DECO: Liberating Web Data Using Decentralized Oracles for TLS

Fan Zhang*
Cornell Tech

Deepak Maram*
Cornell Tech

Harjasleen Malvai*
Cornell University

Steven Goldfeder*
Cornell Tech

Ari Juels*
Cornell Tech

ABSTRACT

Thanks to the widespread deployment of TLS, users can access private data over channels with end-to-end confidentiality and integrity. What they cannot do, however, is prove to third parties the *provenance* of such data, i.e., that it genuinely came from a particular website. Existing approaches either introduce undesirable trust assumptions or require server-side modifications.

Users' private data is thus locked up at its point of origin. Users cannot export data in an integrity-protected way to other applications without help and permission from the current data holder.

We propose DECO (short for decentralized oracle) to address the above problems. DECO allows users to prove that a piece of data accessed via TLS came from a particular website and optionally prove statements about such data in zero-knowledge, keeping the data itself secret. DECO is the first such system that works without trusted hardware or server-side modifications.

DECO can liberate private data from centralized web-service silos, making it accessible to a rich spectrum of applications. To demonstrate the power of DECO, we implement three applications that are hard to achieve without it: a private financial instrument using smart contracts, converting legacy credentials to anonymous credentials, and verifiable claims against price discrimination.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; **Security protocols**.

KEYWORDS

Oracles; Blockchains; Smart Contracts; Transport Layer Security

ACM Reference Format:
Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. 2020. DECO: Liberating Web Data Using Decentralized Oracles for TLS. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3372297.3417239

*Also affiliated with IC3, The Initiative for CryptoCurrencies & Contracts.

CCS '20, November 9–13,2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417239

1 INTRODUCTION

TLS is a powerful, widely deployed protocol that allows users to access web data over confidential, integrity-protected channels. But TLS has a serious limitation: it doesn't allow a user to prove to third parties that a piece of data she has accessed authentically came from a particular website. As a result, data use is often restricted to its point of origin, curtailing data portability by users, a right acknowledged by recent regulations such as GDPR [8].

Specifically, when a user accesses data online via TLS, she cannot securely *export* it, without help (hence permission) from the current data holder. Vast quantities of private data are thus intentionally or unintentionally locked up in the "deep web"—the part of the web that isn't publicly accessible.

To understand the problem, suppose Alice wants to prove to Bob that she's over 18. Currently, age verification services [1] require users to upload IDs and detailed personal information, which raises privacy concerns. But various websites, such as company payroll records or DMV websites, in principle store and serve verified birth dates. Alice could send a screenshot of her birth date from such a site, but this is easily forged. And even if the screenshot could somehow be proven authentic, it would leak information—revealing her exact birth date, not just that she's over 18.

Proposed to prove provenance of online data to smart contracts, *oracles* are a step towards exporting TLS-protected data to other systems with provenance and integrity assurances. Existing schemes, however, have serious limitations. They either only work with deprecated TLS versions and offer no privacy from the oracle (e.g., TLSNotary [7]) or rely on trusted hardware (e.g., Town Crier [78]), against which various attacks have recently emerged, e.g., [24].

Another class of oracle schemes assumes server-side cooperation, mandating that servers install TLS extensions (e.g., [65]) or change application-layer logic (e.g., [31, 77]). Server-facilitated oracle schemes suffer from two fundamental problems. First, they break legacy compatibility, causing a significant barrier to wide adoption. Moreover, such solutions only provide *conditional* exportability because the web servers have the sole discretion to determine which data can be exported, and can censor export attempts at will. A mechanism that allows users to export *any* data they have access to would enable a whole host of currently unrealizable applications.

1.1 DECO

To address the above problems, we propose DECO, a decentralized oracle for TLS. Unlike oracle schemes that require per-website support, DECO is source-agnostic and supports *any* website running standard TLS. Unlike solutions that rely on websites' participation, DECO requires no server-side cooperation. Thus a single instance of DECO could enable *anyone* to become an oracle for *any* website.

DECO makes rich Internet data accessible with authenticity and privacy assurances to a wide range of applications, including ones that cannot access the Internet such as smart contracts. DECO could fundamentally shift today's model of web data dissemination by providing private data delivery with an option for transfer to third parties or public release. This technical capability highlights potential future legal and regulatory challenges, but also anticipates the creation and delivery of appealing new services. Importantly, DECO does not require trusted hardware, unlike alternative approaches that could achieve a similar vision, e.g., [54, 78].

At a high level, the prover commits to a piece of data D and proves to the verifier that D came from a TLS server S and optionally a statement π_D about D. E.g., in the example of proving age, the statement π_D could be the predicate "$D{=}y/m/d$ is Alice's date of birth and the current date - D is at least 18 years."

Informally, DECO achieves *authenticity*: The verifier is convinced only if the asserted statement about D is true and D is indeed obtained from website S. DECO also provides *privacy* in that the verifier only learns the that the statement π_D holds for some D obtained from S.

1.2 Technical challenges

Designing DECO with the required security and practical performance, while using legacy-(TLS)-compatible primitives, introduces several important technical challenges. The main challenge stems from the fact that TLS generates symmetric encryption and authentication keys that are *shared* by the client (prover in DECO) and web server. Thus, the client can *forge* arbitrary TLS session data, in the sense of signing the data with valid authentication keys.

To address this challenge, DECO introduces a novel *three-party handshake* protocol among the prover, verifier, and web server that creates an *unforgeable commitment* by the prover to the verifier on a piece of TLS session data D. The verifier can check that D is authentically from the TLS server. From the prover's perspective, the three-party handshake preserves the security of TLS in presence of a malicious verifier.

Efficient selective opening. After committing to D, the prover proves statements about the commitment. Although arbitrary statements can be supported in theory, we optimize for what are likely to be the most popular applications—revealing only substrings of the response to the verifier. We call such statements *selective opening*. Fine-grained selective opening allows users to hide sensitive information and reduces the input length to the subsequent proofs.

A naïve solution would involve expensive verifiable decryption of TLS records using generic zero-knowledge proofs (ZKPs), but we achieve an orders-of-magnitude efficiency improvement by exploiting the TLS record structure. For example, a direct implementation of verifiable decryption of a TLS record would involve proving correct execution of a circuit of 1024 AES invocations in zero-knowledge, whereas by leveraging the MAC-then-encrypt structure of CBC-HMAC, we achieve the same with only 3 AES invocations.

Context integrity. Selective opening allows the prover to only reveal a substring D' of the server's response D. However, a substring may mean different things depending on when it appears and a malicious prover could cheat by quoting out of context. Therefore we need to prove not just that D' appears in D, but that it appears in the expected context, i.e., D' has *context integrity* with respect to D. (Note that this differs from "contextual integrity" in privacy theory [57].)

Context-integrity attacks can be thwarted if the session content is structured and can be parsed. Fortunately most web data takes this form (e.g., in JSON or HTML). A generic solution is to parse the entire session and prove that the revealed part belongs to the necessary branch of a parse tree. But, under certain constraints that web data generally satisfies, parsing the entire session is not necessary. We propose a novel *two-stage parsing scheme* where the prover pre-processes the session content, and only parses the outcome that is usually much smaller. We draw from the definition of equivalence of programs, as used in programming language theory, to build a formal framework to reason about the security of two-stage parsing schemes. We provide several practical realizations for specific grammars. Our definitions and constructions generalize to other oracles too. For example, it could prevent a generic version of the content-hidden attack mentioned in [65].

1.3 Implementation and evaluation

We designed and implemented DECO as a complete end-to-end system. To demonstrate the system's power, we implemented three applications: 1) a confidentiality-preserving *financial instrument* using smart contracts; 2) converting legacy credentials to *anonymous credentials*; and 3) verifiable claims against *price discrimination*.

Our experiments with these applications show that DECO is highly efficient. For example, for TLS 1.2 in the WAN setting, online time is 2.85s to perform the three-party handshake and 2.52s for 2PC query execution. It takes 3-13s to generate zero-knowledge proofs for the applications described above. More details are in Sec. 7.

Contributions. In summary, our contributions are as follows:

- We introduce DECO, a provably secure decentralized oracle scheme, along with an implementation and performance evaluation. DECO is the first oracle scheme for modern TLS versions (both 1.2 and 1.3) that doesn't require trusted hardware or server-side modifications. We provide an overview of the protocol in Sec. 3 and specify the full protocol in Sec. 4.
- **Selective opening**: In Sec. 5.1, we introduce a broad class of statements for TLS records that can be proven efficiently in zero-knowledge. They allow users to open only substrings of a session-data commitment. The optimizations achieve substantial efficiency improvement over generic ZKPs.
- **Context-integrity attacks and mitigation**: We identify a new class of context-integrity attacks universal to privacy-preserving oracles (e.g. [65]). In Sec. 5.2, we introduce our mitigation involving a novel, efficient two-stage parsing scheme, along with a formal security analysis, and several practical realizations.
- **Security definitions and proofs**: Oracles are a key part of the smart contract ecosystem, but a coherent security definition has been lacking. We formalize and strengthen existing oracle schemes and present a formal security definition using an ideal functionality in Sec. 3.2. We prove the functionality is securely realized by our protocols in App. D.
- **Applications and evaluation**: In Sec. 6, we present three representative applications that showcase DECO's capabilities, and evaluate them in Sec. 7.

- **Legal and compliance considerations**: DECO can export data from websites without their explicit approval or even awareness. We discuss the resulting legal and compliance issues in Sec. 8.

2 BACKGROUND

2.1 Transport Layer Security (TLS)

We now provide necessary background on the TLS handshake and record protocols on which DECO builds.

TLS is a family of protocols that provides privacy and data integrity between two communicating applications. Roughly speaking, it consists of two protocols: a handshake protocol that sets up the session using asymmetric cryptography, establishing shared client and server keys for the next protocol, the record protocol, in which data is transmitted with confidentiality and integrity protection using symmetric cryptography.

Handshake. In the handshake protocol, the server and client first agree on a set of cryptographic algorithms (also known as a cipher suite). They then authenticate each other (client authentication optional), and finally securely compute a shared secret to be used for the subsequent record protocol.

DECO supports the recommended elliptic curve DH key exchange with ephemeral secrets (ECDHE [20]).

Record protocol. To transmit application-layer data (e.g., HTTP messages) in TLS, the record protocol first fragments the application data $\boldsymbol{D}$ into fixed sized plaintext *records* $\boldsymbol{D} = (D_1, \cdots, D_n)$. Each record is usually padded to a multiple of blocks (e.g., 128 bits). The record protocol then optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level protocols. The specific cryptographic operations depend on the negotiated ciphersuite. DECO supports the AES cipher in two commonly used modes: CBC-HMAC and GCM. We refer readers to [36] for how these primitives are used in TLS.

Differences between TLS 1.2 and 1.3. Throughout the paper we focus on TLS 1.2 and discuss how to generalize our techniques to TLS 1.3 in Sec. 4.1.2. Here we briefly note the major differences between these two TLS versions. TLS 1.3 removes the support for legacy non-AEAD ciphers. The handshake flow has also been restructured. All handshake messages after the ServerHello are now encrypted. Finally, a different key derivation function is used. For a complete description, see [64].

2.2 Multi-party computation

Consider a group of n parties $\mathcal{P}_1,...,\mathcal{P}_n$, each of whom holds some secret s_i. Secure multi-party computation (MPC) allows them to jointly compute $f(s_i,\cdots,s_n)$ without leaking any information other than the output of f, i.e., $\mathcal{P}_i$ learns nothing about $s_{j\neq i}$. Security for MPC protocols generally considers an adversary that corrupts t players and attempts to learn the private information of an honest player. Two-party computation (2PC) refers to the special case of $n=2$ and $t=1$. We refer the reader to [52] for a full discussion of the model and formal security definitions.

There are two general approaches to 2PC protocols. Garbled-circuit protocols based on Yao [76] encode f as a boolean circuit, an approach best-suited for bitwise operations (e.g., SHA-256). Other protocols leverage *threshold secret sharing* and are best suited for arithmetic operations. The functions we compute in this paper using 2PC, though, include both bitwise and arithmetic operations. We separate them into two components, and use the optimized garbled-circuit protocol from [75] for the bitwise operations and the secret-sharing based MtA protocol from [41] for the arithmetic operations.

3 OVERVIEW

In this section we state the problem we try to solve with DECO and present a high-level overview of its architecture.

3.1 Problem statement: Decentralized oracles

Broadly, we investigate protocols for building "oracles," i.e., entities that can prove provenance and properties of online data. The goal is to allow a prover $\mathcal{P}$ to prove to a verifier $\mathcal{V}$ that a piece of data came from a particular website $\mathcal{S}$ and optionally prove statements about such data in zero-knowledge, keeping the data itself secret. Accessing the data may require private input (e.g., a password) from $\mathcal{P}$ and such private information should be kept secret from $\mathcal{V}$ as well.

We focus on servers running TLS, the most widely deployed security protocol suite on the Internet. However, TLS alone does not prove data provenance. Although TLS uses public-key signatures for authentication, it uses symmetric-key primitives to protect the integrity and confidentiality of exchanged messages, using a shared session key established at the beginning of each session. Hence $\mathcal{P}$, who knows this symmetric key, cannot prove statements about cryptographically authenticated TLS data to a third party.

A web server itself could assume the role of an oracle, e.g., by simply signing data. However, server-facilitated oracles would not only incur a high adoption cost, but also put users at a disadvantage: the web server could impose arbitrary constraints on the oracle capability. We are interested in a scheme where anyone can prove provenance of any data she can access, without needing to rely on a single, central point of control, such as the web server providing the data.

We tackle these challenges by introducing *decentralized oracles* that don't rely on trusted hardware or cooperation from web servers. The problem is much more challenging than for previous oracles, as it precludes solutions that require servers to modify their code or deploy new software, e.g., [65], or use of prediction markets, e.g., [12, 62], while at the same time going beyond these previous approaches by supporting proofs on arbitrary predicates over data. Another approach, introduced in [78], is to use trusted execution environments (TEEs) such as Intel SGX. The downside is that recent attacks [24] may deter some users from trusting TEEs.

Authenticated data feeds for smart contracts. An important application of oracle protocols is to construct authenticated data feeds (ADFs, as coined in [78]), i.e., data with verifiable provenance and correctness, for smart contracts. Protocols such as [78] generate ADFs by signing TLS data using a key kept secret in a TEE. However, the security of this approach relies on that of TEEs. Using multiple TEEs could help achieve stronger integrity, but not privacy. If a single TEE is broken, TLS session content, including user credentials, can leak from the broken TEE.

DECO operates in a different model. Since smart contracts can't participate in 2PC protocols, they must rely on oracle nodes to participate as $\mathcal{V}$ on their behalf. Therefore we envision DECO being

Functionality $\mathcal{F}_{\text{Oracle}}$ between $\mathcal{S}$,$\mathcal{P}$ and $\mathcal{V}$

Input: The prover $\mathcal{P}$ holds some private input θ_s. The verifier $\mathcal{V}$ holds a query template Query and a statement Stmt.

Functionality:

- If at any point during the session, a message (sid, receiver, m) with receiver $\in \{\mathcal{S}, \mathcal{P}, \mathcal{V}\}$ is received from $\mathcal{A}$, forward (sid, m) to receiver and forward any responses to $\mathcal{A}$.
- Upon receiving input (sid,Query,Stmt) from $\mathcal{V}$, send (sid,Query,Stmt) to $\mathcal{P}$. Wait for $\mathcal{P}$ to reply with "ok" and θ_s.
- Compute Q=Query(θ_s) and send (sid,Q) to $\mathcal{S}$ and record its response (sid,R). Send (sid,$|Q|$,$|R|$) to $\mathcal{A}$.
- Send (sid,Q,R) to $\mathcal{P}$ and (sid, Stmt(R),$\mathcal{S}$) to $\mathcal{V}$.

Figure 1: The oracle functionality.

deployed in a decentralized oracle network similar to [39], where a set of independently operated oracles are available for smart contracts to use. Note that oracles running DECO are trusted only for integrity, not for privacy. Smart contracts can further hedge against integrity failures by querying multiple oracles and requiring, e.g., majority agreement, as already supported in [39]. We emphasize that DECO's privacy is preserved even all oracles are compromised. Thus DECO enables users to provide ADFs derived from private data to smart contracts while hiding private data from oracles.

3.2 Notation and definitions

We use $\mathcal{P}$ to denote the prover, $\mathcal{V}$ the verifier and $\mathcal{S}$ the TLS server. We use letters in boldface (e.g., $\boldsymbol{M}$) to denote vectors and M_i to denote the ith element in $\boldsymbol{M}$.

We model the essential properties of an oracle using an ideal functionality $\mathcal{F}_{\text{Oracle}}$ in fig. 1. To separate parallel runs of $\mathcal{F}_{\text{Oracle}}$, all messages are tagged with a unique *session id* denoted sid. We refer readers to [30] for details of ideal protocol execution.

$\mathcal{F}_{\text{Oracle}}$ accepts a secret parameter θ_s (e.g., a password) from $\mathcal{P}$, a query template Query and a statement Stmt from $\mathcal{V}$. A query template is a function that takes $\mathcal{P}$'s secret θ_s and returns a complete query, which contains public parameters specified by $\mathcal{V}$. An example query template would be Query(θ_s) = "stock price of GOOG on Jan 1st, 2020 with API key = θ_s". The prover $\mathcal{P}$ can later prove that the query sent to the server is well-formed, i.e., built from the template, without revealing the secret. The statement Stmt is a function that $\mathcal{V}$ wishes to evaluate on the server's response. Following the previous example, as the response R is a number, the following statement would compare it with a threshold: Stmt(R) = "$R > \$1{,}000$".

After $\mathcal{P}$ acknowledges the query template and the statement (by sending "ok" and θ_s), $\mathcal{F}_{\text{Oracle}}$ retrieves a response R from $\mathcal{S}$ using a query built from the template. We assume an honest server, so R is the ground truth. $\mathcal{F}_{\text{Oracle}}$ sends Stmt(R) and the data source to $\mathcal{V}$.

As stated in Def. 3.1, we are interested in decentralized oracles that don't require any server-side modifications or cooperation, i.e., $\mathcal{S}$ follows the unmodified TLS protocol.

Definition 3.1. A decentralized oracle protocol for TLS is a three-party protocol Prot = (Prot$_{\mathcal{S}}$,Prot$_{\mathcal{P}}$,Prot$_{\mathcal{V}}$) such that 1) Prot realizes $\mathcal{F}_{\text{Oracle}}$ and 2) Prot$_{\mathcal{S}}$ is the standard TLS, possibly along with an application-layer protocol.

Adversarial model and security properties. We consider a static, malicious network adversary $\mathcal{A}$. Corrupted parties may deviate arbitrarily from the protocol and reveal their states to $\mathcal{A}$. As a network adversary, $\mathcal{A}$ learns the message length from $\mathcal{F}_{\text{Oracle}}$ since TLS is not length-hiding. We assume $\mathcal{P}$ and $\mathcal{V}$ choose and agree on an appropriate query (e.g., it should be idempotent for most applications) and statement according to the application-layer protocol run by $\mathcal{S}$.

For a given query Q, denote the server's honest response by $\mathcal{S}(Q)$. We require that security holds when either $\mathcal{P}$ or $\mathcal{V}$ is corrupted. The functionality $\mathcal{F}_{\text{Oracle}}$ reflects the following security guarantees:

- *Prover-integrity:* A malicious $\mathcal{P}$ cannot forge content provenance, nor can she cause $\mathcal{S}$ to accept invalid queries or respond incorrectly to valid ones. Specifically, if the verifier inputs (Query,Stmt) and outputs (b,$\mathcal{S}$), then $\mathcal{P}$ must have sent Q=Query(θ_s) to $\mathcal{S}$ in a TLS session, receiving response R=$\mathcal{S}(Q)$ such that b=Stmt(R).
- *Verifier-integrity:* A malicious $\mathcal{V}$ cannot cause $\mathcal{P}$ to receive incorrect responses. Specifically, if $\mathcal{P}$ outputs (Q,R) then R must be the server's response to query Q submitted by $\mathcal{P}$, i.e., R=$\mathcal{S}(Q)$.
- *Privacy:* A malicious $\mathcal{V}$ learns only public information (Query,$\mathcal{S}$) and the evaluation of Stmt(R).

3.3 A strawman protocol

We focus on two widely used representative TLS cipher suites: CBC-HMAC and AES-GCM. Our technique generalizes to other ciphers (e.g., Chacha20-Poly1305, etc.) as well. Throughout this section we use CBC-HMAC to illustrate the ideas, with discussion of GCM deferred to later sections.

TLS uses separate keys for each direction of communication. Unless explicitly specified, we don't distinguish between the two and use $\mathsf{k}^{\mathsf{Enc}}$ and $\mathsf{k}^{\mathsf{MAC}}$ to denote session keys for both directions.

In presenting our design of DECO, we start with a strawman protocol and incrementally build up to the full protocol.

A strawman protocol. A strawman protocol that realizes $\mathcal{F}_{\text{Oracle}}$ between ($\mathcal{P}$,$\mathcal{V}$) is as follows. $\mathcal{P}$ queries the server $\mathcal{S}$ and records all messages sent to and received from the server in $\hat{\boldsymbol{Q}}=(\hat{Q}_1,\ldots,\hat{Q}_n)$ and $\hat{\boldsymbol{R}}=(\hat{R}_1,\ldots,\hat{R}_n)$, respectively. Let $\hat{\boldsymbol{M}}=(\hat{\boldsymbol{Q}},\hat{\boldsymbol{R}})$ and ($\mathsf{k}^{\mathsf{MAC}}$,$\mathsf{k}^{\mathsf{Enc}}$) be the session keys.

She then proves in zero-knowledge that 1) each $\hat{R}_i$ decrypts to $R_i \| \sigma_i$, a plaintext record and a MAC tag; 2) each MAC tag σ_i for R_i verifies against $\mathsf{k}^{\mathsf{MAC}}$; and 3) the desired statement evaluates to b on the response, i.e., b=Stmt($\boldsymbol{R}$). Using the now standard notation introduced in [28], $\mathcal{P}$ computes

$$p_r = \text{ZK-PoK}\{\mathsf{k}^{\mathsf{Enc}},\boldsymbol{R}:\forall i\in[n],\mathsf{Dec}(\mathsf{k}^{\mathsf{Enc}},\hat{R}_i)=R_i\|\sigma_i \wedge \mathsf{Verify}(\mathsf{k}^{\mathsf{MAC}},\sigma_i,R_i)=1\wedge\mathsf{Stmt}(\boldsymbol{R})=b\}.$$

She also proves that $\boldsymbol{Q}$ is well-formed as $\boldsymbol{Q}$=Query(θ_s) similarly in a proof p_q and sends (p_q,p_r,$\mathsf{k}^{\mathsf{MAC}}$,$\hat{\boldsymbol{M}}$,$b$) to $\mathcal{V}$.

Given that $\hat{\boldsymbol{M}}$ is an authentic transcript of the TLS session, the prover-integrity property seems to hold. Intuitively, CBC-HMAC ciphertexts bind to the underlying plaintexts, thus $\hat{\boldsymbol{M}}$ can be treated as secure commitments [42] to the session data. That is, a given $\hat{\boldsymbol{M}}$ can only be opened (i.e., decrypted and MAC checked) to a unique message. The binding property prevents $\mathcal{P}$ from opening $\hat{\boldsymbol{M}}$ to a different message other than the original session with the server.

Unfortunately, this intuition is flawed. The strawman protocol fails completely because it *cannot* ensure the authenticity of $\hat{\boldsymbol{M}}$. The prover $\mathcal{P}$ has the session keys, and thus she can include the encryption of arbitrary messages in $\hat{\boldsymbol{M}}$.

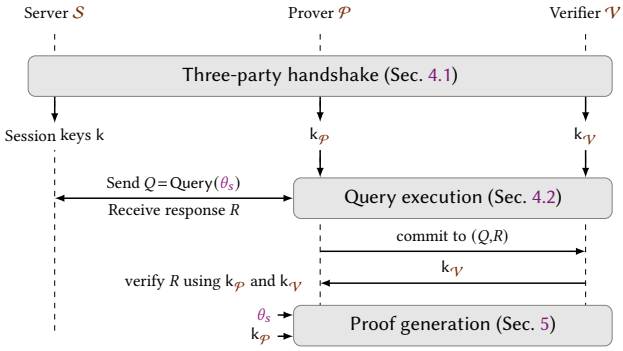

Figure 2: An overview of the workflow in DECO.

Moreover, the zero-knowledge proofs that $\mathcal{P}$ needs to construct involve decrypting and hashing the entire transcript, which can be prohibitively expensive. For the protocol to be practical, we need to significantly reduce the cost.

3.4 Overview of DECO

The critical failing of our strawman approach is that $\mathcal{P}$ learns the session key before she commits to the session. One key idea in DECO is to withhold the MAC key from $\mathcal{P}$ until *after* she commits. The TLS session between $\mathcal{P}$ and $\mathcal{S}$ must still provide confidentiality and integrity. Moreover, the protocol must not degrade performance below the requirements of TLS (e.g., triggering a timeout).

As shown in fig. 2, DECO is a three-phase protocol. The first phase is a novel **three-party handshake** protocol in which the prover $\mathcal{P}$, the verifier $\mathcal{V}$, and the TLS server $\mathcal{S}$ establish session keys that are *secret-shared between* $\mathcal{P}$ *and* $\mathcal{V}$. After the handshake is a **query execution** phase during which $\mathcal{P}$ accesses the server following the standard TLS protocol, but with help from $\mathcal{V}$. After $\mathcal{P}$ commits to the query and response, $\mathcal{V}$ reveals her key share. Finally, $\mathcal{P}$ proves statements about the response in a **proof generation** phase.

3.4.1 Three-party handshake. Essentially, $\mathcal{P}$ and $\mathcal{V}$ *jointly* act as a TLS client. They negotiate a shared session key with $\mathcal{S}$ in a secret-shared form. We emphasize that this phase, like the rest of DECO, is completely transparent to $\mathcal{S}$, requiring no server-side modifications.

For the CBC-HMAC cipher suite, at the end of the three-party handshake, $\mathcal{P}$ and $\mathcal{V}$ receive $k_{\mathcal{P}}^{MAC}$ and $k_{\mathcal{V}}^{MAC}$ respectively, while $\mathcal{S}$ receives $k^{MAC} = k_{\mathcal{P}}^{MAC} + k_{\mathcal{V}}^{MAC}$. As with the standard handshake, both $\mathcal{P}$ and $\mathcal{S}$ get the encryption key k^{Enc}.

Three-party handshake can make the aforementioned session-data commitment unforgeable as follows. At the end of the session, $\mathcal{P}$ first commits to the session in $\hat{\boldsymbol{M}}$ as before, then $\mathcal{V}$ reveals her share $k_{\mathcal{V}}^{MAC}$. From $\mathcal{V}$'s perspective, the three-party handshake protocol ensures that a fresh MAC key (for each direction) is used for every session, despite the influence of a potential malicious prover, and that the keys are unknown to $\mathcal{P}$ until she commits. Without knowledge of the MAC key, $\mathcal{P}$ cannot forge or tamper with session data before committing to it. The unforgeability of the session-data commitment in DECO thus reduces to the unforgeability of the MAC scheme used in TLS.

Other ciphersuites such as GCM can be supported similarly. In GCM, a single key (for each direction) is used for both encryption and MAC. The handshake protocol similarly secret-shares the key between $\mathcal{P}$ and $\mathcal{V}$. The handshake protocol are presented in Sec. 4.1.

3.4.2 Query execution. Since the session keys are secret-shared, as noted, $\mathcal{P}$ and $\mathcal{V}$ execute an interactive protocol to construct a TLS message encrypting the query. $\mathcal{P}$ then sends the message to $\mathcal{S}$ as a standard TLS client. For CBC-HMAC, they compute the MAC tag of the query, while for GCM they perform authenticated encryption. Note that the query is private to $\mathcal{P}$ and should not be leaked to $\mathcal{V}$. Generic 2PC would be expensive for large queries, so we instead introduce custom 2PC protocols that are orders-of-magnitude more efficient than generic solutions, as presented in Sec. 4.2.

As explained previously, $\mathcal{P}$ commits to the session data $\hat{\boldsymbol{M}}$ before receiving $\mathcal{V}$'s key share, making the commitment unforgeable. Then $\mathcal{P}$ can verify the integrity of the response, and prove statements about it, which we present now.

3.4.3 Proof generation. With unforgeable commitments, if $\mathcal{P}$ opens the commitment $\hat{\boldsymbol{M}}$ completely (i.e., reveals the encryption key) then $\mathcal{V}$ could easily verify the authenticity of $\hat{\boldsymbol{M}}$ by checking MACs on the decryption.

Revealing the encryption key for $\hat{\boldsymbol{M}}$, however, would breach privacy: it would reveal *all* session data exchanged between $\mathcal{P}$ and $\mathcal{S}$. In theory, $\mathcal{P}$ could instead prove any statement Stmt over $\hat{\boldsymbol{M}}$ in zero knowledge (i.e., without revealing the encryption key). Generic zero-knowledge proof techniques, though, would be prohibitively expensive for many natural choices of Stmt.

DECO instead introduces two techniques to support efficient proofs for a broad, general class of statement, namely *selective opening* of a TLS session transcript. Selective opening involves either *revealing* a substring to $\mathcal{V}$ or *redacting*, i.e., excising, a substring, concealing it from $\mathcal{V}$.

As an example, fig. 3 shows a simplified JSON bank statement for Bob. Suppose Bob ($\mathcal{P}$) wants to reveal his checking account balance to $\mathcal{V}$. Revealing the decryption key for his TLS session would be undesirable: it would *also* reveal the entire statement, including his transactions. Instead, using techniques we introduce, Bob can efficiently reveal only the substring in lines 5-7. Alternatively, if he doesn't mind revealing his savings account balance, he might redact his transactions after line 7.

The two selective opening modes, revealing and redacting substrings, are useful privacy protection mechanisms. They can also serve as pre-processing for a subsequent zero-knowledge proof. For example, Bob might wish to prove that he has an account with a balance larger than $1000, without revealing the actual balance. He would then prove in zero knowledge a predicate ("balance > $1000") over the substring that includes his checking account balance.

Selective opening *alone*, however, is not enough for many applications. This is because the *context* of a substring affects its meaning. Without what we call *context integrity*, $\mathcal{P}$ could cheat and reveal a substring that falsely appears to prove a claim to $\mathcal{V}$. For example, Bob might not have a balance above $1000. After viewing his bank statement, though, he might in the same TLS session post a message to customer service with the substring `"balance": $5000` and then view his pending messages (in a form of reflection attack). He could then reveal this substring to fool $\mathcal{V}$.

Various sanitization heuristics on prover-supplied inputs to $\mathcal{V}$, e.g., truncating session transcripts, could potentially prevent some

```
{"name": "Bob",
"savings a/c": {
    "balance": $5000
},
"checking a/c": {
    "balance": $2000
},
"transactions": {...}}
```

Figure 3: Example bank statement to demonstrate selective opening and context-integrity attacks.

such attacks, but, like other forms of web application input sanitization, are fragile and prone to attack [68].

Instead, we introduce a rigorous technique by which session data are explicitly but confidentially parsed. We call this technique *zero-knowledge two-stage parsing*. The idea is that $\mathcal{P}$ parses $\hat{M}$ locally in a first stage and then proves to $\mathcal{V}$ a statement in zero knowledge about constraints on a resulting substring. For example, in our banking example, if bank-supplied key-value stores are always escaped with a distinguished character λ, then Bob could prove a correct balance by extracting via local parsing and revealing to $\mathcal{V}$ a substring `"balance": $5000` preceded by λ. We show for a very common class of web API grammars (unique keys) that this two-phase approach yields much more efficient proofs than more generic techniques.

Section 5 gives more details on proof generation in DECO.

4 THE DECO PROTOCOL

We now specify the full DECO protocol, which consists of a three-party handshake in Sec. 4.1, followed by 2PC protocols for query execution in Sec. 4.2, and a proof generation phase. We prove its security in Sec. 4.3.

4.1 Three-party handshake

The goal of the three-party handshake (3P-HS) is to secret-share between the prover $\mathcal{P}$ and verifier $\mathcal{V}$ the session keys used in a TLS session with server $\mathcal{S}$, in a way that is completely transparent to $\mathcal{S}$. We first focus on CBC-HMAC for exposition, then adapt the protocol to support GCM.

As with the standard TLS handshake, 3P-HS is two-step: first, $\mathcal{P}$ and $\mathcal{V}$ compute additive shares of a secret $Z \in EC(\mathbb{F}_p)$ shared with the server through a TLS-compatible key exchange protocol. ECDHE is the recommended and the focus here; second, $\mathcal{P}$ and $\mathcal{V}$ derive secret-shared session keys by securely evaluating the TLS-PRF [36] with their shares of Z as inputs. The full protocol is specified in fig. 6. Below we give text descriptions so formal specifications are not required for understanding.

4.1.1 Step 1: key exchange. Let $EC(\mathbb{F}_p)$ denote the EC group used in ECDHE and G its generator.

The prover $\mathcal{P}$ initiates the handshake by sending a regular TLS handshake request and a random nonce r_c to $\mathcal{S}$ (in the ClientHello message). On receiving a certificate, the server nonce r_s, and a signed ephemeral DH public key $Y_S = s_S \cdot G$ from $\mathcal{S}$ (in the ServerHello and ServerKeyExchange messages), $\mathcal{P}$ checks the certificate and the signature and forwards them to $\mathcal{V}$. After performing the same check, $\mathcal{V}$ samples a secret s_V and sends her part of the DH public key $Y_V = s_V \cdot G$ to $\mathcal{P}$, who then samples another secret s_P and sends the combined DH public key $Y_P = s_P \cdot G + Y_V$ to $\mathcal{S}$.

Since the server $\mathcal{S}$ runs the standard TLS, $\mathcal{S}$ will compute a DH secret as $Z = s_S \cdot Y_P$. $\mathcal{P}$ (and $\mathcal{V}$) computes its share of Z as $Z_P = s_P \cdot Y_S$ (and $Z_V = s_V \cdot Y_S$). Note that $Z = Z_P + Z_V$ where + is the group operation of $EC(\mathbb{F}_p)$. Assuming the discrete logarithm problem is hard in the chosen group, Z is unknown to either party.

4.1.2 Step 2: key derivation. Now that $\mathcal{P}$ and $\mathcal{V}$ have established additive shares of Z (in the form of *EC points*), they proceed to derive session keys by evaluating the TLS-PRF [36] keyed with the x coordinate of Z.

A technical challenge here is to harmonize arithmetic operations (i.e., addition in $EC(\mathbb{F}_p)$) with bitwise operations (i.e., TLS-PRF) in 2PC. It is well-known that boolean circuits are not well-suited for arithmetic in large fields. As a concrete estimate, an EC Point addition resulting in just the x coordinate involves 4 subtractions, one modular inversion, and 2 modular multiplications. An estimate of the AND complexity based on the highly optimized circuits of [34] results in over 900,000 AND gates just for the subtractions, multiplications, and modular reductions—not even including inversion, which would require running the Extended Euclidean algorithm inside a circuit.

Due to the prohibitive cost of adding EC points in a boolean circuit, $\mathcal{P}$ and $\mathcal{V}$ convert the additive shares of an EC point in $EC(\mathbb{F}_p)$ to additive shares of its x-coordinate in $\mathbb{F}_p$, using the ECtF protocol presented below. Then the boolean circuit just involves adding two numbers in $\mathbb{F}_p$, which can be done with only $\sim 3|p|$ AND gates, that is ~ 768 AND gates in our implementation where p is 256-bit.

ECtF: Converting shares in $EC(\mathbb{F}_p)$ to shares in $\mathbb{F}_p$. The inputs to an ECtF protocol are two EC points $P_1, P_2 \in EC(\mathbb{F}_p)$, denoted $P_i = (x_i, y_i)$. Suppose $(x_s, y_s) = P_1 \star P_2$ where $\star$ is the EC group operation, the output of the protocol is $\alpha, \beta \in \mathbb{F}_p$ such that $\alpha + \beta = x_s$. Specifically, for the curve we consider, $x_s = \lambda^2 - x_1 - x_2$ where $\lambda = (y_2 - y_1)/(x_2 - x_1)$. Shares of the y_s can be computed similarly but we omit that since TLS only uses the x_s.

ECtF uses a Multiplicative-to-Additive (MtA) share-conversion protocol as a building block. We use $\alpha, \beta := \mathsf{MtA}(a, b)$ to denote a run of MtA between Alice and Bob with inputs a and b respectively. At the end of the run, Alice and Bob receive α and β such that $a \cdot b = \alpha + \beta$. The protocol can be generalized to handle vector inputs without increasing the communication complexity. Namely for vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}_p^n$, if $\alpha, \beta := \mathsf{MtA}(\boldsymbol{a}, \boldsymbol{b})$, then $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \alpha + \beta$. See, e.g., [41] for a Paillier [61]-based construction.

Now we specify the protocol of ECtF. ECtF has two main ingredients. Let $[a]$ denote a 2-out-of-2 sharing of a, i.e., $[a] = (a_1, a_2)$ such that party i has a_i for $i \in \{1,2\}$ while $a = a_1 + a_2$. The first ingredient is share inversion: given $[a]$, compute $[a^{-1}]$. As shown in [41], we can use the inversion protocol of Bar-Ilan and Beaver [17] together with MtA as follows: party i samples a random value r_i and executes MtA to compute $\delta_1, \delta_2 := \mathsf{MtA}((a_1, r_1), (r_2, a_2))$. Note that $\delta_1 + \delta_2 = a_1 \cdot r_2 + a_2 \cdot r_1$. Party i publishes $v_i = \delta_i + a_i \cdot r_i$ and thus both parties learn $v = v_1 + v_2$. Finally, party i outputs $\beta_i = r_i \cdot v^{-1}$. The protocol computes a correct sharing of a^{-1} because $\beta_1 + \beta_2 = a^{-1}$. Moreover, the protocol doesn't leak a to any party assuming MtA is secure. In fact, party i's view consists of $(a_1 + a_2)(r_1 + r_2)$, which is uniformly random since r_i is uniformly random.

The second ingredient is share multiplication: compute $[ab]$ given $[a], [b]$. $[ab]$ can be computed using MtA as follows: parties

execute MtA to compute α_1,α_2 such that $\alpha_1+\alpha_2 = a_1\cdot b_2 + a_2\cdot b_2$. Then, party i outputs $m_i = \alpha_i + a_i\cdot y_i$. The security and correctness of the protocol can be argued similarly as above.

Combining these two ingredients, fig. 7 in the Appendix presents the ECtF protocol, with communication complexity 8 ciphertexts.

Secure evaluation of the TLS-PRF. Having computed shares of the x-coordinate of Z, the so called premaster secret in TLS, in ECtF, $\mathcal{P}$ and $\mathcal{V}$ evaluate the TLS-PRF in 2PC to derive session keys. Beginning with the SHA-256 circuit of [29], we hand-optimized the TLS handshake circuit resulting in a circuit with total AND complexity of 779,213.

Adapting to support GCM. For GCM, a single key (for each direction) is used for both encryption and MAC. Adapting the above protocol to support GCM in TLS 1.2 is straightforward. The first step would remain identical, while output of the second step needs to be truncated, as GCM keys are shorter.

Adapting to TLS 1.3. The specification of TLS 1.3 [64] has been recently published. To support TLS 1.3, the 3P-HS protocol must be adapted to a new handshake flow and a different key derivation circuit. Notably, all handshake messages after the ServerHello are now *encrypted*. A naïve strategy would be to decrypt them in 2PC, which would be costly as certificates are usually large. However, thanks to the key independence property of TLS 1.3 [37], we can construct a 3P-HS protocol of similar complexity to that for TLS 1.2, as outlined in App. C.1.

4.2 Query execution

After the handshake, the prover $\mathcal{P}$ sends her query Q to the server $\mathcal{S}$ as a standard TLS client, but with help from the verifier $\mathcal{V}$. Specifically, since session keys are secret-shared, the two parties need to interact and execute a 2PC protocol to construct TLS records encrypting Q. Although generic 2PC would in theory suffice, it would be expensive for large queries. We instead introduce custom 2PC protocols that are orders-of-magnitude more efficient.

We first focus on one-round sessions where $\mathcal{P}$ sends all queries to $\mathcal{S}$ before receiving any response. Most applications of DECO, e.g., proving provenance of content retrieved via HTTP, are one-round. Extending DECO to multi-round sessions is discussed in App. C.

4.2.1 CBC-HMAC. Recall that $\mathcal{P}$ and $\mathcal{V}$ hold shares of the MAC key, while $\mathcal{P}$ holds the encryption key. To construct TLS records encrypting Q—potentially private to $\mathcal{P}$, the two parties first run a 2PC protocol to compute the HMAC tag τ of Q, and then $\mathcal{P}$ encrypts $Q\|\tau$ locally and sends the ciphertext to $\mathcal{S}$.

Let H denote SHA-256. Recall that the HMAC of message m with key k is $\mathsf{HMAC}(\mathsf{k},m) = \mathsf{H}((\mathsf{k}\oplus\mathsf{opad})\|\underbrace{\mathsf{H}((\mathsf{k}\oplus\mathsf{ipad})\|m)}_{\text{inner hash}})$.

A direct 2PC implementation would be expensive for large queries, as it requires hashing the entire query in 2PC to compute the inner hash. The key idea in our optimization is to make the computation of the inner hash local to $\mathcal{P}$ (i.e., without 2PC). If $\mathcal{P}$ knew $\mathsf{k}\oplus\mathsf{ipad}$, she could compute the inner hash. We cannot, though, simply give $\mathsf{k}\oplus\mathsf{ipad}$ to $\mathcal{P}$, as she could then learn k and forge MACs.

Our optimization exploits the Merkle–Damgård structure in SHA-256. Suppose m_1 and m_2 are two correctly sized blocks. Then

$\mathsf{Prot}_{\mathsf{DECO}}$

$\mathsf{Prot}_{\mathcal{S}}$: follow the standard TLS protocol.

$\mathsf{Prot}_{\mathcal{P}}$ and $\mathsf{Prot}_{\mathcal{V}}$:

- $\mathcal{V}$ sends (sid,Query, Stmt) to $\mathcal{P}$, where Query is the query template and Stmt the statement to be proven over the response to $\mathcal{P}$.
- $\mathcal{P}$ examines them and chooses whether to proceed. If so, $\mathcal{P}$ starts the handshake.
- (**3P-HS**) $\mathcal{P},\mathcal{V}$ execute the three-party handshake protocol. $\mathcal{P}$ gets the encryption key $\mathsf{k}^{\mathsf{Enc}}$ and a share of the MAC key $\mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}}$, while $\mathcal{V}$ gets the other share $\mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}}$.
- (**Query**) $\mathcal{P}$ computes a query using the template $Q = \mathsf{Query}(\theta_s)$. $\mathcal{P}$ invokes 2PC-HMAC with $\mathcal{V}$ to compute a tag τ. $\mathcal{P}$ sends $(\mathsf{sid},\hat{Q} = \mathsf{Enc}(\mathsf{k}^{\mathsf{Enc}},Q\|\tau))$ to $\mathcal{S}$.
- (**Commit and verify**) After receiving a response $(\mathsf{sid},\hat{R})$ from $\mathcal{S}$, $\mathcal{P}$ sends $(\mathsf{sid},\hat{Q},\hat{R},\mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}})$ to $\mathcal{V}$ as a commitment to the session data. After receiving $(\mathsf{sid},\mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}})$ from $\mathcal{V}$, $\mathcal{P}$ computes $\mathsf{k}^{\mathsf{MAC}} = \mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}} + \mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}}$, decrypts $R\|\tau = \mathsf{Dec}(\mathsf{k}^{\mathsf{Enc}},\hat{R})$, and verifies τ against $\mathsf{k}^{\mathsf{MAC}}$.
- (**Proof gen**) Let $b = \mathsf{Stmt}(R)$, $x = (\mathsf{k}^{\mathsf{Enc}},\theta_s,Q,R)$ and $w = (\hat{Q},\hat{R},\mathsf{k}^{\mathsf{MAC}},b)$. $\mathcal{P}$ sends $(\mathsf{sid},$ "prove", $x,w)$ to $\mathcal{F}_{\mathsf{ZK}}$ and outputs (Q,R). If $\mathcal{V}$ receives $(\mathsf{sid},$"proof",$1,(\hat{Q},\hat{R},\hat{\mathsf{k}}^{\mathsf{MAC}},b))$ from $\mathcal{F}_{\mathsf{ZK}}$, $\mathcal{V}$ checks if $\hat{\mathsf{k}}^{\mathsf{MAC}} = \mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}} + \mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}}$. If so, $\mathcal{V}$ outputs $(\mathsf{sid},b,\mathcal{S})$.

Figure 4: The DECO protocol. We only show the CBC-HMAC variant for clarify, while the GCM variant is described in Sec. 4.3.

$\mathsf{H}(m_1\|m_2)$ is computed as $f_{\mathsf{H}}(f_{\mathsf{H}}(\mathsf{IV},m_1),m_2)$ where f_{H} denotes the one-way compression function of H, and IV the initial vector.

After the three-party handshake, $\mathcal{P}$ and $\mathcal{V}$ execute a simple 2PC protocol to compute $s_0 = f_{\mathsf{H}}(\mathsf{IV},\mathsf{k}^{\mathsf{MAC}}\oplus\mathsf{ipad})$, and reveal it to $\mathcal{P}$. To compute the inner hash of a message m, $\mathcal{P}$ just uses s_0 as the IV to compute a hash of m. Revealing s_0 does not reveal $\mathsf{k}^{\mathsf{MAC}}$, as f_{H} is assumed to be one-way. To compute $\mathsf{HMAC}(\mathsf{k},m)$ then involves computing the outer hash in 2PC on the inner hash, a much shorter message. Thus, we manage to reduce the amount of 2PC computation to a few blocks regardless of query length, as opposed to up to 256 SHA-2 blocks in each record with generic 2PC. The protocol is formally specified in fig. 8.

4.2.2 AES-GCM. For GCM, $\mathcal{P}$ and $\mathcal{V}$ perform authenticated encryption of Q. 2PC-AES is straightforward with optimized circuits (e.g., [11]), but computing tags for large queries is expensive as it involves evaluating long polynomials in a large field *for each record*. Our optimized protocol makes polynomial evaluation local via precomputation. We refer readers to App. B.2 for details. Since 2PC-GCM involves not only tag creation but also AES encryption, it incurs higher computational cost and latency than CBC-HMAC.

In App. C.4, we present a highly efficient alternative protocol that avoids post-handshake 2PC protocols altogether, with additional trust assumptions.

4.3 Full protocol

After querying the server and receiving a response, $\mathcal{P}$ commits to the session by sending the ciphertexts to $\mathcal{V}$, and receives $\mathcal{V}$'s MAC key share. Then $\mathcal{P}$ can verify the integrity of the response, and prove statements about it. Figure 4 specifies the full DECO protocol for CBC-HMAC (the protocol for GCM is similar and described later).

For clarity, we abstract away the details of zero-knowledge proofs in an ideal functionality $\mathcal{F}_{\mathsf{ZK}}$ like that in [45]. On receiving ("prove",x,w) from $\mathcal{P}$, where x and w are private and public witnesses respectively, $\mathcal{F}_{\mathsf{ZK}}$ sends w and the relationship $\pi(x,w)\in\{0,1\}$ (defined below) to $\mathcal{V}$. Specifically, for CBC-HMAC, x,w,π are defined as follows: $x = (\mathsf{k}^{\mathsf{Enc}},\theta_s,Q,R)$ and $w = (\hat{Q},\hat{R},\mathsf{k}^{\mathsf{MAC}},b)$. The relationship $\pi(x,w)$ outputs 1 if and only if (1) $\hat{Q}$ (and $\hat{R}$) is the

CBC-HMAC ciphertext of Q (and R) under key $\mathsf{k}^{\mathsf{Enc}}, \mathsf{k}^{\mathsf{MAC}}$; (2) $\mathsf{Query}(\theta_s) = Q$; and (3) $\mathsf{Stmt}(R) = b$. Otherwise it outputs 0.

Assuming functionalities for secure 2PC and ZKPs, it can be shown that $\mathsf{Prot}_{\mathsf{DECO}}$ UC-securely realizes $\mathcal{F}_{\mathsf{Oracle}}$ for malicious adversaries, as stated in Theorem 4.1. We provide a simulation-based proof (sketch) in App. D.

Theorem 4.1 (Security of $\mathsf{Prot}_{\mathsf{DECO}}$). *Assuming the discrete log problem is hard in the group used in the three-party handshake, and that f (the compression function of SHA-256) is an random oracle,* $\mathsf{Prot}_{\mathsf{DECO}}$ *UC-securely realizes $\mathcal{F}_{Oracle}$ in the $(\mathcal{F}_{2PC}, \mathcal{F}_{ZK})$-hybrid world, against a static malicious adversary with abort.*

The protocol for GCM has a similar flow. We've specified the GCM variants of the three-party handshake and query construction protocols. Unlike CBC-HMAC, GCM is not committing [42]: for a given ciphertext C encrypted with key k, one knowing k can efficiently find $\mathsf{k}' \neq \mathsf{k}$ that decrypts C to a different plaintext while passing the integrity check. To prevent such attacks, we require $\mathcal{P}$ to commit to her key share $\mathsf{k}_{\mathcal{P}}$ before learning $\mathcal{V}$'s key share. In the proof generation phase, in addition to proving statements about Q and R, $\mathcal{P}$ needs to prove that the session keys used to decrypt $\hat{Q}$ and $\hat{R}$ are valid against the commitment to $\mathsf{k}_{\mathcal{P}}$. Proof of the security of the GCM variant is like that for CBC-HMAC.

5 PROOF GENERATION

Recall that the prover $\mathcal{P}$ commits to the ciphertext $\hat{\boldsymbol{M}}$ of a TLS session and proves to $\mathcal{V}$ that the plaintext $\boldsymbol{M}$ satisfies certain properties. Without loss of generality, we assume $\hat{\boldsymbol{M}}$ and $\boldsymbol{M}$ contain only one TLS record, and henceforth call them the *ciphertext record* and the *plaintext record*. Multi-record sessions can be handled by repeating the protocol for each record.

Proving only the provenance of $\boldsymbol{M}$ is easy: just reveal the encryption keys. But this sacrifices privacy. Alternatively, $\mathcal{P}$ could prove any statement about $\boldsymbol{M}$ using general zero-knowledge techniques. But such proofs are often expensive.

In this section, we present two classes of statements optimized for what are likely to be the most popular applications: revealing only a substring of the response while proving its provenance (Sec. 5.1), or further proving that the revealed substring appears in a context expected by $\mathcal{V}$ (Sec. 5.2).

5.1 Selective opening

We introduce *selective opening*, techniques that allow $\mathcal{P}$ to efficiently *reveal* or *redact* substrings in the plaintext. Suppose the plaintext record is composed of chunks $\boldsymbol{M} = (B_1, \cdots, B_n)$ (details of chunking are discussed shortly). Selective opening allows $\mathcal{P}$ to prove that the ith chunk of $\boldsymbol{M}$ is B_i, without revealing the rest of $\boldsymbol{M}$; we refer to this as Reveal mode. It can also prove that $\boldsymbol{M}_{-i}$ is the same as $\boldsymbol{M}$ but with the chunks removed. We call this Redact mode. Both modes are simple, but useful for practical privacy goals. The granularity of selective opening depends on the cipher suite, which we now discuss.

5.1.1 CBC-HMAC. Recall that for proof generation, $\mathcal{P}$ holds both the encryption and MAC keys $\mathsf{k}^{\mathsf{Enc}}$ and $\mathsf{k}^{\mathsf{MAC}}$, while $\mathcal{V}$ only has the MAC key $\mathsf{k}^{\mathsf{MAC}}$. Our performance analysis assumes a ciphersuite with SHA-256 and AES-128, which matches our implementation, but the techniques are applicable to other parameters. Recall that MAC-then-encrypt is used: a plaintext record $\boldsymbol{M}$ contains up to 1024 AES blocks of data and 3 blocks of MAC tag σ, which we denote as $\boldsymbol{M} = (B_1, \ldots, B_{1024}, \sigma)$ where $\sigma = (B_{1025}, B_{1026}, B_{1027})$. $\hat{\boldsymbol{M}}$ is a CBC encryption of $\boldsymbol{M}$, consisting of the same number of blocks: $\hat{\boldsymbol{M}} = (\hat{B}_1, \ldots, \hat{B}_{1024}, \hat{\sigma})$ where $\hat{\sigma} = (\hat{B}_{1025}, \hat{B}_{1026}, \hat{B}_{1027})$.

Revealing a TLS record. A naïve way to prove that $\hat{\boldsymbol{M}}$ encrypts $\boldsymbol{M}$ without revealing $\mathsf{k}^{\mathsf{Enc}}$ is to prove correct encryption of each AES block in ZKP. However, this would require up to 1027 invocations of AES in ZKP, resulting in impractical performance.

Leveraging the MAC-then-encrypt structure, the same can be done using only 3 invocations of AES in ZKP. The idea is to prove that the last few blocks of $\hat{\boldsymbol{M}}$ encrypt a tag σ and reveal the plaintext directly. Specifically, $\mathcal{P}$ computes $\pi_\sigma = \text{ZK-PoK}\{\mathsf{k}^{\mathsf{Enc}} : \hat{\sigma} = \mathsf{CBC}(\mathsf{k}^{\mathsf{Enc}}, \sigma)\}$ and sends $(\boldsymbol{M}, \pi_\sigma)$ to $\mathcal{V}$. Then $\mathcal{V}$ verifies π and checks the MAC tag over $\boldsymbol{M}$ (note that $\mathcal{V}$ knows the MAC key.) Its security relies on the collision-resistance of the underlying hash function in HMAC, i.e., $\mathcal{P}$ cannot find $\boldsymbol{M}' \neq \boldsymbol{M}$ with the same tag σ.

Revealing a record with redacted blocks. Suppose the ith block contains sensitive information that $\mathcal{P}$ wants to redact. A direct strategy is to prove that $\boldsymbol{B}_{i-} = (B_1, \cdots, B_{i-1})$ and $\boldsymbol{B}_{i+} = (B_{i+1}, \cdots, B_n)$ form the prefix and suffix of the plaintext encrypted by $\hat{\boldsymbol{M}}$, by computing π_σ (see above) and $\text{ZK-PoK}\{B_i : \sigma = \mathsf{HMAC}(\mathsf{k}^{\mathsf{MAC}}, \boldsymbol{B}_{i-} \| B_i \| \boldsymbol{B}_{i+})\}$. This is expensive though as it would involve 3 AES and 256 SHA-256 compression in ZKP.

Leveraging the Merkle-Damgård structure of SHA-256 (c.f. Sec. 4.2.1), several optimization is possible. Let f denote the compression function of SHA-256, and s_{i-1} the state after applying f on $\boldsymbol{B}_{i-}$. First, if both s_{i-1} and s_i can be revealed, e.g., when B_i contains high-entropy data such as API keys, the above goal can be achieved using just 1 SHA-256 in ZKP. To do so, $\mathcal{P}$ computes $\pi = \text{ZK-PoK}\{B_i : f(s_{i-1}, B_i) = s_i\}$ and sends $(\pi, s_{i-1}, s_i, \boldsymbol{B}_{i-}, \boldsymbol{B}_{i+})$ to $\mathcal{V}$, who then 1) checks s_{i-1} by recomputing it from $\boldsymbol{B}_{i-}$; 2) verifies π; and 3) checks the MAC tag σ by recomputing it from s_i and $\boldsymbol{B}_{i+}$. Assuming B_i is high entropy, revealing s_{i-1} and s_i doesn't leak B_i since f is one-way.

On the other hand, if both s_{i-1} and s_i cannot be revealed to $\mathcal{V}$ (e.g., when brute-force attacks against B_i is feasible), we can still reduce the cost by having $\mathcal{P}$ redact a prefix (or suffix) of the record containing the block B_i. The cost incurred then is $256 - i$ SHA-2 hashes in ZKP. We relegate the details to App. A.2. Generally ZKP cost is proportional to record sizes so TLS fragmentation can also lower the cost by a constant factor.

5.1.2 GCM. Unlike CBC-HMAC, revealing a block is very efficient in GCM. First, $\mathcal{P}$ reveals $\mathsf{AES}(\mathsf{k}, IV)$ and $\mathsf{AES}(\mathsf{k}, 0)$, with proofs of correctness in ZK, to allow $\mathcal{V}$ to verify the integrity of the ciphertext. Then, to reveal the ith block, $\mathcal{P}$ just reveals the encryption of the ith counter $C_i = \mathsf{AES}(\mathsf{k}, \mathsf{inc}^i(IV))$ with a correctness proof. $\mathcal{V}$ can decrypt the ith block as $\hat{B}_i \oplus C_i$. IV is the public initial vector for the session, and $\mathsf{inc}^i(IV)$ denotes incrementing IV for i times (the exact format of inc is immaterial.) To reveal a TLS record, $\mathcal{P}$ repeat the above protocol for each block. We defer details to App. B.3.

5.2 Context integrity by two-stage parsing

For many applications, the verifier $\mathcal{V}$ may need to verify that the revealed substring appears in the right context. We refer to this property as *context integrity*. In this section we present techniques for $\mathcal{V}$ to specify contexts and for $\mathcal{P}$ to prove context integrity efficiently.

For ease of exposition, our description below focuses on the revealing mode, i.e., $\mathcal{P}$ reveals a substring of the server's response to $\mathcal{V}$. We discuss how redaction works in Sec. 5.2.3.

5.2.1 Specification of contexts. Our techniques for specifying contexts assume that the TLS-protected data sent to and from a given server $\mathcal{S}$ has a well-defined context-free grammar $\mathcal{G}$, known to both $\mathcal{P}$ and $\mathcal{V}$. In a slight abuse of notation, we let $\mathcal{G}$ denote both a grammar and the language it specifies. Thus, $R \in \mathcal{G}$ denotes a string R in the language given by $\mathcal{G}$. We assume that $\mathcal{G}$ is *unambiguous*, i.e., every $R \in \mathcal{G}$ has a unique associated parse-tree T_R. JSON and HTML are examples of two widely used languages that satisfy these requirements, and are our focus here.

When $\mathcal{P}$ then presents a substring R_{open} of some response R from $\mathcal{S}$, we say that R_{open} has *context integrity* if R_{open} is produced in a certain way expected by $\mathcal{V}$. Specifically, $\mathcal{V}$ specifies a set S of positions in which she might expect to see a valid substring R_{open} in R. In our definition, S is a set of paths from the root in a parse-tree defined by $\mathcal{G}$ to internal nodes. Thus $s \in S$, which we call a *permissible path*, is a sequence of non-terminals. Let ρ_R denote the root of T_R (the parse-tree of R in $\mathcal{G}$). We say that a string R_{open} has context-integrity with respect to (R,S) if T_R has a subtree whose leaves *yield* (i.e. concatenate to form) the string R_{open}, and that there is a path $s \in S$ from ρ_R to the root of the said subtree.

Formally, we define context integrity in terms of a predicate $\mathsf{CTX}_{\mathcal{G}}$ in Def. 5.1. At a high level, our definition is reminiscent of the production-induced context in [67].

Definition 5.1. Given a grammar $\mathcal{G}$ on TLS responses, $R \in \mathcal{G}$, a substring R_{open} of R, a set S of permissible paths, we define a *context function* $\mathsf{CTX}_{\mathcal{G}}$ as a boolean function such that $\mathsf{CTX}_{\mathcal{G}} : (S,R,R_{\text{open}}) \mapsto \text{true}$ iff $\exists$ a sub-tree $T_{R_{\text{open}}}$ of T_R with a path $s \in S$ from ρ_{T_R} to $\rho_{T_{R_{\text{open}}}}$ and $T_{R_{\text{open}}}$ yields R_{open}. R_{open} is said to have *context integrity* with respect to (R,S) if $\mathsf{CTX}_{\mathcal{G}}(S,R,R_{\text{open}}) = \text{true}$.

As an example, consider the JSON string J in fig. 3. JSON contains (roughly) the following rules:

```
Start → object          object → { pairs }
pair → "key" : value    pairs → pair | pair, pairs
key → chars             value → chars | object
```

In that example, $\mathcal{V}$ was interested in learning the derivation of the `pair` $p_{\texttt{balance}}$ with `key "balance"` in the `object` given by the `value` of the `pair` $p_{\texttt{checking}}$ with `key "checking a/c"`. Each of these non-terminals is the label for a node in the parse-tree T_J. The path from the root `Start` of T_J to $p_{\texttt{checking}}$ requires traversing a sequence of nodes of the form `Start → object → pairs* →` $p_{\texttt{checking}}$, where `pairs*` denotes a sequence of zero or more `pairs`. So S is the set of such sequences and R_{open} is the string `"checking a/c": {"balance": $2000}`.

5.2.2 Two-stage parsing. Generally, proving R_{open} has context integrity, i.e., $\mathsf{CTX}_{\mathcal{G}}(S,R,R_{\text{open}}) = \text{true}$, without directly revealing R would be expensive, since computing $\mathsf{CTX}_{\mathcal{G}}$ may require computing T_R for a potentially long string R. However, we observed that under certain assumptions that TLS-protected data generally satisfies, much of the overhead can be removed by having $\mathcal{P}$ *pre-process* R by applying a transformation Trans agreed upon by $\mathcal{P}$ and $\mathcal{V}$, and prove that R_{open} has context integrity with respect to R' (a usually much shorter string) and S' (a set of permissible paths specified by $\mathcal{V}$ based on S and Trans).

Based on this observation, we introduce a *two-stage parsing scheme* for efficiently computing R_{open} and proving $\mathsf{CTX}_{\mathcal{G}}(S,R,R_{\text{open}}) = \text{true}$. Suppose $\mathcal{P}$ and $\mathcal{V}$ agree upon $\mathcal{G}$, the grammar used by the web server, and a transformation Trans. Let $\mathcal{G}'$ be the grammar of strings $\mathsf{Trans}(R)$ for all $R \in \mathcal{G}$. Based on Trans, $\mathcal{V}$ specifies permissible paths S' and a constraint-checking function $\mathsf{cons}_{\mathcal{G},\mathcal{G}'}$. In the first stage, $\mathcal{P}$: (1) computes a substring R_{open} of R by parsing R (such that $\mathsf{CTX}_{\mathcal{G}}(S,R,R_{\text{open}}) = \text{true}$) (2) computes another string $R' = \mathsf{Trans}(R)$. In the second stage, $\mathcal{P}$ proves to $\mathcal{V}$ in zero-knowledge that (1) $\mathsf{cons}_{\mathcal{G},\mathcal{G}'}(R,R') = \text{true}$ and (2) $\mathsf{CTX}_{\mathcal{G}'}(S',R',R_{\text{open}}) = \text{true}$. Note that in addition to public parameters $\mathcal{G},\mathcal{G}',S,S',\mathsf{Trans},\mathsf{cons}_{\mathcal{G},\mathcal{G}'}$, the verifier only sees a commitment to R, and finally, R_{open}.

This protocol makes the zero-knowledge computation significantly less expensive by deferring actual parsing to a non-verifiable computation. In other words, the computation of $\mathsf{CTX}_{\mathcal{G}'}(S',R',R_{\text{open}})$ and $\mathsf{cons}_{\mathcal{G},\mathcal{G}'}(R,R')$ can be much more efficient than that of $\mathsf{CTX}_{\mathcal{G}}(S,R,R_{\text{open}})$.

We formalize the correctness condition for the two-stage parsing in an operational semantics rule in Def. 5.2. Here, $\langle f,\sigma\rangle$ denotes applying a function f on input σ, while $\frac{P}{C}$ denotes that if the premise P is true, then the conclusion C is true.

Definition 5.2. Given a grammar $\mathcal{G}$, a context function and permissible paths $\mathsf{CTX}_{\mathcal{G}}(S,\cdot,\cdot)$, a transformation Trans, a grammar $\mathcal{G}' = \{R' : R' = \mathsf{Trans}(R), R \in \mathcal{G}\}$ with context function and permissible paths $\mathsf{CTX}_{\mathcal{G}'}(S',\cdot,\cdot)$ and a function $\mathsf{cons}_{\mathcal{G},\mathcal{G}'}$, we say $(\mathsf{cons}_{\mathcal{G},\mathcal{G}'},S')$ are correct w.r.t. S, if for all (R,R',R_{open}) such that $R \in \mathcal{G}$, booleans b the following rule holds:

$$\frac{\langle \mathsf{cons}_{\mathcal{G},\mathcal{G}'},(R,R')\rangle \Rightarrow \text{true} \quad \langle \mathsf{CTX}_{\mathcal{G}'},(S',R',R_{\text{open}})\rangle \Rightarrow \text{b}}{\langle \mathsf{CTX}_{\mathcal{G}},(S,R,R_{\text{open}})\rangle \Rightarrow \text{b}}.$$

Below, we focus on a grammar that most DECO applications use, and present concrete constructions of two-stage parsing schemes.

5.2.3 DECO focus: Key-value grammars. A broad class of data formats, such as JSON, have a notion of key-value pairs. Thus, they are our focus in the current version of DECO.

A key-value grammar $\mathcal{G}$ produces key-value pairs according to the rule, "`pair → start key middle value end`", where `start`, `middle` and `end` are delimitors. For such grammars, an array of optimizations can greatly reduce the complexity for proving context. We discuss a few such optimizations below, with formal specification relegated to App. F.

Revelation for a globally unique key. For a key-value grammar $\mathcal{G}$, set of paths S, if for an $R \in \mathcal{G}$, a substring R_{open} satisfying context-integrity requires that R_{open} is parsed as a key-value pair with a globally unique key `K` (formally defined in App. F.4), R_{open} simply needs to be a substring of R and correctly be parsed as a `pair`. Specifically, $\mathsf{Trans}(R)$ outputs a substring R' of R containing the desired key, i.e., a substring of the form "`start K middle value end`" and $\mathcal{P}$ can output $R_{\text{open}} = R'$. $\mathcal{G}'$ can be defined by the rule $\mathtt{S}_{\mathcal{G}'}$ → `pair` where $\mathtt{S}_{\mathcal{G}'}$ is the start symbol in the production rules for $\mathcal{G}'$. Then (1) $\mathsf{cons}_{\mathcal{G},\mathcal{G}'}(R,R')$ checks that R' is a substring of R and (2) for $S' = \{\mathtt{S}_{\mathcal{G}'}\}$, $\mathsf{CTX}_{\mathcal{G}'}(S',R',R_{\text{open}})$ checks that (a) $R' \in \mathcal{G}'$ and (b) $R_{\text{open}} = R'$. Globally unique keys arise in Sec. 6.2 when selectively opening the response for `age`.

Redaction in key-value grammars. Thus far, our description of two-stage parsing assumes the Reveal mode in which $\mathcal{P}$ reveals a substring R_{open} of R to $\mathcal{V}$ and proves that R_{open} has context integrity with respect to the set of permissible paths specified by $\mathcal{V}$. In the Redact mode, the process is similar, but instead of revealing R_{open} in the clear, $\mathcal{P}$ generates a commitment to R_{open} using techniques from Sec. 5.1 and reveals R, with R_{open} removed, for e.g. by replacing its position with a dummy character.

6 APPLICATIONS

DECO can be used for any oracle-based application. To showcase its versatility, we have implemented and evaluated three applications that leverage its various capabilities: 1) a confidential financial instrument realized by smart contracts; 2) converting legacy credentials to anonymous credentials; and 3) privacy-preserving price discrimination reporting. Due to lack of space, we only present concrete implementation details for the first application, and refer readers to App. E for others. Evaluation results are presented in Sec. 7.2.

6.1 Confidential financial instruments

Financial derivatives are among the most commonly cited smart contract applications [32, 60], and exemplify the need for authenticated data feeds (e.g., stock prices). For example, one popular financial instrument that is easy to implement in a smart contract is a *binary option* [9]. This is a contract between two parties betting on whether, at a designated future time, e.g., the close of day D, the price P^* of some asset N will equal or exceed a predetermined target price P, i.e., $\mathsf{P}^* \geq \mathsf{P}$. A smart contract implementing this binary option can call an oracle $\mathcal{O}$ to determine the outcome.

In principle, $\mathcal{O}$ can conceal the underlying asset N and target price P for a binary option on chain. It simply accepts the option details off chain, and reports only a bit specifying the outcome $\mathsf{Stmt} := \mathsf{P}^* \geq ?\ \mathsf{P}$. This approach is introduced in [48], where it is referred to as a *Mixicle*.

A limitation of a basic Mixicle construction is that $\mathcal{O}$ itself learns the details of the financial instrument. Prior to DECO, only oracle services that use TEE (e.g., [78]) could conceal queries from $\mathcal{O}$. We now show how DECO can support execution of the binary option *without* $\mathcal{O}$ *learning the details of the financial instrument, i.e.,* N *or* P[1].

The idea is that the option winner plays the role of $\mathcal{P}$, and obtains a signed result of Stmt from $\mathcal{O}$, which plays the role of $\mathcal{V}$. We now describe the protocol and its implementation.

Protocol. Let $\{\mathsf{sk}_{\mathcal{O}}, \mathsf{pk}_{\mathcal{O}}\}$ denote the oracles' key pair. In our scheme, a binary option is specified by an asset name N, threshold price P, and settlement date D. We denote the commitment of a message M by $C_M = \mathsf{com}(M, r_M)$ with a witness r_M. Figure 5 shows the workflow steps in a confidential binary option:

1) Setup: Alice and Bob agree on the binary option $\{\mathsf{N}, \mathsf{P}, \mathsf{D}\}$ and create a smart contract $\mathcal{SC}$ with identifier $\mathsf{ID}_{\mathcal{SC}}$, The contract contains $\mathsf{pk}_{\mathcal{O}}$, addresses of the parties, and commitments to the option $\{C_{\mathsf{N}}, C_{\mathsf{P}}, C_{\mathsf{D}}\}$ with witnesses known to both parties. They also agree on public parameters θ_p (e.g., the URL to retrieve asset prices).

[1]The predicate direction $\geq ?$ or $\leq ?$ can be randomized. Concealing winner and loser identities and payment amounts is discussed in [48]. Additional steps can be taken to conceal other metadata, e.g., the exact settlement time.

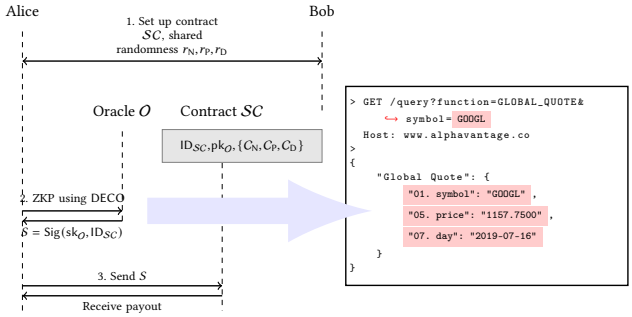

Figure 5: Two parties Alice and Bob execute a confidential binary option. Alice uses DECO **to access a stock price API and convince $\mathcal{O}$ she has won. Examples of request and response are shown to the right. Text in red is sensitive information to be redacted.**

2) Settlement: Suppose Alice wins the bet. To claim the payout, she uses DECO to generate a ZK proof that the current asset price retrieved matches her position. Alice and $\mathcal{O}$ execute the DECO protocol (with $\mathcal{O}$ acting as the verifier) to retrieve the asset price from θ_p (the target URL). We assume the response contains $(\mathsf{N}^*, \mathsf{P}^*, \mathsf{D}^*)$. In addition to the ZK proof in DECO to prove origin θ_p, Alice proves knowledge of $(\mathsf{P}, \mathsf{N}^*, \mathsf{P}^*, \mathsf{D}^*, r_{\mathsf{N}}, r_{\mathsf{P}}, r_{\mathsf{D}})$ such that $(\mathsf{P} \leq \mathsf{P}^*) \wedge C_{\mathsf{N}} = \mathsf{com}(\mathsf{N}^*, r_{\mathsf{N}}) \wedge C_{\mathsf{P}} = \mathsf{com}(\mathsf{P}, r_{\mathsf{P}}) \wedge C_{\mathsf{D}} = \mathsf{com}(\mathsf{D}^*, r_{\mathsf{D}})$.

Upon successful proof verification, the oracle returns a signed statement with the contract ID, $S = \mathsf{Sig}(\mathsf{sk}_{\mathcal{O}}, \mathsf{ID}_{\mathcal{SC}})$.

3) Payout: Alice provides the signed statement S to the contract, which verifies the signature and pays the winning party.

Alice and Bob need to trust $\mathcal{O}$ for integrity, but not for privacy. They can further hedge against integrity failure by using multiple oracles, as explained in Sec. 3.1. Decentralizing trust over oracles is a standard and already deployed technique [39]. We emphasize that DECO ensures privacy even if all the oracles are malicious.

Implementation details. Figure 5 shows the request and response of a stock price API. Let $\hat{R}$ and R denote the response ciphertext and the plaintext respectively. To settle an option, $\mathcal{P}$ proves to $\mathcal{V}$ that R contains evidence that he won the option, using the two-stage parsing scheme introduced in Sec. 5.2. In the first stage, $\mathcal{P}$ parses R locally and identifies the smallest substring of R that can convince $\mathcal{V}$. E.g., for stock prices, $R_{\text{price}} = \texttt{"05. price": "1157.7500"}$ suffices. In the second stage, $\mathcal{P}$ proves knowledge of $(R_{\text{price}}, \mathsf{P}, r_{\mathsf{P}})$ in ZK such that 1) R_{price} is a substring of the decryption of $\hat{R}$; 2) R_{price} starts with `"05. price"`; 3) the subsequent characters form a floating point number P^* and that $\mathsf{P}^* \geq \mathsf{P}$; 4) $\mathsf{com}(\mathsf{P}, r_{\mathsf{P}}) = C_{\mathsf{P}}$.

This two-stage parsing is secure assuming the keys are unique and the key `"05. price"` is followed by the price, making the grammar of this response a *key-value grammar with unique keys*, as discussed in Sec. 5.2. Similarly, $\mathcal{P}$ proves that the stock name and date in R match the commitments. With the CBC-HMAC ciphersuite, the zero-knowledge proof circuit involves redacting an entire record (408 bytes), computing commitments, and string processing.

6.2 Legacy credentials to anonymous credentials: Age proof

User credentials are often inaccessible outside a service provider's environment. Some providers offer third-party API access via OAuth

tokens, but such tokens reveal user identifiers. DECO allows users holding credentials in existing systems (what we call *legacy credentials*) to prove statements about them to third parties (verifiers) *anonymously*. Thus, DECO is the first system that allows users to convert *any* web-based legacy credential into an anonymous credential without server-side support [65] or trusted hardware [78].

We showcase an example where a student proves her/his age is over 18 using credentials (demographic details) stored on a University website. A student can provide this proof of age to any third party, such as a state issuing a driver's license or a hospital seeking consent for a medical test. We implement this example using the AES-GCM cipher suite and two-stage parsing (See fig. 10) with optimizations based on unique keys as in Sec. 5.2.

6.3 Price discrimination

Price discrimination refers to selling the same product or service at different prices to different buyers. Ubiquitous consumer tracking enables online shopping and booking websites to employ sophisticated price discrimination [72], e.g., adjusting prices based on customer zip codes [47]. Price discrimination can lead to economic efficiency [59], and is thus widely permissible under existing laws.

In the U.S., however, the FTC forbids price discrimination if it results in competitive injury [40], while new privacy-focused laws in Europe, such as the GDPR, are bringing renewed focus to the legality of the practice [21]. Consumers in any case generally dislike being subjected to price discrimination. Currently, however, there is no trustworthy way for users to report online price discrimination.

DECO allows a buyer to make a verifiable claim about perceived price discrimination by proving the advertised price of a good is higher than a threshold, while hiding sensitive information such as name and address. We implement this example using the AES-GCM cipher suite for the TLS session and reveal 24 AES blocks containing necessary order details and the request URL (See fig. 11).

7 IMPLEMENTATION AND EVALUATION

In this section, we discuss implementation details and evaluation results for DECO and our three applications.

7.1 DECO protocols

We implemented the three-party handshake protocol (3P-HS) for TLS 1.2 and query execution protocols (2PC-HMAC and 2PC-GCM) in about 4700 lines of C++ code. We built a hand-optimized TLS-PRF circuit with total AND complexity of 779,213. We also used variants of the AES circuit from [11]. Our implementation uses Relic [13] for the Paillier cryptosystem and the EMP toolkit [74] for the maliciously secure 2PC protocol of [75].

We integrated the three-party handshake and 2PC-HMAC protocols with mbedTLS [14], a popular TLS implementation, to build an end-to-end system. 2PC-GCM can be integrated to TLS similarly with more engineering effort. We evaluated the performance of 2PC-GCM separately. The performance impact of integration should be negligible. We did not implement 3P-HS for TLS 1.3, but we conjecture the performance should be comparable to that for TLS 1.2, since the circuit complexity is similar (c.f. Sec. 4.1.2).

Evaluation. We evaluated the performance of DECO in both the LAN and WAN settings. Both the prover and verifier run on a `c5.2xlarge` AWS node with 8 vCPU cores and 16GB of RAM. We located the two nodes in the same region (but different availability zones) for the LAN setting, but in two distinct data centers (in Ohio and Oregon) in the WAN setting. The round-trip time between two nodes in the LAN and WAN is about 1ms and 67ms, respectively, and the bandwidth is about 1Gbps.

Table 1: Run time (in ms) of 3P-HS and query execution protocols.

		LAN		WAN	
		Online	Offline	Online	Offline
3P-Handshake	TLS 1.2 only	368.5 (0.6)	1668 (4)	2850 (20)	10290 (10)
2PC-HMAC	TLS 1.2 only	133.8 (0.5)	164.9 (0.4)	2520 (20)	3191 (8)
2PC-GCM (256B)	1.2 and 1.3	36.65 (0.02)	392 (8)	1208.5 (0.2)	12010 (70)
2PC-GCM (512B)	1.2 and 1.3	53.0 (0.5)	610 (10)	2345 (1)	12520 (70)
2PC-GCM (1KB)	1.2 and 1.3	101.9 (0.5)	830 (20)	4567 (4)	14300 (200)
2PC-GCM (2KB)	1.2 and 1.3	204.7 (0.9)	1480 (30)	9093.5 (0.9)	18500 (200)

Table 2: Costs of generating and verifying ZKPs in proof-generation phase of DECO for applications in Sec. 6.

	Binary Option	Age Proof	Price Discrimination
prover time	12.97 ± 0.04s	3.67 ± 0.02s	12.68 ± 0.02s
verifier time	0.01s	0.01s	0.05s
proof size	861B	574B	1722B
# constraints	617k	164k	535k
memory	1.78GB	0.69GB	0.92GB

Table 1 summarizes the runtime of DECO protocols during a TLS session. 50 samples were used to compute the mean and standard error of the mean (in parenthesis). The MPC protocol we used relies on offline preprocessing to improve performance. Since the offline phase is input- and target-independent, it can be done prior to the TLS session. Only the online phase is on the critical path.

As shown in table 1, DECO protocols are very efficient in the LAN setting. It takes 0.37 seconds to finish the three-party handshake. For query execution, 2PC-HMAC is efficient (0.13s per record) as it only involves one SHA-2 evaluation in 2PC, regardless of record size. 2PC-GCM is generally more expensive and the cost depends on the query length, as it involves 2PC-AES over the entire query. We evaluated its performance with queries ranging from 256B to 2KB, the typical sizes seen in HTTP GET requests [63]. In the LAN setting, the performance is efficient and comparable to 2PC-HMAC.

In the WAN setting, the runtime is dominated by the network latency because MPC involves many rounds of communication. Nonetheless, the performance is still acceptable, given that DECO is likely to see only periodic use for most applications we consider.

7.2 Proof generation

We instantiated zero-knowledge proofs with a standard proof system [18] in libsnark [5]. We have devised efficiently provable statement templates, but users of DECO need to adapt them to their specific applications. SNARK compilers enable such adaptation in a high-level language, concealing low-level details from developers. We used xjsnark [50] and its Java-like high-level language to build statement templates and libsnark compatible circuits.

Our rationale in choosing libsnark is its relatively mature tooling support. The proofs generated by libsnark are constant-size and very efficient to verify, the downside being the per-circuit trusted setup. With more effort, DECO can be adapted to use, e.g., Bulletproofs [25], which requires no trusted setup but has large proofs and verification time.

Evaluation. We measure five performance metrics for each example—prover time (the time to generate the proofs), verifier time (the time to verify proofs), proof size, number of arithmetic constraints in the circuit, and the peak memory usage during proof generation.

Table 2 summarizes the results. 50 samples were used to compute the mean and its standard error. Through the use of efficient statement templates and two-stage parsing, DECO achieves very practical prover performance. Since libsnark optimizes for low verification overhead, the verifier time is negligible. The number of constraints (and prover time) is highest for the binary option application due to the extra string parsing routines. We use multiple proofs in each application to reduce peak memory usage. For the most complex application, the memory usage is 1.78GB. As libsnark proofs are of a constant size 287B, the proof sizes shown are multiples of that.

7.3 End-to-end performance

DECO end-to-end performance depends on the available TLS ciphersuites, the size of private data, and the complexity of application-specific proofs. Here we present the end-to-end performance of the most complex application of the three we implemented—the binary option. It takes about 13.77s to finish the protocol, which includes the time taken to generate unforgeable commitments (0.50s), to run the first stage of two-stage parsing (0.30s), and to generate zero-knowledge proofs (12.97s). These numbers are computed in the LAN setting; in the WAN setting, MPC protocols are more time-consuming (5.37s), pushing the end-to-end time up to 18.64s.

In comparison, Town Crier uses TEEs to execute a similar application in about 0.6s [78, Table I], i.e., around 20x faster than DECO, *but with added trust assumptions*. Since DECO is likely to be used only periodically for most applications, its overhead in achieving cryptographic-strength security assurances seems reasonable.

8 LEGAL AND COMPLIANCE ISSUES

Although users can already retrieve their data from websites, DECO allows users to export the data *with integrity proofs* without their explicit approval or even awareness. We now briefly discuss the resulting legal and compliance considerations.

Critically, however, DECO *users cannot unilaterally export data* to a third party with integrity assurance, but rely on oracles as verifiers for this purpose. While DECO keeps user data private, oracles learn what websites and types of data a user accesses. Thus oracles can enforce appropriate data use, e.g., denying transactions that may result in copyright infringement.

Both users and oracles bear legal responsibility for the data they access. Recent case law on the Computer Fraud and Abuse Act (CFAA), however, shows a shift away from criminalization of web scraping [69], and federal courts have ruled that violating websites' terms of service is not a criminal act *per se* [46, 49]. Users and oracles that violate website terms of service, e.g., "click wrap" terms, instead risk *civil* penalties [15]. DECO compliance with a given site's terms of service is a site- and application-specific question.

Oracles have an incentive to establish themselves as trustworthy within smart-contract and other ecosystems. We expect that reputable oracles will provide users with menus of the particular attestations they issue and the target websites they permit, vetting these options to maximize security and minimize liability and perhaps informing or cooperating with target servers.

The legal, performance, and compliance implications of incorrect attestations based on incorrect (and potentially subverted) data are also important. Internet services today have complex, multi-site data dependencies, though, so these issues aren't specific to DECO. Oracle services already rely on multiple data sources to help ensure correctness [39]. Oracle services in general could ultimately spawn infrastructure like that for certificates, including online checking and revocation capabilities [56] and different tiers of security [19].

9 RELATED WORK

Application-layer data-provenance. Signing content at the application layer is a way to prove data provenance. For example, [31, 77] aim to retrofit signing capabilities into HTTP. Application-layer solutions, however, suffer from poor modularity and reusability, as they are application-specific. They also require application-layer key management, violating the principle of layer separation in that cryptographic keys are no longer confined to the TLS layer.

Cinderella [33] uses verifiable computation to convert X.509 certificates into other credential types. Its main drawback is that few users possess certificates. Open ID Connect [6] providers can issue signed claims about users. However, adoption is still sparse and claims are limited to basic info such as names and email addresses.

Server-facilitated TLS-layer solutions. Several proposed TLS-layer data-provenance proofs [22, 44, 65] require server-side modifications. TLS-N [65] is a TLS 1.3 extension that enables a server to sign the session using the existing PKI, and also supports chunk-level redaction for privacy. We refer readers to [65] and references therein for a survey of TLS-layer solutions. Server-facilitated solutions suffer from high adoption cost, as they involve modification to security-critical server code. Moreover, they only benefit users when server administrators are able to and choose to cooperate.

Smart contract oracles. Oracles [26, 39, 78] relay authenticated data from, e.g., websites, to smart contracts. TLSNotary [7], used by Provable [10], allows a third party auditor to attest to a TLS connection between a server and a client, but relies on deprecated TLS versions (1.1 or lower). Town Crier [78] is an oracle service that uses TEEs (e.g., Intel SGX) for publicly verifiable evidence of TLS sessions and privacy-preserving computation on session data. While flexible and efficient, it relies on TEEs, which some users may reject given recently reported vulnerabilities, e.g., [24].

Selective opening with context integrity. Selective opening, i.e., decrypting part of a ciphertext to a third party while proving its integrity, has been studied previously. Sanitizable signatures [16, 23, 55, 70] allow a signed document to be selectively revealed. TLS-N [65] allows "chunk-level" redacting of TLS records. These works, however, consider a weaker adversarial model than DECO. They fail to address the critical property of context integrity. DECO enforces proofs of context integrity in the rigorous sense of Sec. 5.2, using a novel two-stage parsing scheme that achieves efficiency by greatly reducing the length of the input to the zero-knowledge proof.

ACKNOWLEDGEMENTS

This work was funded by NSF grants CNS-1514163, CNS-1564102, CNS-1704615, and CNS-1933655, and ARO grant W911NF16-1-0145.

Personal financial interests: Ari Juels is a technical advisor to Chainlink Smartcontract LLC and Soluna.

REFERENCES

[1] [n.d.]. Age Checker. https://agechecker.net.
[2] [n.d.]. Best BGP Route Network Monitoring Solution | ThousandEyes. https://www.thousandeyes.com/solutions/bgp-and-route-monitoring
[3] [n.d.]. BGPmon | BGPmon. https://bgpmon.net
[4] [n.d.]. BGPStream. https://bgpstream.com
[5] [n.d.]. libsnark. https://github.com/scipr-lab/libsnark.
[6] [n.d.]. Open ID Connect. https://openid.net/connect
[7] [n.d.]. TLSNotary. https://tlsnotary.org/.
[8] 2014. Art. 20, GDPR, Right to data portability. https://gdpr-info.eu/art-20-gdpr/.
[9] 2019. Binary option. https://en.wikipedia.org/wiki/Binary_option.
[10] 2019. Provable blockchain oracle. http://provable.xyz.
[11] Aug 2019. (Bristol Format) Circuits of Basic Functions Suitable For MPC. https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html.
[12] John Adler, Ryan Berryhill, Andreas G. Veneris, Zissis Poulos, Neil Veira, and Anastasia Kastania. 2018. Astraea: A Decentralized Blockchain Oracle. In *IEEE iThings/GreenCom/CPSCom/SmartData*.
[13] D. F. Aranha and C. P. L. Gouvêa. [n.d.]. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.
[14] ARM. 2019. mbedTLS. https://github.com/ARMmbed/mbedtls.
[15] American Bar Association. [n.d.].
[16] Giuseppe Ateniese, Daniel H Chou, Breno De Medeiros, and Gene Tsudik. 2005. Sanitizable signatures. In *ESORICS*.
[17] Judit Bar-Ilan and Donald Beaver. 1989. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *ACM PODC*.
[18] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security*.
[19] Robert Biddle, Paul C Van Oorschot, Andrew S Patrick, Jennifer Sobey, and Tara Whalen. 2009. Browser interfaces and extended validation SSL certificates: an empirical study. In *ACM workshop on Cloud computing security*. 19–30.
[20] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. 2006. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. RFC 4492.
[21] Frederik Zuiderveen Borgesius and Joost Poort. 2017. Online price discrimination and EU data privacy law. *Journal of consumer policy* (2017).
[22] Mark Brown and Russ Housle. 2007. Transport Layer Security (TLS) Evidence Extensions. https://tools.ietf.org/html/draft-housley-evidence-extns-01.
[23] Christina Brzuska, Marc Fischlin, Tobias Freudenreich, Anja Lehmann, Marcus Page, Jakob Schelbert, Dominique Schröder, and Florian Volk. 2009. Security of sanitizable signatures revisited. In *PKC*. Springer.
[24] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.
[25] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *IEEE S&P*.
[26] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3 (2014), 37.
[27] Kevin R. B. Butler, Toni R. Farley, Patrick D. McDaniel, and Jennifer Rexford. 2010. A Survey of BGP Security Issues and Solutions. *Proc. IEEE* 98, 1 (2010), 100–122.
[28] Jan Camenisch and Markus Stadler. 1997. Efficient group signature schemes for large groups. In *Annual International Cryptology Conference*.
[29] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. 2017. Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services. In *ACM CCS*.
[30] Ran Canetti. 2000. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067. https://eprint.iacr.org/2000/067.
[31] Mark Cavage and Manu Sporny. 2019. *Signing HTTP Messages*. Internet-Draft draft-cavage-http-signatures-11.
[32] CFTC. 2018. A Primer on Smart Contrats. https://www.cftc.gov/sites/default/files/2018-11/LabCFTC_PrimerSmartContracts112718.pdf.
[33] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. 2016. Cinderella: Turning shabby X. 509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *IEEE S&P*.
[34] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2015. Automated synthesis of optimized circuits for secure computation. In *ACM CCS*.
[35] Shivani Deshpande, Marina Thottan, Tin Kam Ho, and Biplab Sikdar. 2009. An online mechanism for BGP instability detection and analysis. *IEEE transactions on Computers* 58, 11 (2009), 1470–1484.
[36] T. Dierks and E. Rescorla. 2008. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246.
[37] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. 2015. A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates. In *ACM Conference on Computer and Communications Security*. ACM, 1197–1210.
[38] Morris J Dworkin. 2007. *SP 800-38d. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC*. Technical Report.
[39] Steve Ellis, Ari Juels, and Sergey Nazarov. 4 Sept. 2017. ChainLink: A Decentralized Oracle Network. https://link.smartcontract.com/whitepaper.
[40] FTC. 2017. Price Discrimination: Robinson-Patman Violations. https://www.ftc.gov/tips-advice/competition-guidance/guide-antitrust-laws/price-discrimination-robinson-patman.
[41] Rosario Gennaro and Steven Goldfeder. 2018. Fast multiparty threshold ECDSA with fast trustless setup. In *ACM CCS*.
[42] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. 2017. Message Franking via Committing Authenticated Encryption. In *CRYPTO*.
[43] Dick Grune. 2010. *Parsing Techniques: A Practical Guide* (2nd ed.). Springer.
[44] Ibrahim Hajjeh and Mohamad Badra. 2017. TLS Sign. https://tools.ietf.org/html/draft-hajjeh-tls-sign-04.
[45] Carmit Hazay and Yehuda Lindell. 2010. A Note on Zero-Knowledge Proofs of Knowledge and the ZKPOK Ideal Functionality. Cryptology ePrint Archive, Report 2010/552. https://eprint.iacr.org/2010/552.
[46] Marcia Hofmann. 21 July 2010. Court: Violating Terms of Service Is Not a Crime, But Bypassing Technical Barriers Might Be. Electronic Frontier Foundation (EFF) News Update.
[47] Neil Howe. 2017. A Special Price Just for You. *Forbes* (Nov 2017). https://www.forbes.com/sites/neilhowe/2017/11/17/a-special-price-just-for-you/.
[48] Ari Juels, Lorenz Breidenbach, Alex Coventry, Sergey Nazarov, and Steve Ellis. 2019. Mixicles: Private Decentralized Finance Made Simple. Chainlink whitepaper.
[49] George Khoury. 24 Jan. 2018. Violation of a Website's Terms of Service is Not Criminal. Findlaw blog post.
[50] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. 2018. xJsnark: A Framework for Efficient Verifiable Computation. In *IEEE S&P*.
[51] Christopher Krügel, Darren Mutz, William K. Robertson, and Fredrik Valeur. 2003. Topology-Based Detection of Anomalous BGP Messages. In *RAID (Lecture Notes in Computer Science, Vol. 2820)*. Springer, 17–35.
[52] Yehida Lindell. 2005. Secure multiparty computation for privacy preserving data mining. In *Encyclopedia of Data Warehousing and Mining*. IGI Global.
[53] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. 2020. CanDID: Can-Do Decentralized Identity with Legacy Compatibility, Sybil-Resistance, and Accountability. Cryptology ePrint Archive, Report 2020/934. https://eprint.iacr.org/2020/934.
[54] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. 2018. DelegaTEE: Brokered Delegation Using Trusted Execution Environments. In *USENIX Security*.
[55] Kunihiko Miyazaki, Mitsuru Iwamura, Tsutomu Matsumoto, Ryôichi Sasaki, Hiroshi Yoshiura, Satoru Tezuka, and Hideki Imai. 2005. Digitally Signed Document Sanitizing Scheme with Disclosure Condition Control. *IEICE Transactions* (2005).
[56] Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle Adams. 1999. *X. 509 Internet public key infrastructure online certificate status protocol-OCSP*. Technical Report. RFC 2560.
[57] Helen Nissenbaum. 2009. *Privacy in context: Technology, policy, and the integrity of social life*. Stanford University Press.
[58] Goldreich Oded. 2009. *Foundations of Cryptography: Volume 2, Basic Applications* (1st ed.). Cambridge University Press.
[59] Andrew Odlyzko. 2003. Privacy, economics, and price discrimination on the Internet. In *5th international conference on Electronic commerce*.
[60] @OpenLawOfficial. 2018. The Future of Derivatives: An End-to-End, Legally Enforceable Option Contract Powered by Ethereum.
[61] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT*.
[62] Jack Peterson and Joseph Krug. 2015. Augur: a decentralized, open-source platform for prediction markets. *arXiv:1501.01042* (2015).
[63] The Chromium Projects. [n.d.]. The SPDY whitepaper. https://dev.chromium.org/spdy/spdy-whitepaper.
[64] E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446.
[65] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, and Srdjan Čapkun. 2018. TLS-N: Non-repudiation over TLS Enabling Ubiquitous Content Signing.. In *NDSS*.
[66] J. Salowey, A. Choudhury, and D. McGrew. 2008. *AES Galois Counter Mode (GCM) Cipher Suites for TLS*. RFC 5288.
[67] Prateek Saxena, David Molnar, and Benjamin Livshits. 2011. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *ACM CCS*.
[68] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. 2011. Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications. In *Financial Cryptography*.

[69] Andrew Sellars. 2018. Twenty Years of Web Scraping and the Computer Fraud and Abuse Act. *BUJ Sci. & Tech. L.* 24 (2018), 372.

[70] Ron Steinfeld, Laurence Bull, and Yuliang Zheng. 2001. Content extraction signatures. In *International Conference on Information Security and Cryptology*.

[71] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. 2015. RAPTOR: Routing Attacks on Privacy in Tor. In *USENIX Security*.

[72] Jerry Useem. 2017. How Online Shopping Makes Suckers of Us All. *The Atlantic* (Jul 2017). https://www.theatlantic.com/magazine/archive/2017/05/how-online-shopping-makes-suckers-of-us-all/521448/.

[73] L. Wang, G. Asharov, R. Pass, T. Ristenpart, and A. Shelat. 2019. Blind Certificate Authorities. In *IEEE S&P*.

[74] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[75] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation. In *ACM CCS*.

[76] Andrew Chi-Chih Yao. 1982. Protocols for secure computations. In *FOCS*.

[77] Jeffrey Yasskin. 2019. *Signed HTTP Exchanges*. Internet-Draft draft-yasskin-http-origin-signed-responses-05.

[78] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An authenticated data feed for smart contracts. In *ACM CCS*.

[79] Jian Zhang, Jennifer Rexford, and Joan Feigenbaum. 2005. Learning-based anomaly detection in BGP updates. In *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*. 219–220.

[80] Ke Zhang, Amy Yen, Xiaoliang Zhao, Dan Massey, S Felix Wu, and Lixia Zhang. 2004. On detection of anomalous routing dynamics in BGP. In *International Conference on Research in Networking*. Springer, 259–270.

A PROTOCOLS DETAILS

A.1 Formal specification

We gave a self-contained informal description of the three-party handshake protocol in Sec. 4.1. The formal specification is given in fig. 6 along with its building block ECtF in fig. 7. The post-handshake protocols for CBC-HMAC described in Sec. 4.2 is specified in fig. 8.

A.2 Selective opening (CBC-HMAC)

Redacting a suffix. When a suffix $\boldsymbol{B}_{i+}$ is to be redacted, $\mathcal{P}$ computes $\pi = \text{ZK-PoK}\{\boldsymbol{B}_{i+}, \mathsf{k}^{\mathsf{Enc}} : f(s_i, \boldsymbol{B}_{i+}) = ih \wedge H(\mathsf{k}^{\mathsf{MAC}} \oplus \text{opad} || ih) = \sigma \wedge B_{1025} \| B_{1026} \| B_{1027} = \text{CBC}(\mathsf{k}^{\mathsf{Enc}}, \sigma)\}$ and s_i is the state after applying f on $\boldsymbol{B}_{i-} \| B_i$. $\mathcal{P}$ sends $(\pi, \boldsymbol{B}_{i-} \| B_i)$ to $\mathcal{V}$. The verifier then 1) checks s_{i-1} by applying f on $\boldsymbol{B}_{i-} \| B_i$, and 2) verifies π. Essentially, the security of this follows from pre-image resistance of f. Moreover, $\mathcal{V}$ doesn't learn the redacted suffix since $ih = f(s, \boldsymbol{B}_{i+})$ is kept secret from $\mathcal{V}$. The total cost is 3 AES and $256 - i$ SHA-2 hashes in ZKP.

Redacting a prefix. $\mathcal{P}$ computes two ZKPs: 1) $\pi_1 = \text{ZK-PoK}\{\boldsymbol{B}_{i-}, \mathsf{k}^{\mathsf{MAC}} : H(\mathsf{k}^{\mathsf{MAC}} \oplus \text{ipad} || \boldsymbol{B}_{i-}) = s_{i-1}\}$; 2) $\pi_2 = \text{ZK-PoK}\{\mathsf{k}^{\mathsf{MAC}}, \mathsf{k}^{\mathsf{Enc}} : H(\mathsf{k}^{\mathsf{MAC}} \oplus \text{opad} || ih) = \sigma \wedge B_{1025} \| B_{1026} \| B_{1027} = \text{CBC}(\mathsf{k}^{\mathsf{Enc}}, \sigma)\}$. $\mathcal{P}$ sends $(\pi_1, \pi_2, s_{i-1}, B_i \| \boldsymbol{B}_{i+})$ to $\mathcal{V}$. The verifier checks that 1) s_{i-1} is correct using π_1 and then computes $f(s_{i-1}, B_i \| \boldsymbol{B}_{i+})$ to obtain the inner hash ih, 2) π_2 is verified using the computed ih. The cost incurred is 3 AES and $256 - i$ SHA-2 hashes in ZKP.

Note that redacting a prefix/suffix only makes sense if the revealed portion does not contain any private user data. Otherwise, $\mathcal{P}$ would have to find the smallest substring containing all the sensitive blocks and redact either the prefix/suffix similar to above.

B PROTOCOLS DETAILS FOR GCM

B.1 Preliminaries

GCM is an authenticated encryption with additional data (AEAD) cipher. To encrypt, the GCM cipher takes as inputs a tuple $(\mathsf{k}, IV, \boldsymbol{M}, \boldsymbol{A})$:

The three-party handshake (3P-HS) protocol among $\mathcal{P}$, $\mathcal{V}$ and $\mathcal{S}$

Public information: Let EC be the Elliptic Curve used in ECDHE over $\mathbb{F}_p$ with order p, G a parameter, and Y_S the server public key.

Output: $\mathcal{P}$ and $\mathcal{V}$ output $\mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}}$ and $\mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}}$ respectively, while the TLS server outputs $\mathsf{k}^{\mathsf{MAC}} = \mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}} + \mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}}$. Besides, both $\mathcal{S}$ and $\mathcal{P}$ outputs $\mathsf{k}^{\mathsf{Enc}}$.

TLS server $\mathcal{S}$: follow the standard TLS protocol.

Prover $\mathcal{P}$:

On initialization: $\mathcal{P}$ samples $r_c \leftarrow\$ \{0,1\}^{256}$ and sends ClientHello(r_c) to $\mathcal{S}$ to start a standard TLS handshake.

On receiving ServerHello(r_s), ServerKeyEx(Y,σ,cert) from $\mathcal{S}$:

- $\mathcal{P}$ verifies that cert is a valid certificate and that σ is a valid signature over (r_c, r_s, Y_S) signed by a key contained in cert. $\mathcal{P}$ sends $(r_c, r_s, Y_S, \sigma, \text{cert})$ to $\mathcal{V}$.
- $\mathcal{V}$ checks cert and σ similarly. $\mathcal{V}$ then samples $s_V \leftarrow\$ \mathbb{F}_p$ and computes $Y_V = s_V \cdot G$. Send Y_V to $\mathcal{P}$.
- $\mathcal{P}$ samples $s_P \leftarrow\$ \mathbb{F}_p$ and computes $Y_P = s_P \cdot G$. Send ClientKeyEx($Y_P + Y_V$) to $\mathcal{S}$.
- $\mathcal{P}$ and $\mathcal{V}$ run ECtF to compute a sharing of the x-coordinate of $Y_P + Y_V$, denoted z_P, z_V.
- $\mathcal{P}$ (and $\mathcal{V}$) send z_P (and z_V) to $\mathcal{F}_{\text{2PC}}^{\text{hs}}$ (specified below) to compute shares of session keys and the master secret. $\mathcal{P}$ receives $(\mathsf{k}^{\mathsf{Enc}}, \mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}}, m_{\mathcal{P}})$, while $\mathcal{V}$ receives $(\mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}}, m_{\mathcal{V}})$.
- $\mathcal{P}$ computes a hash (denoted h) of the handshake messages sent and received thus far, and runs 2PC-PRF with $\mathcal{V}$ to compute $s = \text{PRF}(m_{\mathcal{P}} \oplus m_{\mathcal{V}}, \text{"client finished"}, h)$ on the hash of the handshake messages and send a Finished(s) to $\mathcal{S}$.

On receiving other messages from $\mathcal{S}$:

- If it's Finished(s), $\mathcal{P}$ and $\mathcal{V}$ run a 2PC to check $s \stackrel{?}{=} \text{PRF}(m_{\mathcal{P}} \oplus m_{\mathcal{V}}, \text{"server finished"}, h)$ and abort if not.
- Otherwise respond according to the standard TLS protocol.

$\mathcal{F}_{\text{2PC}}^{\text{hs}}$ with $\mathcal{P}$ and $\mathcal{V}$

Public Input: nonce r_c, r_s

Private Input: $z_P \in \mathbb{F}_p$ from $\mathcal{P}$; $z_V \in \mathbb{F}_p^2$ from $\mathcal{V}$

- $z := z_P + z_v$
- $m := \text{PRF}(z, \text{"master secret"}, r_c \| r_s)$ (truncate at 48 bytes)
- $\mathsf{k}^{\mathsf{MAC}}, \mathsf{k}^{\mathsf{Enc}} := \text{PRF}(m, \text{"key expansion"}, r_s \| r_c)$ // key expansion
- Sample $r_k, r_m \leftarrow\$ \mathbb{F}_p$. Send $(\mathsf{k}^{\mathsf{Enc}}, r_k, r_m)$ to $\mathcal{P}$, and $(r_k \oplus \mathsf{k}^{\mathsf{MAC}}, r_m \oplus m)$ to $\mathcal{V}$ privately.

Figure 6: The protocol of three-party handshake.

a secret key, an initial vector, a plaintext of multiple AES blocks, and additional data to be included in the integrity protection; it outputs a ciphertext $\boldsymbol{C}$ and a tag T. Decryption reverses the process. The decryption cipher takes as input $(\mathsf{k}, IV, \boldsymbol{C}, \boldsymbol{A}, T)$ and first checks the integrity of the ciphertext by comparing a recomputed tag with T, then outputs the plaintext.

The ciphertext is computed in the counter mode: $C_i = \text{AES}(\mathsf{k}, \text{inc}^i(IV)) \oplus M_i$ where inc^i denotes incrementing IV for i times (the exact format of inc is immaterial.)

The tag $\text{Tag}(\mathsf{k}, IV, \boldsymbol{C}, \boldsymbol{A})$ is computed as follows. Given a vector $\boldsymbol{X} \in \mathbb{F}_{2^{128}}^m$, the associated GHASH polynomial $P_{\boldsymbol{X}} : \mathbb{F}_{2^{128}} \to \mathbb{F}_{2^{128}}$ is defined as $P_{\boldsymbol{X}}(h) = \sum_{i=1}^{m} X_i \cdot h^{m-i+1}$ with addition and multiplication done in $\mathbb{F}_{2^{128}}$. Without loss of generality, suppose $\boldsymbol{A}$ and $\boldsymbol{C}$ are properly padded. Let ℓ_A and ℓ_C denote their length. A GCM tag is

$$\text{Tag}(\mathsf{k}, IV, \boldsymbol{C}, \boldsymbol{A}) := \text{AES}(\mathsf{k}, IV) \oplus P_{\boldsymbol{A} \| \boldsymbol{C} \| \ell_A \| \ell_C}(h) \tag{1}$$

where $h = \text{AES}(\mathsf{k}, 0)$.

When GCM is used in TLS, each plaintext record D is encrypted as follows. A unique nonce n is chosen and the additional data κ is computed as a concatenation of the sequence number, version, and length of D. GCM encryption is invoked to generate the payload

ECtF between $\mathcal{P}$ and $\mathcal{V}$

Input: $P_1=(x_1,y_1)\in EC(\mathbb{F}_p)$ from $\mathcal{P}$, $P_2=(x_2,y_2)\in EC(\mathbb{F}_p)$ from $\mathcal{V}$.

Output: $\mathcal{P}$ and $\mathcal{V}$ output s_1 and s_2 such that $s_1+s_2=x$ where $(x,y)=P_1+P_2$ in EC.

Protocol:

- $\mathcal{P}$ (and $\mathcal{V}$) sample $\rho_i \leftarrow\$\mathbb{Z}_p$ for $i\in\{1,2\}$ respectively. $\mathcal{P}$ and $\mathcal{V}$ run $\alpha_1,\alpha_2 := \mathsf{MtA}((-x_1,\rho_1),(\rho_2,x_2))$.
- $\mathcal{P}$ computes $\delta_1=-x_1\rho_1+\alpha_1$ and $\mathcal{V}$ computes $\delta_2=x_2\rho_2+\alpha_2$.
- $\mathcal{P}$ (and $\mathcal{V}$) reveal δ_1 (and δ_2) to each other and compute $\delta=\delta_1+\delta_2$.
- $\mathcal{P}$ (and $\mathcal{V}$) compute $\eta_i=\rho_i\cdot\delta^{-1}$ for $i\in\{1,2\}$ respectively.
- $\mathcal{P}$ and $\mathcal{V}$ run $\beta_1,\beta_2 := \mathsf{MtA}((-y_1,\eta_1),(\eta_2,y_2))$.
- $\mathcal{P}$ computes $\lambda_1=-y_1\cdot\eta_1+\beta_1$ and $\mathcal{V}$ computes $\lambda_2=y_2\cdot\eta_2+\beta_2$. They run $\gamma_1,\gamma_2 := \mathsf{MtA}(\lambda_1,\lambda_2)$.
- $\mathcal{P}$ (and $\mathcal{V}$) computes $s_i=2\gamma_i+\lambda_i^2-x_i$ for $i\in\{1,2\}$ respectively.
- $\mathcal{P}$ outputs s_1 and $\mathcal{V}$ outputs s_2.

Figure 7: (ECtF) A protocol for converting shares of EC points in $EC(\mathbb{F})$ to shares of coordinates in $\mathbb{F}$.

2PC-HMAC between $\mathcal{P}$ and $\mathcal{V}$

Input: $\mathcal{P}$ inputs $\mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}}$, m and $\mathcal{V}$ inputs $\mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}}$.

Output: $\mathcal{P}$ outputs $\mathsf{HMAC}(\mathsf{k}^{\mathsf{MAC}},m)$ where $\mathsf{k}^{\mathsf{MAC}}=\mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}}\oplus\mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}}$.

One-time setup: $\mathcal{P}$ and $\mathcal{V}$ use 2PC to compute $s_0=f(\mathsf{IV},\mathsf{k}^{\mathsf{MAC}}\oplus\mathsf{ipad})$ and reveal s_0 to $\mathcal{P}$.

To compute a tag for message m:

- $\mathcal{P}$ computes inner hash $h_i=f(s_0,m)$.
- $\mathcal{P}$ inputs $\mathsf{k}_{\mathcal{P}}^{\mathsf{MAC}}$, h_i and $\mathcal{V}$ inputs $\mathsf{k}_{\mathcal{V}}^{\mathsf{MAC}}$ to 2PC which reveals $\mathsf{H}(\mathsf{k}^{\mathsf{MAC}}\oplus\mathsf{opad}\|h_i)$ to both parties.

Figure 8: The 2PC-HMAC protocol. f denotes the compression function of the hash function H and IV denotes the initial value.

record as $M=n\|\mathsf{GCM}(\mathsf{k},n,D,\kappa)$. We refer readers to [38] for a complete specification.

B.2 Query execution

The 2PC protocols for verifying tags and decrypting records are specified in fig. 9.

Tag creation/verification. Computing or verifying a GCM tag involves evaluating eq. (1) in 2PC. A challenge is that eq. (1) involves both arithmetic computation (e.g., polynomial evaluation in $\mathbb{F}_{2^{128}}$) as well as binary computation (e.g., AES). Performing multiplication in a large field in a binary circuit is expensive, while computing AES (defined in $\mathsf{GF}(2^8)$) in $\mathbb{F}_{2^{128}}$ incurs high overhead. Even if the computation could somehow separated into two circuits, evaluating the polynomial alone—which takes approximately 1,000 multiplications in $\mathbb{F}_{2^{128}}$ for *each* record—would be unduly expensive.

Our protocol removes the need for polynomial evaluation. The actual 2PC protocol involves only binary operations and thus can be done in a single circuit. Moreover, the per-record computation is reduced to only one invocation of 2PC-AES.

The idea is to compute shares of $\{h^i\}$ (in a 2PC protocol) in a preprocessing phase at the beginning of a session. The overhead of preprocessing is amortized over the session because the same h used for all records that follow. With shares of $\{h^i\}$, $\mathcal{P}$ and $\mathcal{V}$ can compute shares of a polynomial evaluation $P_{\boldsymbol{A}\|\boldsymbol{C}\|\ell_A\|\ell_C}(h)$ locally. They also compute $\mathsf{AES}(\mathsf{k},IV)$ in 2PC to get a share of $\mathsf{Tag}(\mathsf{k},IV,\boldsymbol{C},\boldsymbol{A})$. In total, only one invocation of 2PC-AES in needed to check the tag for each record.

Post-handshake protocols for GCM

Private input: $\mathsf{k}_{\mathcal{P}}$ and $\mathsf{k}_{\mathcal{V}}$ from $\mathcal{P}$ and $\mathcal{V}$ respectively. $\mathsf{k}=\mathsf{k}_{\mathcal{P}}+\mathsf{k}_{\mathcal{V}}$ is the encryption key.

Protocol for preprocessing

On initialization: $\mathcal{P}$ (and $\mathcal{V}$) sends $\mathsf{k}_{\mathcal{P}}$ and ($\mathsf{k}_{\mathcal{V}}$) to $\mathcal{F}_{\mathrm{PP}}$ and wait for output $\{h_{\mathcal{P},i}\}_i$ (and $\{h_{\mathcal{V},i}\}_i$).

$\mathcal{F}_{\mathrm{PP}}$

After receiving $\mathsf{k}_1,\mathsf{k}_2$ from two parties, compute $h:=\mathsf{AES}(\mathsf{k}_1+\mathsf{k}_2,0)$. Sample n random numbers $\{r_i\}_{i=1}^n$ and compute $\{h^i\}_{i=1}^n$ in $\mathbb{F}_{2^{128}}$. For $i\in[n]$, send r_i to player 1 and $r_i\oplus h^i$ to player 2.

Protocol for decrypting TLS records

Prover $\mathcal{P}$:

On receiving a record $(IV,\boldsymbol{C},\boldsymbol{A},T)$ from $\mathcal{S}$:

- Let $\boldsymbol{X}=\boldsymbol{A}\|\boldsymbol{C}\|\ell_A\|\ell_C$.
- Send $(\mathsf{k}_{\mathcal{P}},IV)$ to $\mathcal{F}_{\text{AES-EqM}}$ and wait for output $c_{\mathcal{P}}$.
- Send $(IV,\boldsymbol{X})$ to $\mathcal{V}$ and wait for the response P.
- Compute $T'=P+c_{\mathcal{P}}+\sum_i X_i\cdot h_{\mathcal{P},i}$ in $\mathbb{F}_{2^{128}}$.
- Abort if $T'\neq T$. Otherwise, compute $\boldsymbol{K}$ such that $K_i=\mathsf{inc}^i(IV)$ for $i\in[\ell_C]$. Send $(IV,\ell_C,\text{Decrypt})$ to $\mathcal{V}$.
- Send $(\mathsf{k}_{\mathcal{P}},\boldsymbol{K})$ to $\mathcal{F}_{\text{AES-EqM-Asym}}$ as party 1 and wait for output $\boldsymbol{K'}$.
- Decrypt the message as $M_i=K'_i\oplus C_i$.

Verifier $\mathcal{V}$:

On receiving $(IV,\boldsymbol{X})$ from $\mathcal{P}$:

- If IV found in store, abort. Otherwise store IV and proceed.
- Send $(\mathsf{k}_{\mathcal{V}},IV)$ to $\mathcal{F}_{\text{AES-EqM}}$ and wait for output $c_{\mathcal{V}}$.
- Compute $P=c_{\mathcal{V}}+\sum_i X_i\cdot h_{\mathcal{V},i}$ in $\mathbb{F}_{2^{128}}$
- Send P to $\mathcal{P}$.

On receiving $(IV,n,\text{Decrypt})$ from $\mathcal{P}$:

- Compute $\boldsymbol{K}$ such that $K_i=\mathsf{inc}^i(IV)$ for $i\in[n]$.
- Abort if any K_i is found in store (as previously used IVs.)
- Send $(\mathsf{k}_{\mathcal{V}},\boldsymbol{K})$ to $\mathcal{F}_{\text{AES-EqM-Asym}}$ as party 2.

$\mathcal{F}_{\text{AES-EqM}}$

Wait for input (k_i,m_i) from party i for $i\in\{1,2\}$. Abort if $m_1\neq m_2$. Sample $r\leftarrow\$\mathbb{F}$. Compute $c=\mathsf{AES}(\mathsf{k}_1\oplus\mathsf{k}_2,m_1)$. Send r to party 1 and $c\oplus r$ to party 2.

$\mathcal{F}_{\text{AES-EqM-Asym}}$

Wait for input (k_i,m_i) from party i for $i\in\{1,2\}$. Abort if $m_1\neq m_2$. Compute $c=\mathsf{AES}(\mathsf{k}_1\oplus\mathsf{k}_2,m_1)$. Send c to party 1 and $\perp$ to party 2.

Figure 9: The post-handshake protocols for AES-GCM.

It is critical that $\mathcal{V}$ never responds to the same IV more than once; otherwise $\mathcal{P}$ would learn h. Specifically, in each response, $\mathcal{V}$ reveals a blinded linear combination of her shares $\{h_{\mathcal{V},i}\}$ in the form of $\mathcal{L}_{IV,X}=\mathsf{AES}(\mathsf{k},IV)\oplus\sum_i X_i\cdot h_{\mathcal{V},i}$. It is important that the value is blinded by $\mathsf{AES}(\mathsf{k},IV)$ because a single unblinded linear combination of $\{h_{\mathcal{V},i}\}$ would allow $\mathcal{P}$ to solve for h. Therefore, if $\mathcal{V}$ responds to the same IV twice, the blinding can be removed by adding the two responses (in $\mathbb{F}_{2^{128}}$): $\mathcal{L}_{IV,X}\oplus\mathcal{L}_{IV,X'}=\sum_i(X_i+X'_i)\cdot h_{\mathcal{V},i}$. This follows from the nonce uniqueness requirement of GCM [66].

Encrypting/decrypting records. Once tags are properly checked, decryption of records is straightforward. $\mathcal{P}$ and $\mathcal{V}$ simply compute AES encryption of $\mathsf{inc}^i(IV)$ with 2PC-AES. A subtlety to note is that $\mathcal{V}$ must check that the counters to be encrypted have *not* been used as IV previously. Otherwise $\mathcal{P}$ would learn h to $\mathcal{P}$ in a manner like that outlined above.

B.3 Proof Generation

Revealing a block. $\mathcal{P}$ wants to convince $\mathcal{V}$ that an AES block B_i is the ith block in the encrypted record rêc. The proof strategy is as follows: 1) prove that AES block B_i encrypts to the ciphertext block $\hat{B}_i$ and 2) prove that the tag is correct. Proving the correct encryption requires only 1 AES in ZKP. Naïvely done, proving the correct tag incurs evaluating the GHASH polynomial of degree 512 and 2 AES block encryptions in ZKP.

We manage to achieve a much more efficient proof by allowing $\mathcal{P}$ to reveal two encrypted messages AES(k,IV) and AES(k,0) to $\mathcal{V}$, thus allowing $\mathcal{V}$ to verify the tag (see eq. (1)). $\mathcal{P}$ only needs to prove the correctness of encryption in ZK and that the key used corresponds to the commitment, requiring 2 AES and 1 SHA-2 ($\mathcal{P}$ commits to $k_{\mathcal{P}}$ by revealing a hash of the key). Thus, the total cost is 3 AES and 1 SHA-2 in ZKP.

Revealing a TLS record. The proof techniques are a simple extension from the above case. $\mathcal{P}$ reveals the entire record rec and proves correct AES encryption of all the AES blocks, resulting in a total 514 AES and 1 SHA-2 in ZKP.

Revealing a TLS record except for a block. Similar to the above case, $\mathcal{P}$ proves encryption of all the blocks in the record except one, resulting in a total 513 AES and 1 SHA-2 in ZKP.

C PROTOCOL EXTENSIONS

C.1 Adapting to support TLS 1.3

To support TLS 1.3, the 3P-HS protocol must be adapted to a new handshake flow and a different key derivation circuit. Notably, all handshake messages after the ServerHello are now *encrypted*. A naïve strategy would be to decrypt them in 2PC, which would be costly as certificates are usually large. However, thanks to the key independence property of TLS 1.3 [37], $\mathcal{P}$ and $\mathcal{V}$ can securely reveal the handshake encryption keys without affecting the secrecy of final session keys [37]. Handshake integrity is preserved because the Finished message authenticates the handshake using yet another independent key. (In fact [37, §3.1] argues that the signatures already authenticate the handshake.)

Therefore the optimized 3P-HS work as follows. $\mathcal{P}$ and $\mathcal{V}$ perform ECDHE the same as before. Then they derive handshake and application keys by executing 2PC-HKDF, and reveal the handshake keys to $\mathcal{P}$, allowing $\mathcal{P}$ to decrypt handshake messages locally (i.e., without 2PC). The 2PC circuit involves roughly 30 invocations of SHA-256, totaling to approximately 70k AND gates, comparable to that for TLS 1.2. Finally, since CBC-HMAC is not supported by TLS 1.3, DECO can only be used in GCM mode.

C.2 Query construction is optional

For applications that bind responses to queries, e.g., when a stock ticker is included with the quote, 2PC query construction protocols can be avoided altogether. Since TLS uses separate keys for each direction of communication, client-to-server keys can be revealed to $\mathcal{P}$ after the handshake so that $\mathcal{P}$ can query the server without interacting with $\mathcal{V}$.

C.3 Supporting multi-round sessions

DECO can be extended to support multi-round sessions where $\mathcal{P}$ sends further queries depending on previous responses. After each round, $\mathcal{P}$ executes similar 2PC protocols as above to verify MAC tags of incoming responses, since MAC verification and creation is symmetric. However an additional commitment is required to prevent prevent $\mathcal{P}$ from abusing MAC verification to forge tags.

In TLS, different MAC keys are used for server-to-client and client-to-server communication. To support multi-round sessions, $\mathcal{P}$ and $\mathcal{V}$ run 2PC to verify tags for former, and create tags on fresh messages for latter. We've specified the protocols to create (and verify) MAC tags. Now we discuss additional security considerations for multi-round sessions.

When checking tags for server-to-client messages, we must ensure that $\mathcal{P}$ cannot forge tags on messages that are not originally from the server. Suppose $\mathcal{P}$ wishes to verify a tag T on message M. The idea is to have $\mathcal{P}$ first commit to T, then $\mathcal{P}$ and $\mathcal{V}$ run a 2PC protocol to compute a tag T' on message M. $\mathcal{P}$ is asked to open the commitment to $\mathcal{V}$ and if $T \neq T'$, $\mathcal{V}$ aborts the protocol. Since $\mathcal{P}$ doesn't know the MAC key, $\mathcal{P}$ cannot compute and commit to a tag on a message that is not from the server.

When creating tags for client-to-server messages, $\mathcal{V}$ makes sure MAC tags are created on messages with increasing sequence numbers, as required by TLS. This also prevents a malicious $\mathcal{P}$ from creating two messages with the same sequence number, because there is no way for $\mathcal{V}$ to distinguish which one was sent to the server.

C.4 An alternative DECO protocol: Proxy mode

As shown in table 1, the HMAC mode of DECO is highly efficient and the runtime of creating and verifying HMAC tags in 2PC is independent of record size (cf. fig. 8). The GCM mode is efficient for small requests with preprocessing, but can be expensive for large records. We now present a highly efficient alternative that avoids post-handshake 2PC protocols altogether.

The idea is to have the verifier $\mathcal{V}$ act as a proxy between the prover $\mathcal{P}$ and the TLS server $\mathcal{S}$, i.e., $\mathcal{P}$ sends/receives messages to/from $\mathcal{S}$ through $\mathcal{V}$. The modified flow of the DECO protocol is as follows: after the three-party handshake, $\mathcal{P}$ commits to her key share $k_{\mathcal{P}}$ then $\mathcal{V}$ reveals $k_{\mathcal{V}}$ to $\mathcal{P}$. Therefore $\mathcal{P}$ now has the entire session key $k = k_{\mathcal{P}} + k_{\mathcal{V}}$. As $\mathcal{P}$ uses k to continue the session with the server, $\mathcal{V}$ records the proxy traffic. After the session concludes, $\mathcal{P}$ proves statements about the recorded session the same as before.

It's worth emphasizing that the three-party handshake is required for unforgeability. Unlike CBC-HMAC, GCM is not committing [42]: for a given ciphertext and tag (C,T) encrypted with key k, one can find $k' \neq k$ that decrypts C to a different plaintext while computing the same tag, as GCM MAC is not collision-resistant. To prevent such attacks, the above protocol requires $\mathcal{P}$ to commit to her key share before learning the session key.

Security properties and network assumptions. The verifier-integrity and privacy properties are clear, as a malicious $\mathcal{V}$ cannot break the integrity and privacy of TLS (by assumption).

For prover integrity, though, we need to assume that the proxy can reliably connect to $\mathcal{S}$ throughout the session. First, we assume the proxy can ascertain that it indeed is connected with $\mathcal{S}$. Moreover, we assume messages sent between the proxy and $\mathcal{S}$ cannot be

tampered with by $\mathcal{P}$, who knows the session keys and thus could modify the session content.

Note that during the three-party handshake, $\mathcal{V}$ can ascertain the server's identity by checking the server's signature over a fresh nonce (in standard TLS). After the handshake, however, $\mathcal{V}$ has to rely on network-layer indicators, such as IP addresses. In practice, $\mathcal{V}$ must therefore have correct, up-to-date DNS records, and that the network between $\mathcal{V}$ and the server (e.g., their ISP and the backbone network) must be properly secured against traffic injection, e.g., throught BGP attacks [71]. (Eavesdropping isn't problematic.)

These assumptions have been embraced by other systems in a similar proxy setting (e.g., [73]), as BGP attacks are challenging to mount in practice. We can further enhance our protocol against traffic interception by distributing verifiers nodes geographically. Moreover, various detection techniques have been proposed [2, 3, 27, 35, 51, 79, 80] that can be deployed by verifiers. Often BGP attacks are documented after the fact (e.g., see [4]), therefore, when applicable, applications of DECO can be enhanced to support revocation of affected sessions (for example, when DECO is used to issue credentials in an identity system such as [53].) We leave further exploration as future work.

This alternative protocol represents a different performance-security tradeoff. It's highly efficient because no intensive cryptography occurs after the handshake, but it requires additional assumptions about the network and therefore only withstands a weaker network adversary.

D SECURITY PROOFS

Recall Theorem 4.1. We now prove that the protocol in fig. 4 securely realizes $\mathcal{F}_{\text{Oracle}}$. Specifically, we show that for any real-world adversary $\mathcal{A}$, we can construct an ideal world simulator Sim, such that for all environments $\mathcal{Z}$, the ideal execution with Sim is indistinguishable from the real execution with $\mathcal{A}$. We refer readers to [30, 58] for simulation-based proof techniques.

Proof. Recall that we assume $\mathcal{S}$ is honest throughout the protocol. Hence, we only consider cases where $\mathcal{A}$ maliciously corrupts either $\mathcal{P}$ or $\mathcal{V}$. This means that we only need to construct ideal-world simulators for the views of $\mathcal{P}$ and $\mathcal{V}$.

Malicious $\mathcal{P}$. We wish to show the prover-integrity guarantee. Basically, if $\mathcal{V}$ receives $(b,\mathcal{S})$, then $\mathcal{P}$ must have input some θ_s such that $\mathcal{S}(\text{Query}(\theta_s)) = R$ and $b = \text{Stmt}(R)$.

Given a real-world PPT adversary $\mathcal{A}$, Sim proceeds as follows:

(1) Sim runs $\mathcal{A}$, $\mathcal{F}_{\text{ZK}}$ and $\mathcal{F}_{\text{2PC}}$ internally. Sim forwards any input z from $\mathcal{Z}$ to $\mathcal{A}$ and records the traffic going to and from $\mathcal{A}$.
(2) Upon request from $\mathcal{A}$, Sim runs 3P-HS as $\mathcal{V}$ (using $\mathcal{F}_{\text{2PC}}$ as a sub-routine). During 3P-HS, when $\mathcal{A}$ outputs a message m intended for $\mathcal{S}$, Sim forwards it to $\mathcal{F}_{\text{Oracle}}$ as $(\text{sid},\mathcal{S},m)$ and forwards (sid,m) to $\mathcal{A}$ if it receives any messages from $\mathcal{F}_{\text{Oracle}}$. By the end, Sim learns $Y_P, s_V, \text{k}_{\mathcal{V}}^{\text{MAC}}$.
(3) Upon request from $\mathcal{A}$, Sim runs 2PC-HMAC as $\mathcal{V}$, using $\text{k}_{\mathcal{V}}^{\text{MAC}}$ as input. Again, Sim uses $\mathcal{F}_{\text{2PC}}$ as a sub-routine to run 2PC-HMAC and forwards messages to $\mathcal{S}$ as above and forwards the response from $\mathcal{S}$ to $\mathcal{A}$. Sim records the messages between $\mathcal{A}$ and $\mathcal{S}$ during this stage in $(\hat{Q},\hat{R})$. Note that these are ciphertext records.
(4) When $\mathcal{A}$ sends $(\text{sid},\hat{Q},\hat{R},\text{k}_{\mathcal{P}}^{\text{MAC}})$, reply with $(\text{sid},\text{k}_{\mathcal{V}}^{\text{MAC}})$.
(5) Upon receiving $(\text{sid},\text{"prove"},x,w)$ (with $x = (\text{k}^{\text{Enc}},\theta_s,Q,R)$ and $w = (\hat{Q},\hat{R},\text{k}^{\text{MAC}},b)$) from $\mathcal{A}$, Sim checks that
$$\hat{Q} = \text{CBC\_HMAC}(\text{k}^{\text{Enc}},\text{k}^{\text{MAC}},Q)$$
$$\hat{R} = \text{CBC\_HMAC}(\text{k}^{\text{Enc}},\text{k}^{\text{MAC}},R)$$
$$Q = \text{Query}(\theta_s).$$
(6) If all of the above checks passed, Sim sends θ_s to $\mathcal{F}_{\text{Oracle}}$ and instructs $\mathcal{F}_{\text{Oracle}}$ to send the output to $\mathcal{V}$. Sim outputs whatever $\mathcal{A}$ outputs.

Now we argue that the ideal execution with Sim is indistinguishable from the real execution with $\mathcal{A}$.

Hybrid $\mathbf{H}_1$ is the real-world execution of $\text{Prot}_{\text{DECO}}$.

Hybrid $\mathbf{H}_2$ is the same as H_1, except that Sim simulates $\mathcal{A}$, $\mathcal{F}_{\text{ZK}}$ and $\mathcal{F}_{\text{2PC}}$ internally. Sim records and forwards its private θ_s input to $\mathcal{A}$. For each step of $\text{Prot}_{\text{DECO}}$, Sim forwards all messages between $\mathcal{A}$ and $\mathcal{V}$ and $\mathcal{A}$ and $\mathcal{S}$, as in the real execution. Since the simulation of ideal functionality is perfect, H_1 and H_2 are indistinguishable.

Hybrid $\mathbf{H}_3$ is the same as H_2, except that $\mathcal{V}$ sends input to $\mathcal{F}_{\text{Oracle}}$, which sends it to Sim and Sim simulates $\mathcal{V}$ internally. Specifically, Sim samples $\hat{s}_V$ and uses $\hat{s}_V \cdot Y$ to derive a share of the MAC key $\hat{K}$, which it uses in the sequential 2PC-HMAC invocations. Upon receiving $(\text{sid},\hat{Q},\hat{R},\text{k}_{\mathcal{P}}^{\text{MAC}})$, Sim sends $(\text{sid},\text{k}_{\mathcal{V}}^{\text{MAC}})$ to $\mathcal{A}$. If Sim receives $(\text{sid},\text{"prove"},x,w)$, it internally forwards it to $\mathcal{F}_{\text{ZK}}$, verifies its output as $\mathcal{V}$ and also, sends θ_s to $\mathcal{F}_{\text{Oracle}}$. The indistinguishability between H_2 and H_3 is immediate because $\hat{s}_V$ is uniformly random.

Hybrid $\mathbf{H}_4$ is the same as H_3, except Sim adds the checks in Step 5. The indistinguishability between H_3 and H_4 can be shown by checking that if any of the checks fails, $\mathcal{V}$ would abort the real-world execution as well. There are two reasons that Sim may abort: 1) Q,R from $\mathcal{A}$ is not originally from $\mathcal{S}$, or 2) $\text{k}^{\text{Enc}},\text{k}^{\text{MAC}}$ from $\mathcal{A}$ is not the same key as derived during the handshake. We now show that both conditions would trigger $\mathcal{V}$ to abort in H_3 as well except with negligible probability.

- Assuming DL is hard in the group used in the handshake, $\mathcal{A}$ cannot learn $\hat{s}_V$. Furthermore, due to the security of 2PC, $\mathcal{A}$ cannot learn the session MAC key k^{MAC}. If $\mathcal{A}$ maliciously selects $\hat{Y}_P$ correlated with $\hat{Y}_V$, it would have to find the discrete log of $\hat{Y}_P - Y_V$, denoted $\hat{s}_P$. Without such a $\hat{s}_P$, except with negligible probability, the output shares $\hat{K}_{\mathcal{V}}^{\text{MAC}}$ and $\hat{K}_{\mathcal{P}}^{\text{MAC}}$ of 3P-HS would fail to verify a MAC from an honest server whose MAC key is derived using $\hat{Y}_P$ in 2PC-HMAC, later in the protocol.
- The unforgeability guarantee of HMAC ensures that without knowledge of k^{MAC}, $\mathcal{A}$ cannot forge tags that verifies against k^{MAC} (checked by $\mathcal{V}$ in the last step of $\text{Prot}_{\text{DECO}}$).
- If $\mathcal{A}$ sends a different $(\text{k}^{\text{Enc}},\text{k}^{\text{MAC}})$ pair than that derived during the handshake to Sim and the decryption and MAC check succeeds, then $\mathcal{A}$ would have broken the receiver-binding property of CBC-HMAC [42].

It remains to show that H_4 is exactly the same the ideal execution. Due to Step 5 and 6, $\mathcal{F}_{\text{Oracle}}$ delivers $(\text{sid},\text{Stmt}(R),\mathcal{S})$ to $\mathcal{V}$ only if $\exists\theta_s$ from $\mathcal{A}$ such that R is the response from $\mathcal{S}$ to $\text{Query}(\theta_s)$.

Malicious $\mathcal{V}$. As the verifier is corrupt, we are interested in showing the verifier-integrity and privacy guarantees. Sim proceeds as follows:

(1) Sim runs $\mathcal{A}$, $\mathcal{F}_{\text{ZK}}$ and $\mathcal{F}_{\text{2PC}}$ internally to simulate the real-world interaction with the prover $\mathcal{P}$. Given input z from the environment $\mathcal{Z}$, Sim forwards it to $\mathcal{A}$.
(2) Upon receipt of Query and Stmt from $\mathcal{A}$, forward them to $\mathcal{F}_{\text{Oracle}}$ and instruct it to send them to $\mathcal{P}$.
(3) After $\mathcal{P}$ sends θ_s to $\mathcal{F}_{\text{Oracle}}$, $\mathcal{F}_{\text{Oracle}}$ sends the output (sid,Q,R) to $\mathcal{P}$. Sim gets (sid, Stmt(R), $\mathcal{S}$) from $\mathcal{F}_{\text{Oracle}}$ and learns the record sizes $|Q|$, $|R|$.
(4) Send (sid,$\mathcal{S}$,handshake) to $\mathcal{F}_{\text{Oracle}}$, where handshake contains client handshake messages and receive certificate and signatures of $\mathcal{S}$ from $\mathcal{F}_{\text{Oracle}}$. Note that at the end of the server handshake, $\mathcal{P}$ receives and sends finished messages, which we denote "serverFinished" and "proverFinished". The finished messages include HMAC tags, which we denote $\tau_{\mathcal{S}}$ and $\tau_{\mathcal{P}}$ (tags on $\mathcal{S}$ and $\mathcal{P}$'s messages respectively).
(5) Upon request from $\mathcal{A}$, Sim runs 3P-HS as $\mathcal{P}$, using the server handshake messages received in the previous step, learning s_P, Y_V, k^{Enc}, $\mathsf{k}_{\mathcal{P}}^{\text{MAC}}$.
(6) Sim starts 2PC-HMAC as $\mathcal{P}$ to compute a tag τ_q on a random $Q' \leftarrow_s \{0,1\}^{|Q|}$.
(7) Sim uses a random key $\hat{k}$ to compute a tag τ_r on a random $R' \leftarrow_s \{0,1\}^{|R|}$.
(8) Let $\hat{Q} = \text{CBC}(\mathsf{k}^{\text{Enc}}, Q' \| \tau_q)$ and $\hat{R} = \text{CBC}(\mathsf{k}^{\text{Enc}}, R' \| \tau_r)$. At the commit phase, Sim sends encrypted data (sid,$\hat{Q}$,$\hat{R}$,$\mathsf{k}_{\mathcal{P}}^{\text{MAC}}$) to $\mathcal{A}$ and receives $\mathsf{k}_{\mathcal{V}}^{\text{MAC}}$ from $\mathcal{A}$.
(9) Sim asserts that $\tau_{\mathcal{S}} = \text{HMAC}(\mathsf{k}^{\text{MAC}}, \text{"serverFinished"})$ and that $\tau_{\mathcal{P}} = \text{HMAC}(\mathsf{k}^{\text{MAC}}, \text{"proverFinished"})$.
(10) Sim asserts that $\tau_q = \text{HMAC}(\mathsf{k}^{\text{MAC}}, Q')$.
(11) To simulate the appropriate delay, Sim also runs a dummy computation $\text{HMAC}(\mathsf{k}^{\text{MAC}}, R')$ in paralell with Step 9.
(12) Sim sends (sid,"proof",1,($\hat{Q}$,$\hat{R}$,k^{MAC},Stmt(R))) to $\mathcal{A}$ and outputs whatever $\mathcal{A}$ outputs.

We argue that the ideal execution with Sim is indistinguishable from the real execution with $\mathcal{A}$ in a series hybrid worlds.

Hybrid $\mathbf{H}_1$ is the real-world execution of $\text{Prot}_{\text{DECO}}$.

Hybrid $\mathbf{H}_2$ is the same as H_1, except that Sim simulates $\mathcal{F}_{\text{ZK}}$ and $\mathcal{F}_{\text{2PC}}$ internally. Sim also invokes $\mathcal{F}_{\text{Oracle}}$ and gets (sid,Stmt(R),$\mathcal{S}$), learns record sizes $|Q|$, $|R|$. Since the simulation of ideal functionality is perfect, H_1 and H_2 is indistinguishable.

Hybrid $\mathbf{H}_3$ is the same as H_2, except that Sim simulates $\mathcal{P}$. Specifically, Sim samples s_P and uses $s_P \cdot Y$ to derive a share of the MAC key $\mathsf{k}_{\mathcal{P}}^{\text{MAC}}$. Then, Sim uses $\mathsf{k}_{\mathcal{P}}^{\text{MAC}}$ and a random $Q' = \{0,1\}^{|Q|}$ as inputs to 2PC-HMAC and receives the tag τ_q. Then, Sim uses a random key $\hat{k}$, and a random $R' = \{0,1\}^{|R|}$ to compute a dummy tag τ_r. Afterwards, Sim commits, i.e., sends encryption of Q' and R' to $\mathcal{A}$. Sim also adds the checks in Step 9 and 10. To simulate the appropriate delay for checking a tag on R', a plaintext of length $|R|$, Sim runs a dummy tag computation. Finally, Sim skips invoking $\mathcal{F}_{\text{ZK}}$ and directly provides $\mathcal{A}$ with the output obtained earlier from $\mathcal{F}_{\text{Oracle}}$, i.e., Stmt($R$), alongwith k^{MAC}, i.e. the tuple (sid,"proof",1,($\hat{Q}$,$\hat{R}$,k^{MAC},Stmt(R))). $\mathcal{A}$ cannot distinguish between the real and ideal executions because:

(1) Since input sizes are equal, the number of invocations of 2PC-HMAC is also equal.
(2) In each invocation of 2PC-HMAC and HMAC, $\mathcal{A}$ learns one SHA-2 hash of the input message which is like a random oracle.
(3) If the value of $\mathsf{k}_{\mathcal{V}}^{\text{MAC}}$ provided by $\mathcal{V}$ is correct, in both the real and ideal world, all tags should verify and the protocol should proceed to the next step and the time to run the checks should be indistinguishable from the real world.
(4) $\mathcal{A}$ can provide a malicious $\mathsf{k}_{\mathcal{V}}^{\text{MAC}}$ in two ways:
- Malicious $\mathsf{k}_{\mathcal{V}}^{\text{MAC}}$ is provided by $\mathcal{V}$ in Step 8: $\tau_{\mathcal{S}}$ and $\tau_{\mathcal{P}}$ will not verify in Step 9. Sim will then abort with the same delay as in the real world.
- $\mathcal{A}$ inputs a malicious $\mathsf{k}_{\mathcal{V}}^{\text{MAC}}$ to the 2PC-HMAC: τ_q will fail to verify in 10 by the same argument as in the malicious $\mathcal{P}$ case.

(5) Since $|Q'| = |Q|$ and $|R'| = |R|$, their encryptions are also of equal size and indistinguishable.
(6) In the end, $\mathcal{A}$ receives the same output as the real execution.

□

E APPLICATION DETAILS

We provide the remaining application details omitted from Sec. 6 here.

Binary Option. The user ($\mathcal{P}$) also needs to reveal enough portion of the HTTP GET request to oracle ($\mathcal{V}$) in order to convince access to the correct API endpoint. The GET request contains several parameters—some to be revealed like the API endpoint, and others with sensitive details like stock name and private API key. $\mathcal{P}$ redacts sensitive params using techniques from Sec. 5.1 and reveals the rest to $\mathcal{V}$. The API key provides enough entropy preventing $\mathcal{V}$ from learning the sensitive params. Without additional care though, a cheating $\mathcal{P}$ can alter the semantics of the GET request and conceal the cheating by redacting extra parameters. To ensure this does not happen, $\mathcal{P}$ needs to prove that the delimiter "&" and separator "=" do not appear in the redacted text. The security is argued below.

HTTP GET requests (and HTML) have a special restriction: the demarcation between a `key` and a `value` (i.e., `middle`) and the start of a key-value pair (i.e., `start`) are never substrings of a `key` or a `value`. This means that to redact more than a single contiguous `key` or `value`, $\mathcal{P}$ must redact characters in {`middle`, `start`}. So we have $\text{cons}_{\mathcal{G},\mathcal{G}'}(R,R')$ check that: (1) $|R| = |R'|$; and (2) $\forall i \in |R'|$, either $R'[i] = D \wedge R[i] \notin \{\texttt{middle}, \texttt{start}\}$ or $R[i] = R'[i]$ (D is a dummy character used to do in-place redaction). Checking $\text{CTX}_{\mathcal{G}'}$ is then unnecessary.

Age Proof. Figure 10 shows the demographic details of a student stored on Univ. website such as the name, birth date, student ID among others. The prover parses 6-7 AES blocks that contain the birth date and proves her age is above 18 in ZK to the verifier. Like other examples, due to the unique HTML tags surrounding the birth date, this is also a key-value grammar with unique keys (see Sec. 5.2). Similar to application 1, this example requires additional string processing to parse the date and compute age.

Price discrimination. Figure 11 shows parts of an order invoice page on a shopping website (Amazon) with personal details such as the name and address of the buyer. The buyer wants to convince a third-party (verifier) about the charged price of a particular product on a particular date. In this example, we use AES-GCM ciphersuite and Reveal mode. Only necessary details in the invoice like the item

```
<title>Demographic Data</title>
<span id='EMPLID'> 111111 </span>
<span id='NAME'> Alice </span>
<span id='BIRTHDATE'> 01/01/1990 </span> ...
```

Figure 10: The demographic details of a student displayed on a Univ. website. Highlighted text contains student age. Reveal **mode is used together with two-stage parsing.**

```
<table>
 <tr>Order Placed: November 23, 2018</tr>
 <tr>Order Total: $34.28</tr>
 <tr>Items Ordered: Food Processor</tr>
</table>
...
<b> Shipping Address: </b>
<ul class="displayAddressUL">
<li class="FullName">Alice</li>
<li class="Address">Wonderland</li>
<li class="City">New York</li>
</ul>
```

Figure 11: The order invoice page on Amazon in HTML. Reveal **mode is used to reveal the necessary text, while sensitive text below is kept hidden.**

name, item price and order date are revealed, while hiding the rest. Number of AES blocks revealed from the response is 20 (thanks to a long product name). In addition, 4 AES blocks from the request are revealed to prove that the correct endpoint is accessed. Context integrity is guaranteed by revealing unique strings around, e.g., the string "<tr>Order Total:" near the item price appears only once in the entire response.

F KEY-VALUE GRAMMARS AND TWO-STAGE PARSING

F.1 Preliminaries and notation

We denote context-free grammars as $\mathcal{G} = (V, \Sigma, P, S)$ where V is a set of non-terminal symbols, Σ a set of terminal symbols, $P : V \rightarrow (V \cup \Sigma)^*$ a set of productions or rules and $S \in V$ the start-symbol. We define production rules for CFGs in standard notation using '-' to denote a set minus and '..' to denote a range. For a string w, a parser determines if $w \in \mathcal{G}$ by constructing a parse tree for w. The parse tree represents a sequence of production rules which can then be used to extract semantics.

F.2 Key-value grammars

These are grammars with the notion of key-value pairs. These grammars are particularly interesting for DECO since most API calls and responses are, in fact, key-value grammars.

Definition F.1. $\mathcal{G}$ is said to be a key-value grammar if there exists a grammar $\mathcal{H}$, such that given any $s \in \mathcal{G}$, $s \in \mathcal{H}$, and $\mathcal{H}$ can be defined by the following rules:

```
S → object
object → noPairsString open pair pairs close
pair → start key middle value end
pairs → pair pairs | ""
key → chars
value → chars | object
chars → char chars | ""
char → Unicode - escaped | escape escaped | addedChars
special → startSpecial | middleSpecial | endSpecial
start → unescaped_s startSpecial
middle → unescaped_m middleSpecial
end → unescaped_e endSpecial
escaped → special | escape | ...
```

In Def. F.1, S is the start non-terminal (represents a sentence in $\mathcal{H}$), the non-terminals open and close demarcate the opening and closing of the set of key-value pairs and start, middle, end are special strings demarcating the start of a key-value pair, separation between a key and a value and the end of the pair respectively.

In order to remove ambiguity in parsing special characters, i.e. characters which have special meaning in parsing a grammar, a special non-terminal, escape is used. For example, in JSON, keys are parsed when preceded by 'whitespace double quotes' (") and succeeded by double quotes. If a key or value expression itself must contain double quotes, they must be preceded by a backslash (\), i.e. escaped. In the above rules, the non-terminal unescaped before special characters means that they can be parsed as special characters. So, moving forward, we can assume that the production of a key-value pair is unambigious. So, if a substring R' of a string R in the key-value grammar $\mathcal{G}$ parses as a pair, R' must correspond to a pair in the parse tree of R.

Note that in Def. F.1, middle cannot derive an empty string, i.e. a non-empty string must mark middle to allow parsing keys from values. However, one of start and end can have an empty derivation, since they only demarcate the separation between value in one pair from key in the next. Finally, we note that in our discussion of two-stage parsing for key-value grammars, we only we consider permissible paths with the requirement that the selectively opened string, R_{open} corresponds to a pair.

F.3 Two-stage parsing for a locally unique key

Many key-value grammars enforce key uniqueness within a scope. For example, in JSON, it can be assumed that keys are unique within a JSON object, even though there might be duplicated keys across objects. The two-stage parsing for such grammars can be reduced to parsing a substring. Specifically, Trans extracts from R a continuous substring R', such that the scope of a pair can be correctly determined, even within R'. For instance, in JSON, if $\text{cons}_{\mathcal{G},\mathcal{G}'}(R,R')$ returns true iff R' is a prefix of R, then only parsing R' as a JSON, up to generating the sub-tree yielding R_{open} is sufficient for determining whether a string R_{open} corresponds to the correct context in R.

F.4 Grammars with unique keys

Given a key-value grammar $\mathcal{G}$ we define a function which checks for uniqueness of keys, denoted $u_{\mathcal{G}}$. Given a string $s \in \mathcal{G}$ and another string k, $u_{\mathcal{G}}(s,k) = \texttt{true}$ iff there exists at most one substring of s that can be parsed as start k middle. Since $s \in \mathcal{G}$, this means, in any parse tree of s, there exists at most one branch with node key and derivation k. Let $\text{Parser}_{\mathcal{G}}$ be a function that returns true if its input is in the grammar $\mathcal{G}$. We say a grammar $\mathcal{G}$ is a *key-value grammar with unique keys* if for all $s \in \mathcal{G}$ and all possible keys k, $u_{\mathcal{G}}(s,k) = \texttt{true}$, i.e. for all strings R, C:

$$\frac{\langle \text{Parser}_{\mathcal{G}}, R\rangle \Rightarrow \text{true}}{\langle u_{\mathcal{G}}, (R,C)\rangle \Rightarrow \text{true}}.$$

F.5 Concrete two-stage parsing for unique-key grammars

Let $\mathcal{U}$ be a unique-key grammar as given above. We assume that $\mathcal{U}$ is LL(1). This is the case for the grammars of interest in Section 6. See [43] for a general LL(1) parsing algorithm.

We instantiate a context function, $\mathsf{CTX}_{\mathcal{U}}$ for a set T, such that T contains the permissible paths to a `pair` for strings in $\mathcal{U}$. We additionally allow $\mathsf{CTX}_{\mathcal{U}}$ to take as input an auxiliary restriction, a key k (the specified key in $\mathcal{P}$'s output R_{open}). The tuple (T,k) is denoted S and $\mathsf{CTX}_{\mathcal{U}}(S,\cdot,\cdot)$ as $\mathsf{CTX}_{\mathcal{U},S}$.

Let $\mathcal{P}$ be a grammar given by the rule $\mathtt{S}_{\mathcal{P}} \rightarrow \mathtt{pair}$, where `pair` is the non-terminal in the production rules for $\mathcal{U}$ and $\mathtt{S}_{\mathcal{P}}$ is the start symbol in $\mathcal{P}$. We define $\mathsf{Parser}_{\mathcal{P},\mathsf{k}}$ as a function that decides whether a string s is in $\mathcal{P}$ and if so, whether the key in s equals k. On input R,R_{open}, $\mathsf{CTX}_{\mathcal{U},S}$ checks that: (a) R_{open} is a valid key-value pair with key k by running $\mathsf{Parser}_{\mathcal{P},\mathsf{k}}$ (b) R_{open} parses as a key-value pair in R by running an LL(1) parsing algorithm to parse R.

To avoid expensive computation of $\mathsf{CTX}_{\mathcal{U},S}$ on a long string R, we introduce the transformation Trans, to extract the substring R' of R, such that $R' = R_{\text{open}}$ as per the requirements.

For string s,t, we also define functions $substring(s,t)$, that returns true if t is a substring of s and $equal(s,t)$ which returns true if $s=t$. We define $\mathsf{cons}_{\mathcal{U},\mathcal{P}}$ with the rule:

$$\frac{\langle substring(R,R')\rangle \Rightarrow \mathsf{true}\langle \mathsf{Parser}_{\mathcal{P},\mathsf{k}},R'\rangle \Rightarrow \mathsf{true}}{\langle cons_{\mathcal{U},\mathcal{P}},(R,R')\rangle \Rightarrow \mathsf{true}}.$$

and $S' = \{\mathtt{S}_{\mathcal{P}}\}$. Meaning, $\mathsf{CTX}_{\mathcal{P}}(S, R', R_{\text{open}}) = \mathsf{true}$ whenever $equal(R',R_{\text{open}})$ and the rule

$$\frac{\langle equal,(R',R_{\text{open}})\rangle \Rightarrow \mathsf{b}}{\langle \mathsf{CTX}_{\mathcal{P}},(S',R',R_{\text{open}})\rangle \Rightarrow \mathsf{b}}$$

holds for all strings R',R_{open}.

Claim 1. $(\mathsf{cons}_{\mathcal{U},\mathcal{P}},S')$ *are correct with respect to* S.

Proof. We defer a formal proof and pseudocode for $\mathsf{CTX}_{\mathcal{U},S}$ to a full version, but the intuition is that if R' is substring of R, a key-value pair R_{open} is parsed by $\mathsf{Parser}_{\mathcal{P}}$, then the same pair must have been a substring of $\mathcal{U}$. Due to global uniqueness of keys in $\mathcal{U}$, there exists only one such pair R_{open} and $\mathsf{CTX}_{\mathcal{U}}(S,R,R_{\text{open}})$ must be true. □