and Request phase:

- The same as in $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ (see Figure 8);

Response phase:

- Upon receiving $\langle \textsc{Response}, sid, \mathrm{OK}\rangle$ from server $\mathcal{W}$, obtain response R from $\mathsf{Data}_{\mathcal{P}}$ according to the request message Q, and send $\langle sid, |R|\rangle$ to adversary $\mathcal{S}$ and verifier $\mathcal{V}$;
- Upon receiving OK from adversary $\mathcal{S}$ and verifier $\mathcal{V}$, send $\langle sid, R\rangle$ to client $\mathcal{P}$;

Prove phase:

- The same as in $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ (see Figure 8).

Figure 14: Oracle Functionality with Fixed Dataset (the differences from $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ (in Figure 8) are marked with underline). Compared to $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ stores client $\mathcal{P}$'s data (i.e., $\mathsf{Data}_{\mathcal{P}}$), to ensure that $\mathsf{Data}_{\mathcal{P}}$ will not be changed once the execution starts; $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ generates response R from $\mathsf{Data}_{\mathcal{P}}$ rather than obtaining it from server $\mathcal{W}$.

similar to how CDY is to CMT as a preimage attack is to a collision attack. More formally, we can define the context forgery attack game as:

Definition 7.1 (Context Forgery)**.** *Fix some AEAD parameter pp and a corresponding AEAD oracle* Π. *The game* $\mathsf{CDY}^{\dagger}$ *is defined as:*

1) *The challenger samples a random ciphertext c from some ciphertext space and its corresponding decryption context* (k, n, a).
2) *The challenger sends* (c, k, n, a) *to some adversary* $\mathcal{A}$.
3) *The adversary wins if it outputs a valid context* (k', n', a) *with* $(k, n) \neq (k', n')$ *that decrypts c successfully.*

The adversary's q-advantage $\Delta^{\mathcal{A}}_{\mathsf{CDY}^{\dagger}}$ *is defined as the probability it wins under q queries to the AEAD oracle* Π.

Analysis on AES-GCM. AES-GCM is known for various commitment weaknesses [15], and it is not difficult to intuit that it is not secure in this scenario. Since a CDY attack implies a CFY attack (see Corollary 7.2), we can use ideas that Menda et al. [22] proposed for the CDY attack to give a similar attack for CFY.

Theorem 7.1. *There is an attack under* $\mathsf{CFY}^{\dagger}$ *that breaks E-GCM with probability at least* $\frac{q}{2^{33}}$, *assuming E is an 128-bit ideal block cipher.*

Proof. For simplicity, let us consider a message with one block of ciphertext and no associated data. If we obtain some key k, nonce n and the only ciphertext block c from the input, the tag t follows the definition of GCM

$$t = E_k(n) + E_k(0)^2 c + E_k(0).$$

Now suppose we sample some other key $k' \neq k$, then we can have the following equation, given the fixed tag t and ciphertext block c.

$$t = E_{k'}(n') + E_{k'}(0)^2 c + E_{k'}(0).$$

It is obvious that given $(E_{k'}(0), c, t)$, $E_{k'}(n)$ can be solved using the above equation. Moreover, since $E_{k'}$ is reversible, we can solve for n'. In AES-GCM the nonce must end with $(0^{31}\|1)$, so this particular n' has $\frac{1}{2^{32}}$ probability of satisfying the requirement. Observing that this try of k' uses two queries of E (that is, one is for $E_{k'}(0)$ and the other is for decrypting $E_{k'}(n')$), we can bound the success probability of q queries to $\frac{q}{2^{33}}$. □

The above attack can be generalized to any length of associated data and ciphertext, as discussed by Menda et al. [22]. Moreover, many works [15], [22] have shown that AES-GCM polyglot ciphertext can be decrypted to different *meaningful* plaintext under different keys.

Analysis on ChaCha20-Poly1305. To analyze game $\mathsf{CFY}^{\dagger}$ under ChaCha20-Poly1305, we first abstract its authentication tag computation mechanism as

$$t = \mathsf{poly}(r) + s,$$

where poly is a polynomial whose coefficient depends only on the concatenation of the associated data and the ciphertext, and $(r, s) = H(k, n)[0 : 256]$ is the secret generated by the ChaCha20 pseudorandom function. We then demonstrate its security in $\mathsf{CFY}^{\dagger}$.

Theorem 7.2. *The adversary's advantage in* $\mathsf{CFY}^{\dagger}$ *when attacking H-Poly1305 is no more than* $\frac{q}{2^{128}}$, *where q is the number of queries the adversary made to H and* $H(k, n)$ *is a 512-bit random oracle.*

Proof. We start by fixing some key and nonce (k', n') the adversary has queried for the first time. Observe that since H is a random oracle, $\Pr\left(H(k', n') = h\right)$ is a uniform distribution regardless of any prior queries. Therefore, $(r', s') = H(k', n')[0 : 256]$ is also uniformly distributed over the whole range.

Now observe that in order for (k', n') to satisfy the tag requirement, we have

$$\mathsf{poly}(r') + s' = t = \mathsf{poly}(r) + s.$$

Rearrange and we have

$$s' - s = \mathsf{poly}(r) - \mathsf{poly}(r').$$

Observe that for some fixed s, $s' - s$ is still uniformly distributed over $\mathbb{Z}/2^{128}\mathbb{Z}$. Denote $\Delta p(r') = \mathsf{poly}(r) - \mathsf{poly}(r')$.

We can bound the probability of (k', n') satisfying the tag requirement $\Pr(\mathsf{BAD})$ by

$$\begin{aligned}\Pr(\mathsf{BAD}) &= \sum_{i \in \mathbb{Z}/2^{128}\mathbb{Z}} \Pr(s' - s = i) \Pr(\Delta p(r') = i) \\ &= \frac{1}{2^{128}} \sum_{i \in \mathbb{Z}/2^{128}\mathbb{Z}} \Pr(\Delta p(r') = i) = \frac{1}{2^{128}}\end{aligned}$$

Since the attacker makes q queries, we bound the probability of success to $\frac{q}{2^{128}}$. □

Protocol Security Under CFY. We demonstrate that the proxying protocol Π_{Proxy} realizes $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ as long as the underlying AEAD is secure under $\mathsf{CFY}^{\dagger}$.

Theorem 7.3. *Given a secure AEAD with context unforgeability security under $\mathsf{CFY}^{\dagger}$, the protocol Π_{Proxy} shown in Figure 9 can securely realize the functionality $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ in Figure 14 in the $\mathcal{F}_{\mathsf{NIZK}}$-hybrid model, against a static active adversary who can corrupt client $\mathcal{P}$ or verifier $\mathcal{V}$.*

Proof. The simulation and the hybrid argument are similar to those in Theorem 5.2. Next, we only detail the main difference that arises in the indistinguishability between Hyb_2 and Hyb_3 for corrupted client $\mathcal{P}$.

Assuming that there is an environment $\mathcal{E}$ that can distinguish Hyb_3 and Hyb_2, we can construct an environment $\mathcal{E}'$ to break the context forgeability security of AEAD. Similarly, the environment $\mathcal{E}$ chooses a dataset $\mathsf{Data}_{\mathcal{P}}$ and a query Q such that applying Q to it yields a response R with property $\mathsf{P}(R) = 0$. The verifier $\mathcal{V}$ in Hyb_3 will output 0 with probability 1. Therefore, when $\mathcal{E}$ obtains 1 from verifier $\mathcal{V}$, the environment $\mathcal{E}$ learns that the execution is in Hyb_2.

The environment $\mathcal{E}'$ runs a copy of $\mathcal{E}$ and simulates the simulator $\mathcal{S}$ (internally running a copy of $\mathcal{A}$) and $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ for $\mathcal{E}$. The environment $\mathcal{E}'$ can obtain $\mathsf{Data}_{\mathcal{P}}$ and the query Q from $\mathcal{E}$. After applying the query Q on $\mathsf{Data}_{\mathcal{P}}$, $\mathcal{E}'$ can obtain the response R. Then, $\mathcal{E}'$ sends R to the CFY challenger $\mathcal{C}_{\mathsf{CFY}}$, and receives (C, K, N) from the challenger $\mathcal{C}_{\mathsf{CFY}}$. When $\mathcal{A}$ queries random oracle for $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$, $\mathcal{E}'$ sends (K, N) as response. The environment $\mathcal{E}'$ needs to simulate $\mathcal{F}_{\mathsf{NIZK}}$, and receive the proving request, which includes the witness $(C, R', (K', N'))$ (i.e., $(C, R', \mathsf{SWK}'_i, \mathsf{SWIV}'_i)$), from $\mathcal{A}$. If $(R \neq R')$ and $(K, N) \neq (K', N')$, $\mathcal{E}'$ sends (R', K', N') to break CFY.

□

7.2. Relationship between CMT, CFY and CDY

An interesting observation is that a similar hierarchy exists concerning CDY, CFY and CMT just like the hash function. CMT-security implies CFY-security, and CFY-security implies CDY-security assuming context compression. We provide our insight in the following two corollaries and defer a formal proof on the generalized case under Menda et al.'s framework to Appendix A.

Corollary 7.1. *For some AEAD Π and any adversary $\mathcal{A}$ that wins the $\mathsf{CFY}^{\dagger}$ game with advantage $\Delta^{\mathcal{A}}_{\mathsf{CFY}^{\dagger}}$, there exists an adversary $\mathcal{B}$ that wins the $\mathsf{CMT}^{\dagger}$ game with advantage $\Delta^{\mathcal{B}}_{\mathsf{CMT}^{\dagger}} = \Delta^{\mathcal{A}}_{\mathsf{CFY}^{\dagger}}$.*

The relationship between CFY and CDY, on the other hand, is not so clear-cut. While a CDY attack on an AEAD scheme often implies a CFY attack, it is not universally true. In the case where a ciphertext cannot be decrypted under two different contexts (i.e., the following BadCtx event), a CDY attack may be easy but a CFY attack will be impossible. However, similar to the analysis of Menda et al. on CDY, we can bound the advantage of the adversary with the probability of BadCtx:

Corollary 7.2. *For some AEAD Π and any adversary $\mathcal{A}$ that wins the $\mathsf{CDY}^{\dagger}$ game with advantage $\Delta^{\mathcal{A}}_{\mathsf{CDY}^{\dagger}}$, there exists an adversary $\mathcal{B}$ that wins the $\mathsf{CFY}^{\dagger}$ game with advantage $\Delta^{\mathcal{B}}_{\mathsf{CFY}^{\dagger}}$ such that*

$$\Delta^{\mathcal{B}}_{\mathsf{CFY}^{\dagger}} \geq \frac{1}{2}(1 - \Pr[\mathsf{BadCtx}])\Delta^{\mathcal{A}}_{\mathsf{CDY}^{\dagger}},$$

where BadCtx is the event that for a randomly sampled (K, N, M, A), the resulting ciphertext C can only be decrypted when the associated data is A.

8. Discussion

Multi-Round Support. We observe that the NIZK proof of the protocol does not reveal the key/nonce pair used in the communication. Therefore, our protocol naturally supports the client and the server communicating in multiple rounds after a handshake connection, as in the original TLS.

In practice, when the data do not involve privacy, the client can choose to directly reveal the key/nonce pair and data to the verifier, rather than using the NIZK proof. In this case, we note that TLS 1.3 supports a key update mechanism that allows us to refresh the key/nonce pair for subsequent communications without needing another handshake.

Side-Channel Connection. Our protocol does not stop an adversarial client from connecting to the server simultaneously without proxying through the verifier. In fact, when the web server is not restricted in its response, there is an attack using a side channel connection (see Theorem 5.1), and we prove that the HTTPS response format can be used to defend against the attack. Moreover, in our scenario, the purpose of the client is to have the data obtained from the server validated by the verifier so that the data can be submitted to the blockchain. Therefore, the client has no motivation to bypass the verifier when he tries to acquire the data.

9. Related Work

AEAD Commitment Security. It is known that a lot of AEAD schemes do not have the most robust commitment security. One prominent example of an exploit is the attack on Facebook Messenger by Dodis et al. [15]. Commitment security on AEAD has received a lot of research since around that time [21], [29]. Menda et al. provided a generalization and analysis on the topic [22], which we leverage in this paper. Albertini et al. analyzed several possible ways to generate polyglot ciphertext and provided possible patches to fix popular AEAD schemes [16].

Our work builds and extends on the framework by Menda et al. [22] and gives it a practical scenario. With this toolset, we are able to reason the security of proxying and give concrete security bounds with proof.

TLS Proxy & Oracle Protocols. Using TLS under multi-party scenarios has been investigated a number of times under different scenarios [30], [31], [32], [33]. TLS oracle protocols have also recently been studied by a variety of academic papers. Earlier results often include modifying the TLS server to some extent and using trusted hardware which are considered not universal [6], [7]. DECO by Zhang et al. [5] in 2020 is one of the first papers that combines a TLS oracle with zero-knowledge proof to preserve user privacy over a general TLS server. Janus by Lauinger et al. [8] demonstrates an efficient two-party computation that optimizes the performance of zero-knowledge proof by the client. A work in a similar direction by Xie et al. [9] uses the garble-then-prove technique instead. DIDO by Chan et al. [10] proposes further optimization based on TLS 1.3. Nevertheless, we notice that all these researches require a three-party handshake at the beginning of the protocol so that the verifier can inject randomness to preserve data integrity, which impacts their performance.

We consider our work parallel to this line of work since we do not specify which particular zero-knowledge proof protocol we should use, but rather point out an improvement in theory that allows us to eliminate the three-party handshake under common circumstances. Integrating proxying and an efficient zero-knowledge proof protocol is an interesting system work left for the future.

Meanwhile, there is also significant industry effort on this topic. The recent Reclaim Protocol [12] provides an implementation that is based purely on proxying without the three-party handshake but provides no security proof of the integrity of the data. Thus our work is also a theoretical discussion of the industry effort that validates its usage under common scenarios but also points out its limitations.

10. Conclusion

In this paper, we formalize the notion of a proxy TLS oracle protocol that does not enter any communication between the client and the server like previous works. We first reason for its limitation on arbitrary TLS protocols, confirming the intuition of previous works.

We then point out its potential use scenarios with common protocols. We proved that the proxy protocol is secure under an application layer protocol that consists of a variable padding, such as HTTPS. Given that HTTPS is the overwhelmingly popular protocol over the Internet, we believe that the proxy protocol provides a much simpler design with little trade-off.

We further explore the scenario where the application layer does not provide variable padding, such as various home-brew protocols. We showed that if the adversary cannot tamper with the response after establishing the connection, such as in the scenario with age verification, context unforgeability (CFY) of the underlying AEAD is sufficient to demonstrate security. We further analyzed the cipher suites in TLS, and found that AES-GCM does not satisfy the property, but ChaCha20-Poly1305 does.

Acknowledgments. This work is supported partially by the National Science Foundation under grant CNS-1846316. We would like to extend our thanks to the Reclaim Protocol for offering their codebase for examining the use cases of the TLS oracle protocol. We would also like to thank Kirill Kutsenok from the Reclaim Protocol for helpful discussions and suggestions.

References

[1] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[2] E. Rescorla and T. Dierks, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, Aug. 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5246

[3] V. Buterin, "Ethereum white paper: A next-generation smart contract and decentralized application platform," https://finpedia.vn/wp-content/uploads/2022/02/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf, accessed: March 28, 2024.

[4] "What is the blockchain oracle problem?" https://blog.chain.link/what-is-the-blockchain-oracle-problem/, accessed: March 28, 2024.

[5] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels, "DECO: Liberating web data using decentralized oracles for TLS," in *ACM CCS 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM Press, Nov. 2020, pp. 1919–1938.

[6] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *ACM CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM Press, Oct. 2016, pp. 270–282.

[7] H. Ritzdorf, K. Wüst, A. Gervais, G. Felley, and S. Capkun, "TLS-N: Non-repudiation over TLS enablign ubiquitous content signing," in *NDSS 2018*. The Internet Society, Feb. 2018.

[8] J. Lauinger, J. Ernstberger, A. Finkenzeller, and S. Steinhorst, "Janus: Fast privacy-preserving data provenance for TLS 1.3," Cryptology ePrint Archive, Report 2023/1377, 2023, https://eprint.iacr.org/2023/1377.

[9] X. Xie, K. Yang, X. Wang, and Y. Yu, "Lightweight authentication of web data via garble-then-prove," Cryptology ePrint Archive, Report 2023/964, 2023, https://eprint.iacr.org/2023/964.

[10] K. Y. Chan, H. Cui, and T. H. Yuen, "DIDO: data provenance from restricted TLS 1.3 websites," in *Information Security Practice and Experience*. Springer Nature, 2023, pp. 154–169.

[11] L. Breidenbach, C. Cachin, B. Chan, A. Coventry, S. Ellis, A. Juels, F. Koushanfar, A. Miller, B. Magauran, D. Moroz *et al.*, "Chainlink 2.0: Next steps in the evolution of decentralized oracle networks," 2021. [Online]. Available: https://chain.link/

[12] R. Protocol, "Reclaim protocol: Claiming and managing self-sovereign credentials," 2023. [Online]. Available: https://www.reclaimprotocol.org/

[13] S. Celi, A. Davidson, H. Haddadi, G. Pestana, and J. Rowell, “Distefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more,” Cryptology ePrint Archive, Paper 2023/1063, 2023, https://eprint.iacr.org/2023/1063.

[14] “The 2021 TLS telemetry report,” https://www.f5.com/labs/articles/threat-intelligence/the-2021-tls-telemetry-report, accessed: April 10, 2024.

[15] Y. Dodis, P. Grubbs, T. Ristenpart, and J. Woodage, “Fast message franking: From invisible salamanders to encryptment,” in *CRYPTO 2018, Part I*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, Heidelberg, Aug. 2018, pp. 155–186.

[16] A. Albertini, T. Duong, S. Gueron, S. Kölbl, A. Luykx, and S. Schmieg, “How to abuse and fix authenticated encryption without key commitment,” in *USENIX Security 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, Aug. 2022, pp. 3291–3308.

[17] “Usage statistics of default protocol https for websites,” https://w3techs.com/technologies/details/ce-httpsdefault, accessed: April 10, 2024.

[18] L. Wang, G. Asharov, R. Pass, T. Ristenpart, and a. shelat, “Blind certificate authorities,” in *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019, pp. 1015–1032.

[19] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, “Poseidon: A new hash function for zero-knowledge proof systems,” in *USENIX Security 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, Aug. 2021, pp. 519–535.

[20] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1,” RFC 2616, Jun. 1999. [Online]. Available: https://www.rfc-editor.org/info/rfc2616

[21] P. Grubbs, J. Lu, and T. Ristenpart, “Message franking via committing authenticated encryption,” in *CRYPTO 2017, Part III*, ser. LNCS, J. Katz and H. Shacham, Eds., vol. 10403. Springer, Heidelberg, Aug. 2017, pp. 66–97.

[22] S. Menda, J. Len, P. Grubbs, and T. Ristenpart, “Context discovery and commitment attacks - how to break CCM, EAX, SIV, and more,” in *EUROCRYPT 2023, Part IV*, ser. LNCS, C. Hazay and M. Stam, Eds., vol. 14007. Springer, Heidelberg, Apr. 2023, pp. 379–407.

[23] M. Bellare and V. T. Hoang, “Efficient schemes for committing authenticated encryption,” in *EUROCRYPT 2022, Part II*, ser. LNCS, O. Dunkelman and S. Dziembowski, Eds., vol. 13276. Springer, Heidelberg, May / Jun. 2022, pp. 845–875.

[24] D. McGrew, “An Interface and Algorithms for Authenticated Encryption,” RFC 5116, Jan. 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5116

[25] T. O. Wiki, “Tls1.3,” 2023. [Online]. Available: https://wiki.openssl.org/index.php/TLS1.3

[26] M. Dworkin, “Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC,” National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST Special Publication (SP) 800-38D, 2007.

[27] Y. Nir and A. Langley, “ChaCha20 and Poly1305 for IETF Protocols,” RFC 7539, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7539

[28] J. Groth, R. Ostrovsky, and A. Sahai, “Perfect non-interactive zero knowledge for NP,” in *EUROCRYPT 2006*, ser. LNCS, S. Vaudenay, Ed., vol. 4004. Springer, Heidelberg, May / Jun. 2006, pp. 339–358.

[29] J. Chan and P. Rogaway, “On committing authenticated-encryption,” in *ESORICS 2022, Part II*, ser. LNCS, V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, Eds., vol. 13555. Springer, Heidelberg, Sep. 2022, pp. 275–294.

[30] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste, “Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS,” in *ACM SIGCOMM 2015*, vol. 45, no. 4. ACM Press, Aug. 2015, pp. 199–212.

[31] K. Bhargavan, I. Boureanu, A. Delignat-Lavaud, P.-A. Fouque, and C. Onete, “A formal treatment of accountable proxying over TLS,” in *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2018, pp. 799–816.

[32] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish, “Zero-knowledge middleboxes,” in *USENIX Security 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, Aug. 2022, pp. 4255–4272.

[33] S. Tan, W. Chen, R. Deng, and R. A. Popa, “MPCAuth: Multi-factor authentication for distributed-trust systems,” in *2023 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2023, pp. 829–847.

Appendix A. Granular Security Framework for Key Commitment

In the work by Menda et al. [22], the concept of the setting of an attack $\Sigma = (\mathsf{ts}, \mathsf{S}, \mathsf{P})$ is defined to capture the difference between key commitment attacks. The context selector S is a distribution that allows the challenger to sample a context to use as the input to the adversary. The target selector ts is a function that picks the info the adversary has access to. The predicate P is a boolean function that specifies the winning condition. While this definition is very abstract at first glance, we will give a formalization of the traditional key commitment property at the end to show how it fits into the two security games proposed in their work.

Context Discovery Attack. The game $\mathsf{CDY}[\Sigma]$ against an adversary $\mathcal{A}$ is defined as:

```
game $\mathsf{CDY}[\Sigma](\mathcal{A})$
    $c \overset{\$}{\leftarrow} \mathsf{S}, a \leftarrow \mathcal{A}(\mathsf{ts}(c))$
    $(C, K, N, A) \leftarrow \mathsf{Merge}(\mathsf{ts}(c), a)$
    $M \leftarrow \mathsf{AEAD.Dec}_K(N, A, C)$
    if $M = \perp$ then
        return false
    end if
    return $\mathsf{P}(C, K, N, A)$
end game
```

To summarize, the game samples some context (e.g. valid ciphertext) from S, feeds the targeted information $\mathsf{ts}(c)$ to the adversary, and then gets the result from the adversary as a. It then puts $(\mathsf{ts}(c), a)$ together to see if it decrypts correctly. If it does, the game returns the predicate on it. Otherwise, it returns false.

Context Commitment Attack. The game $\mathsf{CMT}[\Sigma]$ on some adversary $\mathcal{A}$ is defined as:

```
game $\mathsf{CMT}[\Sigma]$
    $c \overset{\$}{\leftarrow} \mathsf{S}, a \leftarrow \mathcal{A}(\mathsf{ts}(c))$
    $(C, (K_1, N_1, A_1), (K_2, N_2, A_2)) \leftarrow \mathsf{Merge}(c, a)$
    $M_1 \leftarrow \mathsf{AEAD.Dec}_{K_1}(N_1, A_1, C)$
    $M_2 \leftarrow \mathsf{AEAD.Dec}_{K_2}(N_2, A_2, C)$
    if $M_1 = \perp$ or $M_2 = \perp$ then
        return false
    end if
    return $\mathsf{P}((K_1, N_1, A_1), (K_2, N_2, A_2))$
end game
```

To summarize, the game samples some pre-determined context (potentially nothing), feeds it to the adversary, and then asks the adversary to come up with a ciphertext that has two different interpretations. If the adversary succeeds, the game returns the predicate. Otherwise, it returns false.

Context Forgery Attack. Similarly, we can define our context forgery attack game $\mathsf{CFY}[\Sigma]$ using this security framework.

```
game $\mathsf{CFY}[\Sigma]$
    $c \overset{\$}{\leftarrow} \mathsf{S}, a \leftarrow \mathcal{A}(\mathsf{ts}(c))$
    $(C, K_1, N_1, A_1) \leftarrow c, (K_2, N_2, A_2) \leftarrow a$
    $M \leftarrow \mathsf{AEAD.Dec}_{K_2}(N_2, A_2, C)$
    if $M = \perp$ then
        return $\mathsf{false}$
    end if
    return $\mathsf{P}((K_1, N_1, A_1), (K_2, N_2, A_2))$
end game
```

Specification of the Attack Game. We observe that the definitions $\mathsf{CDY}^{\dagger}$, $\mathsf{CFY}^{\dagger}$ and $\mathsf{CMT}^{\dagger}$ in the paper are a specification of the above game definitions under the security framework. Namely,

$$\begin{aligned}\mathsf{CDY}^{\dagger} &= \mathsf{CDY}[\mathsf{S} = (U(k), U(n)), \mathsf{ts} = (C, A), \mathsf{P} = \perp],\\ \mathsf{CFY}^{\dagger} &= \mathsf{CFY}[\mathsf{S} = (U(k), U(n)), \mathsf{ts} = (C, K, N, A),\\ &\quad \mathsf{P} = ((K_1, N_1) \neq (K_2, N_2))],\\ \mathsf{CMT}^{\dagger} &= \mathsf{CFY}[\mathsf{S} = (U(k), U(n)), \mathsf{ts} = (K, N),\\ &\quad \mathsf{P} = ((K_1, N_1) \neq (K_2, N_2))].\end{aligned}$$

Security Implications. Here we formally prove the relationship between CFY, CMT and CDY under the granular security framework.

Theorem A.1 (Relationship between CFY and CMT)**.** *For some AEAD* Π *and any adversary* $\mathcal{A}$ *that wins the* $\mathsf{CFY}[\Sigma]$ *game with advantage* $\Delta^{\mathcal{A}}_{\mathsf{CFY}[\Sigma]}$*, there exists an adversary* $\mathcal{B}$ *that wins the* $\mathsf{CMT}[\Sigma]$ *game with advantage* $\Delta^{\mathcal{B}}_{\mathsf{CMT}[\Sigma]} = \Delta^{\mathcal{A}}_{\mathsf{CFY}[\Sigma]}$.

Proof. We give a straightforward construction of $\mathcal{B}$:

1) $\mathcal{B}$ samples a set of valid parameter (K, N) and encrypts a random plaintext (M, A) to the ciphertext C. It then runs $\mathcal{A}$ on (K, N, M, A, C) and waits for the output.
2) In the case that $\mathcal{A}(K, N, M, A, C)$ outputs, $\mathcal{B}$ delivers the output together with the original tuple (K, N, M, A, C).

We observe that when $\mathcal{A}$ outputs a context that satisfies Σ under CFY, $\mathcal{B}$ also outputs a pair of context that satisfies Σ under CDY. Hence, we get the bound

$$\Delta^{\mathcal{B}}_{\mathsf{CMT}[\Sigma]} = \Delta^{\mathcal{A}}_{\mathsf{CFY}[\Sigma]}.$$

□

Theorem A.2 (Relationship between CFY and CDY)**.** *For some AEAD* Π *and any adversary* $\mathcal{A}$ *that wins the* $\mathsf{CDY}[\Sigma]$ *game with advantage* $\Delta^{\mathcal{A}}_{\mathsf{CDY}[\Sigma]}$*, there exists an adversary* $\mathcal{B}$ *that wins the* $\mathsf{CFY}[\Sigma]$ *game with advantage* $\Delta^{\mathcal{B}}_{\mathsf{CFY}[\Sigma]}$ *such that*

$$\Delta^{\mathcal{B}}_{\mathsf{CFY}[\Sigma]} \geq \frac{1}{2}(1 - \Pr[\mathsf{BadCtx}])\Delta^{\mathcal{A}}_{\mathsf{CDY}[\Sigma]},$$

where BadCtx *is the event that for a randomly sampled* (K, N, M, A)*, the resulting ciphertext* C *can only be decrypted under* $\Sigma(K, N, A)$*.*

Proof. This proof is similar to the proof of the relationship between CDY and CMT by Menda et al. Here we first give a construction of $\mathcal{B}$:

1) $\mathcal{B}$ receives (K, N, M, A, C) as the input. It feeds the necessary input in (K, N, M, A, C), selected by Σ to $\mathcal{A}$ and waits for the output.
2) In the case that $\mathcal{A}(\Sigma(K, N, M, A, C))$ outputs, $\mathcal{B}$ delivers the output.

We then analyze the advantage of $\mathcal{B}$.

In the event of BadCtx, $\mathcal{B}$ will fail since no solution satisfies the constraint of CFY.

On the other hand, if BadCtx does not occur, there exists a set $S = \{\Sigma(K, N, A) | \Sigma(K, N, A) \text{ decrypts } C\}$ with $|S| \geq 2$. Since the input (K, N, M, A, C) is randomly sampled, we can bound the probability of any specific set of parameters

$$\Pr[\Sigma(K, N, A) = (k, n, a) | C = c] = \frac{1}{|S|}.$$

Hence the advantage of $\mathcal{B}$, $\Delta^{\mathcal{B}}_{\mathsf{CFY}[\Sigma] \wedge \neg \mathsf{BadCtx}}$, is

$$\sum_{\Sigma(K,N,A) \in S} \left(1 - \frac{1}{|S|}\right) \Pr[\mathcal{A}(\Sigma(M, A, C)) = \Sigma(K, N, A)].$$

Since

$$\sum_{\Sigma(K,N,A) \in S} \Pr[\mathcal{A}(\Sigma(M, A, C)) = \Sigma(K, N, A)] = \Delta^{\mathcal{A}}_{\mathsf{CDY}[\Sigma]},$$

Therefore, $\Delta^{\mathcal{B}}_{\mathsf{CFY}[\Sigma] \wedge \neg \mathsf{BadCtx}} = (1 - \frac{1}{S})\Delta^{\mathcal{A}}_{\mathsf{CDY}[\Sigma]}$.

We conclude that the overall advantage is

$$\begin{aligned}\Delta^{\mathcal{B}}_{\mathsf{CFY}[\Sigma]} &= (1 - \Pr[\mathsf{BadCtx}])(1 - \frac{1}{S})\Delta^{\mathcal{A}}_{\mathsf{CDY}[\Sigma]}\\ &\geq \frac{1}{2}(1 - \Pr[\mathsf{BadCtx}])\Delta^{\mathcal{A}}_{\mathsf{CDY}[\Sigma]}.\end{aligned}$$

□

Appendix B. Definition of Non-Interactive Zero-Knowledge Proof

Formally, given an NP language $\mathcal{L}$ and its corresponding efficiently decidable binary relation $\mathcal{R}$, we say a statement $x \in \mathcal{L}$ if there exists a witness w such that $(x, w) \in \mathcal{R}$. In NIZK, a prover can generate a proof π to prove the knowledge of w satisfying $(x, w) \in \mathcal{R}$. Through verifying π, a verifier can learn if the prover possesses a valid w without knowing the information about w. Functionality $\mathcal{F}_{\mathsf{NIZK}}$ defined by Groth et al. [28] is shown in Figure 15.

Functionality $\mathcal{F}_{\mathsf{NIZK}}$

Parameters: The non-interactive zero-knowledge functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{NIZK}}$ allows proving of statements in an NP language $\mathcal{L}$. It maintains a set of statement/proof pairs Q, initialized to $\emptyset$. Let $\mathcal{R}$ be an efficiently decidable binary relation for the NP language $\mathcal{L}$.

Functionality:

Prove: Upon receiving $\langle \texttt{PROVE}, sid, x, w \rangle$:

1) If $(x, w) \notin \mathcal{R}$ then return $\langle \texttt{PROOF}, sid, x, \perp \rangle$;
2) Else send $\langle \texttt{PROVE}, sid, x \rangle$ to $\mathcal{A}$ and receive the reply $\langle \texttt{PROOF}, sid, x, \pi \rangle$. Do $Q = Q \cup \{(x, \pi)\}$ and return $\langle \texttt{PROOF}, sid, x, \pi \rangle$;

Verify: Upon receiving$\langle \texttt{VERIFY}, sid, x, \pi \rangle$:

1) If $(x, \pi) \notin Q$ then send $\langle \texttt{VERIFY}, sid, x, \pi \rangle$ to $\mathcal{A}$ and then receive the reply Res;
2) If $\mathsf{Res} = \langle \texttt{WITNESS}, sid, x, \pi, w \rangle \wedge (x, w) \in \mathcal{R}$ then let $Q = Q \cup (x, \pi)$;
3) Return $\langle \texttt{VERIFY}, sid, x, \pi, (x, \pi) \in Q \rangle$.

Figure 15: $\mathcal{F}_{\mathsf{NIZK}}$ functionality

Appendix C. Discussion on Query Restriction

As we described in Section 4, restriction on the user request is not a common requirement in many applications. Here we discuss a few ways that it can be implemented in the protocol. We note that similar to the server response, the proxy protocol is generally not secure over any unrestricted assertion of the user request. However, under very specific application scenarios when restriction on the user request is necessary, it can usually be meaningfully implemented and proven by leveraging one of the methods below.

With TLS 1.3. We observe that by leveraging the TLS 1.3 key update mechanism, we can achieve a partial reveal on the user request that allows the client to prove his request without revealing the secret in the request. Specifically, it can be achieved by:

1) The client establishes a connection with the server through the verifier and completes the handshake.
2) The client transmits non-sensitive information (e.g. HTTP request header) to the server.
3) The client performs a key update.
4) The client transmits sensitive information to the server.
5) The client performs another key update.
6) The client transmits the remainder of the request to the server and receives the response.

In this way, the client can do a full reveal of the keys used in non-sensitive information only to the verifier.

With Variable Padding. Similar to the reasoning in Section 6, the restriction on the user request can be easily implemented when the user request has a variable padding, such as the scenario with HTTPS when the URL is of sufficient length.

With Hybrid. In many common home-brew protocols (such as Facebook Messenger), the user's access is established via transmitting an access token. Assuming that there are only polynomially many tokens available at a time and they are generated by a random oracle, we can assert that a computationally bounded adversary can only access a token not belonging to him with negligible probability via a hybrid argument, and thus the system can be effectively treated as only having tokens belonging to the adversary. Although this line of proof looks more like reasoning on system security, it restricts the adversary's capability of finding a valid token that he does not own, thus also restricting him from sending/forging a token that does not belong to him.