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

Zhongtang Luo
Purdue University, United States

Yanxue Jia
Purdue University, United States

Yaobin Shen
Xiamen University, China

Aniket Kate
Purdue University, United States
Supra Research, United States

Abstract

TLS allows a client to securely obtain data from a server, but does not allow the client to offer the data provenance to an external node. TLS oracle protocols are used to solve the problem. Specifically, the verifier node, as an external node, is convinced that the data is indeed coming from a pre-defined TLS server, while remaining unable to access the client's credentials (e.g., password). Previous TLS oracle protocols such as DECO (CCS 2020) enforced the communication pattern of server-client-verifier and utilized 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 and the verifier. Most recently, some works have proposed 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. Nevertheless, these works still rely on heavy two-party secure computations or zero-knowledge proofs. In this work, we push the proxy model to the extreme, where the verifier only needs to forward messages without performing any other heavy computational operations when only the credentials should be protected and the data retrieved from the server could be open to the verifier. Surprisingly, we prove that the thorough proxy model is enough to guarantee security in some common scenarios, allowing a saving of 60–90% in running time under common scenarios.

We first formalize the proxy-based Oracle protocol and functionality 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 that data integrity can also be guaranteed as long as the underlying Authenticated Encryption with Associated Data (AEAD) satisfies context unforgeability. We further show that ChaCha20-Poly1305 satisfies the concept while AES-GCM does not.

2012 ACM Subject Classification Security and privacy → Block and stream ciphers; Security and privacy → Security protocols

Keywords and phrases Oracle, TLS, AEAD, Key Commitment

Digital Object Identifier 10.4230/LIPIcs.AFT.2025.4

Related Version *Full Version*: `https://ia.cr/2024/733`

Funding This work is supported partially by the National Science Foundation under grant CNS-1846316 and a gift from Supra Labs.
Yaobin Shen: Yaobin Shen is supported by the National Key Research and Development Program of China (2024YFB4504800), and the National Natural Science Foundation of China (62402404).

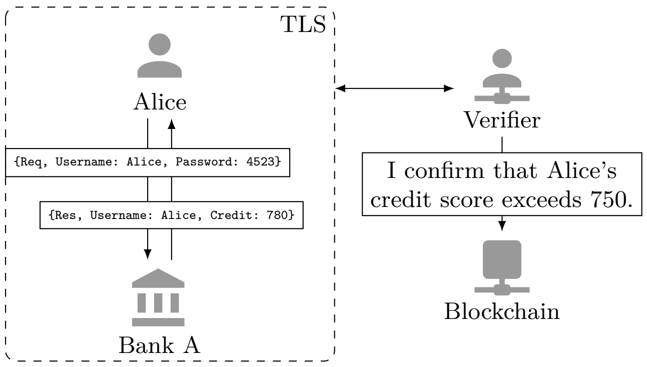

TLS
Alice
{Req, Username: Alice, Password: 4523}
{Res, Username: Alice, Credit: 780}
Bank A
Verifier
I confirm that Alice's credit score exceeds 750.
Blockchain

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

Acknowledgements We would like to thank Madhavan Malolan and Kirill Kutsenok from the Reclaim Protocol for helpful discussions and suggestions.

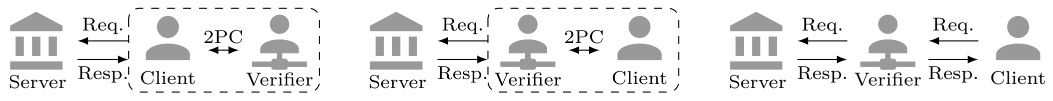
1 Introduction

Transport Layer Security (TLS) [24, 25] 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* [5, 7] 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 security and privacy, as the verifier needs to obtain the client's credentials. 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 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 if "Alice's credit score exceeds 750". Alice uses her password to request the credit score 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 credit score in bank A exceeds 750. If so, the verifier submits the confirmation information to the blockchain, and then the rental contract is activated.

For some applications, only request should be hidden from the verifier, and the response could be open to the verifier. As in the above example, Alice may be willing to open `{Res, Username: Alice, Credit: 780}` to the verifier; however, she does not want to reveal the password in the request to the verifier, as the verifier can obtain more sensitive information using the password. On the other hand, there are also some applications where protecting the



(a) 2PC-based TLS Oracle [33, 31, 9, 6]. **(b)** Proxy-but-2PC TLS Oracle [33, 16]. **(c)** Proxy-based TLS Oracle (this work).

Figure 2 Design frameworks in the previous works use a 2PC protocol to prevent the client from forging a message. In contrast, in our work, the client and the server interact with each other as in TLS, but the messages between them are forwarded by the verifier.

response is also necessary, e.g., the response includes bank statements. An oracle protocol designed for the former scenario can be adapted to the latter by incorporating zero-knowledge proofs, allowing the client to prove that the data in response satisfies a specific predicate without revealing the data to the verifier.

As bringing clients' private data securely and selectively to blockchains is a problem of immense interest, a few academic [33, 32, 26, 16, 31, 9] and industrial [4, 22] efforts in this direction are already available. Earlier works typically involve modifying the TLS server-side code [26] or utilizing trusted hardware [32]. While the approaches remain an interesting theoretical probability, typical servers 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. [33] 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 [33], 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 [31, 9, 6] also follow the model.

In addition, Zhang et al. [33] 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. Xie et al. [31] proposed a solution in this model that utilizes a garble-then-prove scheme.

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?"*

Proxy-based TLS Oracle. To answer this question, in this work, we consider a pure proxy-based TLS oracle as shown in Figure 2c, 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 AEAD (Authenticated Encryption with Associated Data) 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% [29]. 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. 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 Figure 11. On the other hand, in Section 5, we find that without the key commitment, an adversarial client can potentially 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 [10, 1], 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, 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 [23], 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.

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.

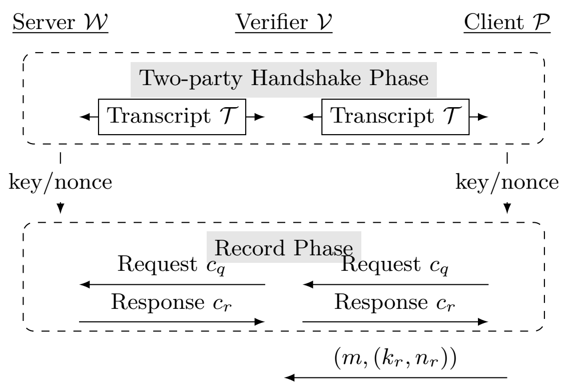


Figure 3 Our Proxy-based Oracle Protocol. The server and the client perform as in TLS, but the messages are forwarded by the verifier. Finally, the client sends a message m, a key/nonce pair (k_r, n_r) to the verifier, such that the verifier can verify that c_r could be decrypted to the message m using the key/nonce pair (k_r, n_r). When the client does not want to open the entire message m to the verifier, the client can use a zero-knowledge proof protocol to generate 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 proof for data is orthogonal to our work. The detailed specification is defined in Π_{Proxy} (see Figure 7).

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). Typically, there are two cases: one is that the message m can be open to the verifier, called "public mode", and the other is that the client does not want to leak the entire m to the verifier, called "private mode". The protocol for the public mode can be transformed to that for the private mode by using a zero-knowledge proof with the opening information in the public mode as witness. Therefore, for simplicity, here we focus on the public mode, and we also show the private mode in Section 4.

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 [28]. Based on the above assumptions, we design a proxy-based oracle protocol as illustrated in Figure 3 and obtain a series of theoretical results summarized in Figure 4. Next, we will detail how we obtain these results.

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 3 (ignoring the verifier), 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

[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.

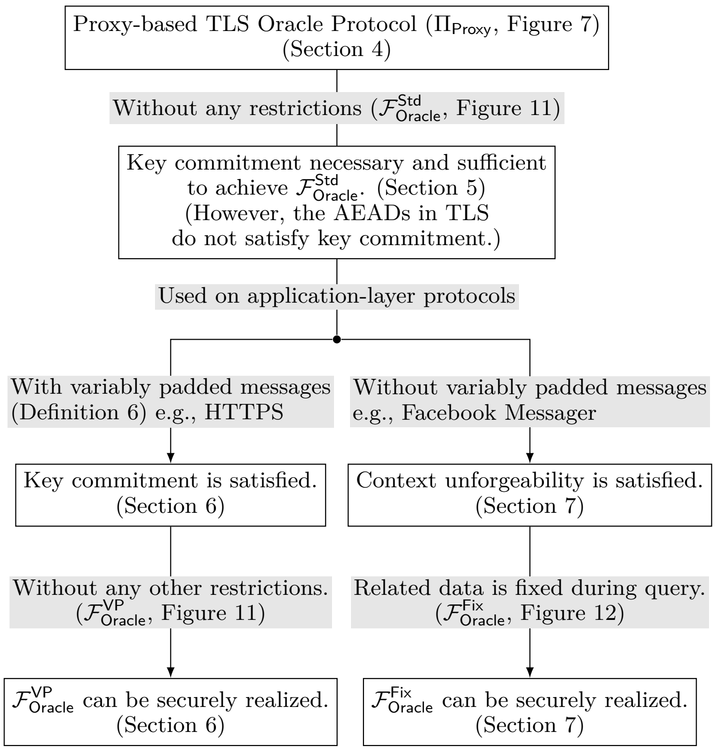

Proxy-based TLS Oracle Protocol (Π_{Proxy}, Figure 7) (Section 4)
Without any restrictions ($\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$, Figure 11)
Key commitment necessary and sufficient to achieve $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$. (Section 5) (However, the AEADs in TLS do not satisfy key commitment.)
Used on application-layer protocols
With variably padded messages (Definition 6) e.g., HTTPS
Without variably padded messages e.g., Facebook Messager
Key commitment is satisfied. (Section 6)
Context unforgeability is satisfied. (Section 7)
Without any other restrictions. ($\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$, Figure 11)
Related data is fixed during query. ($\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$, Figure 12)
$\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$ can be securely realized. (Section 6)
$\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ can be securely realized. (Section 7)

Figure 4 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 $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$.

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 [33, 9, 31, 6, 16] 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?*

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 [33, 9, 31, 6, 16] 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 3. 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 provide $(m, (k_r, n_r))$ to verifier $\mathcal{V}$ to prove the following three statements: he holds a key/nonce pair (k_r, n_r) that can decrypt c_r to a message m; the message m satisfies a preset requirement; the key/nonce pair (k_r, n_r) is derived from the transcript $\mathcal{T}$.

Obviously, once obtaining $(m, (k_r, n_r))$, the verifier can verify the first two statements directly[2]. However, the verifier cannot verify the third statement, since it involves private information (namely, the secret used in Diffie-Hellman Handshake) held by the client (please see Figure 6 for more details). Note that if the client provide the private information used in Diffie-Hellman handshake process to the verifier, the verifier can know all the keys and nonces, and thus the verifier can decrypt the request ciphertext (c_q in Figure 3) to learn the credential.

Using a zero-knowledge proof to prove the third statement is a straightforward way. However, 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 [13]. A concurrent and independent work by Ernstberger [12] have tried to reduce the cost for proving the statement. However, their results showed that the proof for this statement still occupies the cost 90% in the public mode and the cost 60% in the private mode (see Table 1). Therefore, to further reduce the 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 [10, 1], the AEAD schemes used by TLS do not satisfy the key commitment property (see Lemma 3); 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, since client $\mathcal{P}$ obtains the key/nonce pair (k, n) as soon as 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') (see Section 5).

[2] Given that HTTP is a stateless protocol, to avoid using the same key the verifier obtained for future rounds of interaction, the client can terminate the current connection with the server and handshake again to restart one. For other protocols, we note that TLS 1.3 supports a key update mechanism (see Figure 6) that allows us to refresh the key/nonce pair for subsequent communications without needing another handshake.

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 (see Section 6).

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 *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. We 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 5. Interestingly, combined with the previous properties, context commitment[3] [10, 15, 18] and context undiscoverability [18], the hierarchical security of AEAD precisely corresponds to the hierarchical security of hash functions.

3 Preliminary

We briefly outline the relevant background knowledge on TLS, the definition and security properties of Authenticated Encryption with Associated Data (AEAD).

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 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 [24].

[3] The works [10, 15] first proposed the key commitment problem. Then Bellare et al. [2] extended to committing nonce and associated data, and Menda et al. [18] summarized them as context commitment.

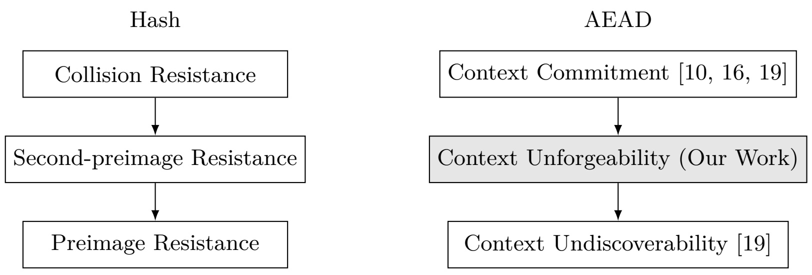

Hash
Collision Resistance
Second-preimage Resistance
Preimage Resistance
AEAD
Context Commitment [10, 16, 19]
Context Unforgeability (Our Work)
Context Undiscoverability [19]

Figure 5 Hierarchical Security of AEAD. Both hash and AEAD can be abstracted as a map: $x \to y$; 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 $x \to y$, 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.

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 6.

Authenticated Encryption with Associated Data. Nonce-based authenticated encryption with associated data (AEAD) schemes [17] are employed in TLS to ensure data confidentiality and integrity. An AEAD scheme consists of the following four algorithms $(\mathsf{AEAD.Setup}, \mathsf{AEAD.Gen}, \mathsf{AEAD.Enc}, \mathsf{AEAD.Dec})$. We refer to Full Version's Appendix A for a full definition.

Cipher Suites in TLS. TLS supports a handful number of cipher suites [24], with AES-GCM and ChaCha20-Poly1305 being the most popular and enabled in OpenSSL by default [30]. 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 $(p_1, p_2, \ldots, p_n)$ and $(a_1, a_2, \ldots, a_m)$, which is then processed by the algorithm to produce the ciphertext.

<u>AES-GCM.</u> 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 $c_i = p_i + E_k(n+i)$, where E is the AES block cipher.

AES-GCM also ensures authenticity by using an authentication tag $t = E_k(n) + \sum_{i=1}^{m+n+1} s_i E_k(0)^{m+n+2-i}$. 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 [11].

<u>ChaCha20-Poly1305.</u> ChaCha20-Poly1305 is an alternative to AES-GCM. In ChaCha20-Poly1305, encryption is done similarly to AES-GCM: $c_i = p_i + H_k(n+i)$, where H is the ChaCha20 stream cipher.

[4] 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 [17], IV is limited to 96 bits, while the nonce used in the encryption is derived from IV by padding and can be longer.

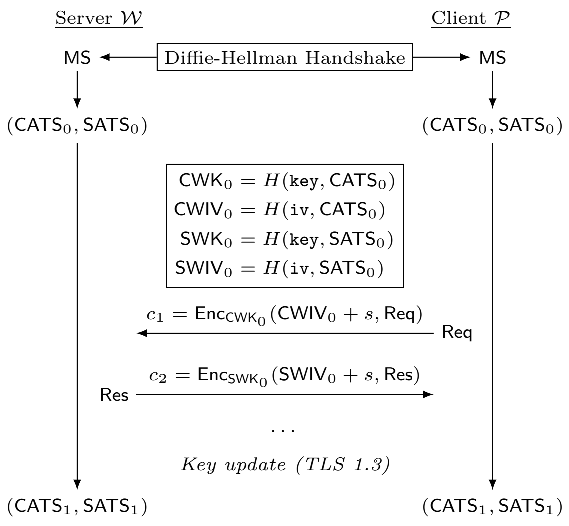

Server $\mathcal{W}$
Client $\mathcal{P}$
MS
Diffie-Hellman Handshake
MS
$(\mathsf{CATS}_0, \mathsf{SATS}_0)$
$(\mathsf{CATS}_0, \mathsf{SATS}_0)$
$\mathsf{CWK}_0 = H(\mathtt{key}, \mathsf{CATS}_0)$
$\mathsf{CWIV}_0 = H(\mathtt{iv}, \mathsf{CATS}_0)$
$\mathsf{SWK}_0 = H(\mathtt{key}, \mathsf{SATS}_0)$
$\mathsf{SWIV}_0 = H(\mathtt{iv}, \mathsf{SATS}_0)$
$c_1 = \mathsf{Enc}_{\mathsf{CWK}_0}(\mathsf{CWIV}_0 + s, \mathsf{Req})$
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 6 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[4] 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.

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 $t = s + \sum_{i=1}^{m+n+1} s_i r^{m+n+2-i}$ where $s = (a, c, m\|n\|0^8)$ is the concatenation of associated data, ciphertext and their length 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 [21].

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 6) 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 6.

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. [18]. 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 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 decrypt 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 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* $\mathcal{A}$.
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. [18] 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 Full Version's Appendix B.

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. 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 discuss the relation between the three games in Section 7.2.

Key Commitment Attacks. The two major cipher suites in TLS, AES-GCM and ChaCha20-Poly1305, are not key committing under this definition. The concrete attack is well-studied and presented in multiple works [1, 18, 10].

▶ **Lemma 3** (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* $2^{-32}q$.

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). A boolean value Mode_p indicates whether the execution is in a private mode.

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 $c \leftarrow \mathsf{AEAD.Enc}_{\mathsf{CWK}_i}(\mathsf{CWIV}_i, a, Q)$, where i is initialized as 0 and increases by 1 with each key update (described below), and a is the associated data in 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 $c' = \mathsf{AEAD.Enc}_{\mathsf{SWK}_i}(\mathsf{SWIV}_i, a, R)$ 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 $m = \mathsf{AEAD.Dec}_{\mathsf{SWK}_i}(\mathsf{SWIV}_i, a', c')$, 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:
  - $\mathsf{Mode}_p = 0$: m does not involve private information:
    * The client $\mathcal{P}$ sends m and $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ to verifier $\mathcal{V}$;
    * The verifier $\mathcal{V}$ checks if $m = \mathsf{AEAD.Dec}_{\mathsf{SWK}_i}(\mathsf{SWIV}_i, a', c')$ and $\mathsf{P}(\tilde{m}) = 1$ where $\tilde{m}$ is the pertinent information extracted from m, output 1 if both conditions hold true, and 0 otherwise.
    * The client $\mathcal{P}$ and server $\mathcal{W}$ update $(\mathsf{CATS}_i, \mathsf{SATS}_i)$ to $(\mathsf{CATS}_{i+1}, \mathsf{SATS}_{i+1})$ and further compute $(\mathsf{CWK}_{i+1}, \mathsf{CWIV}_{i+1})$ and $(\mathsf{SWK}_{i+1}, \mathsf{SWIV}_{i+1})$ using the key update mechanism (see Section 3.1 for more details).
  - $\mathsf{Mode}_p = 1$: m involves private information:
    * The client $\mathcal{P}$ generates a proof π' by invoking $\mathcal{F}_{\mathsf{NIZK}}$ (see Full Version's Appendix A.2) [14] to prove that he knows m, $\tilde{m}$ and $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ such that $m = \mathsf{AEAD.Dec}_{\mathsf{SWK}_i}(\mathsf{SWIV}_i, a', c')$ and $\tilde{m}$ is extracted from m, and sends π';
    * If π' is verified successfully, verifier $\mathcal{V}$ outputs 1, otherwise 0.

Figure 7 Proxy-based Oracle Protocol.

4 Proxy-based Oracle Protocol

Our proxy-based oracle protocol Π_{Proxy} (shown in Figure 7) 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, and define the functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ in Figure 11 in Full Version's Appendix C.1.

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 $(\mathsf{CWK}_i, \mathsf{CWIV}_i)$ and the response ciphertext c' under $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$. The verifier $\mathcal{V}$ is responsible for relaying the two ciphertexts.

In the prove phase, client $\mathcal{P}$ can choose the public mode (i.e., $\mathsf{Mode}_p = 0$) or the private mode (i.e., $\mathsf{Mode}_p = 1$). In the public mode, client $\mathcal{P}$ opens the response m and the server write key/IV pair $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ to verifier $\mathcal{V}$. Then, verifier $\mathcal{V}$ checks if the response ciphertext c' can be decrypted to m using $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ and $\mathsf{P}(\tilde{m}) = 1$ where $\tilde{m}$ is the pertinent message extracted from m. If the verification is successful, verifier $\mathcal{V}$ outputs 1, otherwise, outputs 0. Since that $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ is opened and cannot be used for the subsequent communications, client $\mathcal{P}$ informs server $\mathcal{W}$ to update $(\mathsf{CWK}_i, \mathsf{CWIV}_i)$ and $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ to $(\mathsf{CWK}_{i+1}, \mathsf{CWIV}_{i+1})$ and $(\mathsf{SWK}_{i+1}, \mathsf{SWIV}_{i+1})$ by using the key update mechanism (see Section 3.1 for more details). In the private mode, the client $\mathcal{P}$ invokes $\mathcal{F}_{\mathsf{NIZK}}$ to generate a proof π' proving that he holds a server write key/IV pair $(\mathsf{SWK}_i, \mathsf{SWIV}_i)$ 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.

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 [33, 16], 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 4.** *When the underlying AEAD is vulnerable to key-committing attacks (see Lemma 3), there exists some server $\mathcal{W}$ and predicate $\mathsf{P}(m)$ such that the protocol Π_{Proxy} does not realize the functionality $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ (see Figure 11 in Full Version's Appendix C.1).*

Proof. Recall that in key-committing attack (see Definition 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 [25, 24], 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:

$\mathcal{A}$ starts the protocol. It starts handshaking with the server $\mathcal{W}$ through verifier $\mathcal{V}$ to obtain (k, n).

$\mathcal{A}$ 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).

$\mathcal{A}$ modifies the server-side data to m via side-channel methods, such that the verifier $\mathcal{V}$ receives and records the response ciphertext c.

$\mathcal{A}$ uses (k', n') to convince the verifier $\mathcal{V}$ that m' is the plaintext. ◀

```
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 8 An HTTP response header from an Apache server. The first two lines (status code and date) can be used as a generic variable padding for all HTTP responses.

A real-life attack scenario can also be found in the famous Facebook attack [10]. 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 decrypt 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.** *Given a secure AEAD with key commitment security (see Lemma 3), protocol* Π_{Proxy} *shown in Figure 7 can securely realize the oracle functionality* $\mathcal{F}^{\mathsf{Std}}_{\mathsf{Oracle}}$ *(see Figure 11 in Full Version's Appendix C.1) in the* $\mathcal{F}_{\mathsf{NIZK}}$*-hybrid model, against a static active adversary who can corrupt client* $\mathcal{P}$ *or verifier* $\mathcal{V}$*.*

We refer the readers to Full Version's Appendix E for proof details.

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 8 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. [1] 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** (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 8. 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 7.** *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 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; meanwhile, 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 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 our statement for AES-GCM.

▶ **Theorem 8** (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.*

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, the following equation set holds:

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

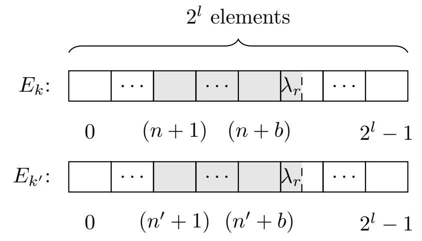


Figure 9 A representation of E_k and $E_{k'}$ as permutations. The shaded part in $E_{k'}$ is uniquely determined from the shaded part in E_k.

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

Without considering the above requirement, $E_{k'}$ could be randomly chosen from $(2^l)!$ permutations. Hence the probability of BAD happening is at most $\Pr(\mathsf{BAD}) \leq \frac{2^{l-\lambda_r}(2^l-b-1)!}{(2^l)!} < \frac{2^{l-\lambda_r}}{(2^l-b)^{b+1}} = \frac{1}{2^\lambda}\left(\frac{2^l}{2^l-b}\right)^{b+1}$.

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

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 $\epsilon \leq 2^{-221}q^2$. Note that $\left(\frac{2^l}{2^l-b}\right)^{b+1} \approx 1$.

Therefore, the protocol is secure on HTTPS servers.

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

▶ **Theorem 9** (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 $\Pr(\mathsf{BAD}) \leq \frac{2^{l-\lambda_r}(2^l)^{2^l-b-1}}{(2^l)^{2^l}} = \frac{2^{l-\lambda_r}}{(2^l)^{b+1}} = \frac{1}{2^\lambda}$. Therefore, the overall probability is bounded by $\Delta^{\mathcal{A}} < \binom{q}{2} \cdot 2^{2v} \cdot |S|^2 \cdot \Pr(\mathsf{BAD}) < \frac{q^2|S|^2}{2^{\lambda-2v+1}}$. ◀

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 $\epsilon \leq 2^{-221}q^2$.

An application of Theorem 8 and Theorem 9 gives us the following desired security property.

▶ **Theorem 10.** *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 7 can securely realize the oracle functionality $\mathcal{F}^{\mathsf{VP}}_{\mathsf{Oracle}}$ defined in Figure 11 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.

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 [10] 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 12 in Full Version's Appendix C.2. 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.

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 1) considered by Menda et al. [18] 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. [18], 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, 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 11** (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 [10], 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 16), we can use ideas that Menda et al. [18] proposed for the CDY attack to give an attack for CFY.

▶ **Theorem 12.** *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 where $t = E_k(n) + E_k(0)^2c + E_k(0)$. Now suppose we sample some other key $k' \neq k$, then we have the following equation, given the fixed tag t and ciphertext block c that $t = E_{k'}(n') + E_{k'}(0)^2c + E_{k'}(0)$.

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. [18]. Moreover, many works [18, 10] 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 13.** *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, $\Pr(H(k', n') = h)$ 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 $\Pr(\mathsf{BAD}) = \sum_{i \in \mathbb{Z}/2^{128}\mathbb{Z}} \Pr(s' - s = i) \Pr(\Delta p(r') = i) = \frac{1}{2^{128}} \sum_i \Pr(\Delta p(r') = i) = \frac{1}{2^{128}}$. Since the attacker makes q queries, we bound the probability 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 14.** *Given a secure AEAD with context unforgeability security under* $\mathsf{CFY}^{\dagger}$*, the protocol* Π_{Proxy} *shown in Figure 7 can securely realize the functionality* $\mathcal{F}^{\mathsf{Fix}}_{\mathsf{Oracle}}$ *in Figure 12 in the* $\mathcal{F}_{\mathsf{NIZK}}$*-hybrid model, against a static active adversary who can corrupt client* $\mathcal{P}$ *or verifier* $\mathcal{V}$*.*

We refer the readers to Full Version's Appendix F for the full proof.

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 Full Version's Appendix B.

▶ **Corollary 15.** *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 16.** *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 Evaluation

Due to the fact that the related works' implementations are close-sourced, including DECO [33], the work by Xie et al. [31], Janus [16], ORIGO[5] [12], we conduct an empirical survey to outline the cost that can be saved by our technique in Table 1, using numbers reported in these papers. We split the result from these papers into the following three parts, and our result in this paper can eliminate the cost in key-related assurance:

[5] The end-to-end codebase in ORIGO (`https://github.com/opex-research/tls-oracle-demo`) is open-sourced, but the zero-knowledge component (`https://github.com/opex-research/tls-zkp`) is not.

Table 1 Time comparison between various related works, using AES-GCM as the cipher suite. Shaded parts represent the cost we can save with our result. For reference, ORIGO [12] measured the TLS overhead to be 1.26 s under 1 Gbps WAN network. DECO tested query and record layer assurance separately, but did not report response size.

Work	Scenario	Record Type	Resp. Size (B)	Key-Related Off. (s)	Key-Related On. (s)	Record Off. (s)	Record On. (s)
ORIGO [12] (Groth16)	Circuit[1]	Open[2] (32 B)	1429	22.47	1.08	10.55	0.61
ORIGO [12] (Plonk)	Circuit	Open (32 B)	1429	27.51	14.84	13.99	7.67
Janus [16]	LAN	Open (256 B)	2048	3.94	4.79	1.13	2.08
Xie et al. [31]	LAN	–	1024	–	0.61[4]	–	
DECO [33] (512 B Query)	LAN	–	–	2.28	0.42	–	
DECO [33] (512 B Query)	WAN (100 Mbps)	–	–	22.81	5.20	–	
DECO [33] (1 KB Query)	LAN	–	–	2.50	0.47	–	
DECO [33] (1 KB Query)	WAN (100 Mbps)	–	–	24.59	7.42	–	
DECO [33]	Circuit	Binary Options[3]	–	–		–	12.97
DECO [33]	Circuit	Age Proof	–	–		–	3.67
DECO [33]	Circuit	Price Discrimination	–	–		–	12.68

1. Local computation of proof circuit.
2. Selective opening of a substring of the response.
3. DECO designed some scenarios where the prover proves a specific statement.
4. Xie et al.'s work only reported the online computation cost of 2-party computations.

1. TLS proxy overhead: cost associated with TLS proxying and network cost.
2. Key-related assurance: cost associated with ensuring key commitment. Some prior work [33, 31, 16] utilizes 2PC protocols to generate query since the key is split between the client and the oracle. Here, we consider the processes of three-party handshake and relevant 2PC to be in this category, since our result does not need to split the key, and eliminates these costs.
3. Record layer assurance: cost on proving statements in the record layer.

Based on empirical data, in the simple case where everything that needs to be hidden resides in the request and the record layer assurance does not need to be deployed, we can achieve around a 90% saving in running time by eliminating key-related assurance. On the other hand, if we consider typical record layer assurance protocols deployed by these works, we can achieve around a 60% saving in performance, depending on the complexity of the record layer circuit.

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. [10]. Commitment security on AEAD has received a lot of research since around that time [15, 8]. Menda et al. provided a generalization and analysis on the topic [18], which we leverage in this paper. Albertini et al. analyzed several possible ways to fix popular AEAD schemes [1]. Our work builds and extends on the framework by Menda et al. [18] 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 [19, 3, 14, 27]. 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 [32, 26]. DECO by Zhang et al. [33] 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. [16] 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. [31] uses the garble-then-prove technique instead. DIDO by Chan et al. [9] proposes further optimization based on TLS 1.3. ORIGO [12], an independent and concurrent work of ours, observed that the ZK proof for key derivation can be optimized.

Meanwhile, there is also significant industry effort on this topic. The recent Reclaim Protocol [22] 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 and Discussion

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 prove that the proxy protocol is secure under an application layer protocol with a variable padding, such as HTTPS. We further explore the scenario where the application layer does not provide variable padding. We show that if the adversary cannot tamper with the response after establishing the connection, context unforgeability (CFY) of the underlying AEAD is sufficient to demonstrate security. We analyze the cipher suites in TLS, and find that AES-GCM does not satisfy the property, but ChaCha20-Poly1305 does.

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, since HTTP is a stateless protocol, the client can terminate the connection and handshake again to start a new one for future rounds. Moreover, we note that TLS 1.3 supports a key update mechanism (see Figure 6) 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 4), 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.

BGP Attacks and Proxy-Server Connection Hijacking. In the main body of the paper, we assumed that the verifier has a trusted connection to the TLS server. Indeed, the problem is not meaningful if the server is not trusted or if there is no trusted way to access the server, since there would not exist any root of trust under this scenario.

Here, we note that if the client can adaptively corrupt the connection between the verifier and the TLS server, it can first establish the connection with the real server and obtain the HTTPS certificate, and then hijack the connection to transmit the message with the master secret it knows. This attack works even if the verifier pins the server certificate in advance, but requires the client to adaptively corrupt the connection.

References

1 Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 3291–3308. USENIX Association, August 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/albertini.

2 Mihir Bellare and Viet Tung Hoang. Efficient schemes for committing authenticated encryption. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 845–875. Springer, Heidelberg, May / June 2022. doi:10.1007/978-3-031-07085-3_29.

3 Karthikeyan Bhargavan, Ioana Boureanu, Antoine Delignat-Lavaud, Pierre-Alain Fouque, and Cristina Onete. A formal treatment of accountable proxying over TLS. In *2018 IEEE Symposium on Security and Privacy*, pages 799–816. IEEE Computer Society Press, May 2018. doi:10.1109/SP.2018.00021.

4 Lorenz Breidenbach, Christian Cachin, Benedict Chan, Alex Coventry, Steve Ellis, Ari Juels, Farinaz Koushanfar, Andrew Miller, Brendan Magauran, Daniel Moroz, et al. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks, 2021. URL: https://chain.link/.

5 Vitalik 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.

6 Sofía Celi, Alex Davidson, Hamed Haddadi, Gonçalo Pestana, and Joe Rowell. Distefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more. Cryptology ePrint Archive, Paper 2023/1063, 2023. URL: https://eprint.iacr.org/2023/1063.

7 Chainlink Foundation. What is the blockchain oracle problem? https://blog.chain.link/what-is-the-blockchain-oracle-problem/. Accessed: March 28, 2024.

8 John Chan and Phillip Rogaway. On committing authenticated-encryption. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *ESORICS 2022, Part II*, volume 13555 of *LNCS*, pages 275–294. Springer, Heidelberg, September 2022. doi:10.1007/978-3-031-17146-8_14.

9 Kwan Yin Chan, Handong Cui, and Tsz Hon Yuen. DIDO: data provenance from restricted TLS 1.3 websites. In *Information Security Practice and Experience*, pages 154–169. Springer Nature, 2023. doi:10.1007/978-981-99-7032-2_10.

10 Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 155–186. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96884-1_6.

11 Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. Technical Report NIST Special Publication (SP) 800-38D, National Institute of Standards and Technology, Gaithersburg, MD, 2007. doi:10.6028/NIST.SP.800-38D.

12 Jens Ernstberger, Jan Lauinger, Yinnan Wu, Arthur Gervais, and Sebastian Steinhorst. ORIGO: Proving provenance of sensitive data with constant communication. Cryptology ePrint Archive, Paper 2024/447, 2024. URL: `https://eprint.iacr.org/2024/447`.

13 Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, August 2021. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/grassi`.

14 Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 4255–4272. USENIX Association, August 2022. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/grubbs`.

15 Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 66–97. Springer, Heidelberg, August 2017. `doi:10.1007/978-3-319-63697-9_3`.

16 Jan Lauinger, Jens Ernstberger, Andreas Finkenzeller, and Sebastian Steinhorst. Janus: Fast privacy-preserving data provenance for TLS 1.3. Cryptology ePrint Archive, Report 2023/1377, 2023. URL: `https://eprint.iacr.org/2023/1377`.

17 David McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, January 2008. `doi:10.17487/RFC5116`.

18 Sanketh Menda, Julia Len, Paul Grubbs, and Thomas Ristenpart. Context discovery and commitment attacks - how to break CCM, EAX, SIV, and more. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part IV*, volume 14007 of *LNCS*, pages 379–407. Springer, Heidelberg, April 2023. `doi:10.1007/978-3-031-30634-1_13`.

19 David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. In *ACM SIGCOMM 2015*, volume 45, pages 199–212. ACM Press, August 2015. `doi:10.1145/2829988.2787482`.

20 Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999. `doi:10.17487/RFC2616`.

21 Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015. `doi:10.17487/RFC7539`.

22 Reclaim Protocol. Reclaim protocol: Claiming and managing self-sovereign credentials, 2023. URL: `https://www.reclaimprotocol.org/`.

23 Q-Success. Usage statistics of default protocol https for websites. `https://w3techs.com/technologies/details/ce-httpsdefault`. Accessed: April 10, 2024.

24 Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. `doi:10.17487/RFC8446`.

25 Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008. `doi:10.17487/RFC5246`.

26 Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, and Srdjan Capkun. TLS-N: Non-repudiation over TLS enablign ubiquitous content signing. In *NDSS 2018*. The Internet Society, February 2018. URL: `https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-4_Ritzdorf_paper.pdf`.

27 Sijun Tan, Weikeng Chen, Ryan Deng, and Raluca Ada Popa. MPCAuth: Multi-factor authentication for distributed-trust systems. In *2023 IEEE Symposium on Security and Privacy*, pages 829–847. IEEE Computer Society Press, May 2023. `doi:10.1109/SP46215.2023.10179481`.

28 Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and abhi shelat. Blind certificate authorities. In *2019 IEEE Symposium on Security and Privacy*, pages 1015–1032. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00007.

29 David Warburton. The 2021 TLS telemetry report. https://www.f5.com/labs/articles/threat-intelligence/the-2021-tls-telemetry-report. Accessed: April 10, 2024.

30 The OpenSSL Wiki. Tls1.3, 2023. URL: https://wiki.openssl.org/index.php/TLS1.3.

31 Xiang Xie, Kang Yang, Xiao Wang, and Yu Yu. Lightweight authentication of web data via garble-then-prove. Cryptology ePrint Archive, Report 2023/964, 2023. URL: https://eprint.iacr.org/2023/964.

32 Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 270–282. ACM Press, October 2016. doi:10.1145/2976749.2978326.

33 Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1919–1938. ACM Press, November 2020. doi:10.1145/3372297.3417239.