DiStefano: Decentralized Infrastructure for Sharing Trusted Encrypted Facts and Nothing More

Private and Efficient Commitments for TLS-encrypted Data

*Sofía Celi**, *Alex Davidson*[†], *Hamed Haddadi**[¶], *Gonçalo Pestana*[‡] *and Joe Rowell*[§]
* *Brave Software, cherenkov@riseup.net, hamed@brave.com*
[†] *NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, a.davidson@fct.unl.pt*
[‡] *Hashmatter, gpestana@hashmatter.com*
[§]*Information Security Group, Royal Holloway, University of London, joe.rowell@rhul.ac.uk*
[¶]*Imperial College London, h.haddadi@imperial.ac.uk*

***Abstract*—We design DiStefano: an efficient framework for generating private commitments over TLS-encrypted web traffic for a designated, untrusted third-party. DiStefano provides many improvements over previous TLS commitment systems, including: a modular security model that is applicable to TLS 1.3 traffic, and support for generating verifiable claims using applicable zero-knowledge systems; inherent 1-out-of-n privacy for the TLS server that the client communicates with; and various cryptographic optimisations to ensure fast online performance of the TLS session. We build an open-source implementation of DiStefano integrated into the BoringSSL cryptographic library, that is used within Chromium-based Internet browsers. We show that DiStefano is practical for committing to facts in arbitrary TLS traffic, with online times that are comparable with existing TLS 1.2 solutions. We also make improvements to certain cryptographic primitives used inside DiStefano, leading to 3× and 2× improvements in online computation time and bandwidth in specific situations.**

1. Introduction

The Transport-Layer Security (TLS) protocol [1] provides encrypted and authenticated channels between clients and servers on the Internet. It is common that such channels transmit trusted information about users behind clients, including: proofs of age, social security statuses, and accepted purchase information. Unfortunately, this traffic cannot be trivially used as a commitment to such information to provide to third parties, since such information is locked in a symmetrically encrypted and authenticated channel. Applications of third-party verification of such claims include Intenet-based verification of age [2], ID numbers, and other social security-type statuses [3]. More generically, such tools could facilitate the creation of privacy-preserving credentials for proving arbitrary facts about a user based on their online behaviour — for example, as intended by the W3C Decentralized Identity specification [4].

A number of *Designated-Commitment*[1] TLS (DCTLS) protocols have been designed in order to allow exporting verifiable claims to a designated third-party (verifier) over the trusted information transmitted in such channels. The most prominent example is the DECO protocol [2], alongside browser-based tools such as TLSNotary/PageSigner.[2] Similar ideas have also been used for devising multi-party TLS clients/servers, such as Oblivious TLS [6], and N-for-1-Auth [7]. DCTLS protocols use a modified TLS handshake (on the client-side) that involves secret-sharing secret session data amongst the client and an entity called the verifier, and computing handshake and record-layer protocol functionality in two-party computation (2PC). This handshake procedure allows the client to eventually commit to certain TLS session data, which they can later prove facts about, in zero-knowledge, using their cryptographic shares.

Unfortunately, existing TLS commitment mechanisms do not provide sufficient privacy — they, for example, expose the browsing history of the client to the verifier. Or, they do not satisfy security in the most obvious settings — PageSigner only targets security against an honest-but-curious client, and DECO uses a non-modular security framework (mandating certain post-commitment offline proving steps) that specifically targets the TLS 1.2 protocol.[3] According to Cloudflare Radar [8]: TLS 1.3 accounts for 63% of secure network traffic as opposed to 8.7% for TLS 1.2. In addition, [9] argues that more than 15% of websites supported TLS 1.3 264 days after the IETF officially standardised the protocol in April 2019, and support surpassed that of TLS 1.2 around December 2020. Hence, supporting TLS 1.3 connections explicitly in formal protocol specifications is a necessity for any tool hoping to see

[1] Also known as *three-party handshake* protocols.

[2] Note that, in this work, we refer exclusively to the older implementation known as PageSigner [5]. As far as we are aware, TLSNotary does not yet have a fixed cryptographic design.

[3] That is, the presented security analysis for TLS 1.3 is rather informal in both cases.

widespread adoption. Finally, there is no established, publicly available implementation of *any* DCTLS scheme that is compatible with TLS 1.3, and that retains security against malicious parties.

Our work. We devise DiStefano, a DCTLS protocol that allows for the generation of private commitments over data communicated explicitly during TLS 1.3 sessions. For analysing the security of DiStefano, we devise a novel security framework that allows proving the security of the individual stages of the protocol, independently of succeeding functionality. One immediate benefit of this change is that this allows DiStefano to improve client privacy guarantees by removing explicit server authentication to the verifier during the TLS handshake. Instead, we replace it with a ring signature that conveys *1-out-of-n* authentication, where the verifier holds a list of n accepted server public keys. The security of the DCTLS handshake satisfies well-known security properties for standard TLS 1.3 [10], and based on previously-established assumptions [6]. Notably, our security model allows providing verifiable claims using any secure framework for doing so — e.g. zero-knowledge proofs, anonymous credentials, or using 2PC protocols.

DiStefano uses a similar suite of cryptographic tools as previous work [2], [6], [7], but we introduce a number of cryptographic improvements and optimisations. This includes establishing a novel mechanism for running AES-GCM encryption and decryption functionalities in 2PC, that improves pre-processing computation and bandwidth usage by factors of 3 and 2, respectively. In addition, we use primitives for Multiplicative-to-Additive (MtA) share transformations and oblivious transfer that confer malicious security, where previous work used primitives with honest-but-curious guarantees for benchmarking performance.

Overall, we provide an open-source implementation of DiStefano that is compliant with modern Chromium-based Internet browsers, using the BoringSSL cryptography library [11].[4][5] Since DiStefano is compatible with any attestations over the commitments that are produced, verifiable claims can be then written using any library of the implementer's choice. We subsequently demonstrate, via experimental analysis, that DiStefano runs very efficiently for privately committing to TLS 1.3 data, by evaluating the performance of each of the individual components.[6] In particular, the online portions of the handshake and record-layer protocols can be executed in 500 and 190 milliseconds, and with 5KB and around 4KB of bandwidth, respectively, for 2KB of encrypted communications.

Formal contributions.

- A private Delegated-Commitment TLS 1.3 (DCTLS) protocol, DiStefano (Section 4), with a novel modular security framework that allows proving security against malicious adversaries (Section 6).
- Novel cryptographic optimisations that allow running secure 2PC TLS 1.3 clients with higher efficiency (Section 5).
- A permissive, open-source, Chromium-compliant implementation of the TLS 1.3 protocol, integrated into the BoringSSL library (Section 7).[7]
- Experimental analysis that shows that DiStefano is practically efficient for committing to web-based Internet traffic, without compromising the client experience (Section 8).

Layout. Our paper is divided as follows: we present background information of DCTLS and similar protocols in Section 2; we present DiStefano and its core primitives in Section 3; we formally describe the phases of DiStefano in Section 4; we describe our performance optimisations for securely running AES-GCM in 2PC in Section 5; we provide our modular DCTLS security framework, and a security analysis of DiStefano in Section 6; we provide implementation and experimentation details in Section 7 and Section 8, respectively; we discuss related work, applications, and limitations in Section 9; and we conclude in Section 10.

2. Background

2.1. General Notation

Vectors are denoted by lower-case bold letters i.e. $\mathbf{a}$. For any string s, $\mathsf{len}(s)$ denotes the length of s. The symbol $[m]$ indicates the set $1, 2, \ldots, m$. We write $a \leftarrow b$ to assign the value of b to a, and we write $a \leftarrow\!\!\$\, S$, where S is a set, to assign a uniformly sampled element from S. λ denotes the security parameter.

We denote a finite field of characteristic q as $\mathbb{F}_q$ and the m-dimensional vector space over $\mathbb{F}_q$ as $\mathbb{F}_{q^m}$. In this work, we are primarily concerned with the smallest field, $\mathbb{F}_2$. In this field, the additive operation on $a, b \in \mathbb{F}_2$ is simply an *exclusive-or* operation, $a \oplus b$, with multiplication corresponding to the AND operation. We extend this notation to refer to operations on m-dimensional vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}_{2^m}$, writing $\mathbf{a} \oplus \mathbf{b}$ and $\mathbf{a} \cdot \mathbf{b}$ to refer to addition and multiplication, respectively. Note that while addition in $\mathbb{F}_{2^m}$ is simply m XOR operations, multiplication over $\mathbb{F}_{2^m}$ requires extra logic compared to multiplications over $\mathbb{F}_2$. We write elliptic curves with a generator G over $\mathbb{F}_q$ as $EC(\mathbb{F}_q)$.

Finally, for a security game Game used some scheme cryptographic scheme Δ, we denote the advantage of an algorithm $\mathcal{A}$ by $\mathsf{Adv}^{\mathsf{game}}_{\mathcal{A},\Delta}(\lambda)$, where:

$$\mathsf{Adv}^{\mathsf{game}}_{\mathcal{A},\Delta}(\lambda) = \Pr[\mathcal{A} \text{ succeeds}] - \Pr[\mathcal{A} \text{ fails}]. \quad (1)$$

[4] https://anonymous.4open.science/r/tls-at-6823

[5] Note that Chromium-based Internet browsers make up a dominant share of all Internet browser usage.

[6] We plan to update this work with full end-to-end performance figures in the near future.

[7] Source code: https://anonymous.4open.science/r/tls-at-6823

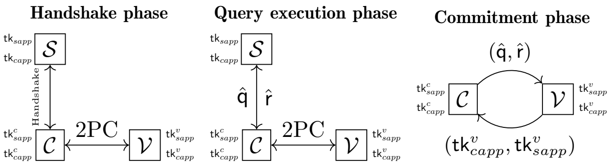

Figure 1. An overview of existing DCTLS protocols. During the **query phase**, $\mathcal{C}$ sends the encrypted query $\hat{\mathsf{q}}$, that is constructed in assistance with $\mathcal{V}$, and receives the encrypted response $\hat{\mathsf{r}}$. In the commitment phase, $\mathcal{C}$ sends the encrypted data to $\mathcal{V}$, and receives the secret TLS parameters from $\mathcal{V}$. This allows $\mathcal{C}$ to later prove facts about their commitment.

We say that Δ is secure with respect to Game, iff $\mathsf{Adv}^{\mathsf{game}}_{\mathcal{A},\Delta}(\lambda) \leq \mathsf{negl}(\lambda)$, for some negligible function $\mathsf{negl}(\lambda)$ and security parameter λ.

2.2. Background on DCTLS Protocols

Designated-Commitment (DCTLS) TLS protocols allow a client ($\mathcal{C}$) to generate commitments to TLS session data communicated with a server ($\mathcal{S}$) that can be sent to a designated third-party verifier ($\mathcal{V}$). They consist of the following phases (which are described in Appendix A): a (verifier-assisted) handshake phase, a (verifier-assisted) query execution phase, and a commitment phase. Previous work, such as DECO, and tools such as PageSigner, provide explicit functionality for generating zero-knowledge proofs that attest to certain data that is part of the committed TLS session. In this work, we prefer to build a modular framework for solving the core DCTLS functionality, and leave the implementation of the subsequent proving stage up to the implementer, see Section 4.5 for more discussion. Note that, without such commitments, proving statements that use TLS data as sources of truth must assume either a trustworthy client, or $\mathcal{C}$ must allow $\mathcal{V}$ to read their TLS traffic in the clear.

DCTLS over TLS 1.3. Previous DCTLS protocols focused on TLS 1.2, with an informal (and, sometimes, incomplete) extension to TLS 1.3. In this work, we crucially focus on TLS 1.3, and argue that previous protocols cannot be easily extended or build for both TLS 1.2 and TLS 1.3. At a protocol level, TLS 1.2 and 1.3 differ substantially. TLS 1.3, as seen in Fig. 2 and using the information of Table 1 and Table 2, has a reduced number of round-trips due to the need for efficiency, parts of the handshake are encrypted, it has resilience to cross-protocol attacks and changes the flow of messages when compared with TLS 1.2. Furthermore, TLS 1.3 introduces non-trivial 2PC implementation issues (when running AES-GCM particularly) that previous works acknowledge, but do not address.

Description of DCTLS phases. In Fig. 1 we give an overview of the stages of the DCTLS protocol, for establishing commitments to TLS-encrypted traffic between a client and server to be sent to a designated verifier. In the following, we describe how the different stages of the protocol function, specifically in relation to the various stages of the TLS 1.3 protocol [1]. TLS 1.3 emerged in response to dissatisfaction with the outdated design of the TLS 1.2 handshake, its two-round-trip overhead, and the increasing number of practical attacks on it [12], [13], [14], [15]. A description and analysis of its handshake can be found in [10, Figure 1]. The following is an informal description of TLS 1.3 (1-RTT with certificate-based authentication) and how can it be extended to support DCTLS-like protocols.

Client **Server**

static (Sig): $\mathsf{pk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}}$

`ClientHello:`
$r_c \leftarrow_\$ \{0,1\}^{256}$
$x_c \leftarrow_\$ \mathbb{Z}_q$
`+ClientKeyShare:`
$X_c \leftarrow g^{x_c}$

`ServerHello:`
$r_s \leftarrow_\$ \{0,1\}^{256}$
$y_s \leftarrow_\$ \mathbb{Z}_q$
`+ServerKeyShare:` $Y_s \leftarrow_\$ g^{y_s}$

$\mathrm{DHE} \leftarrow Y_s^{x_c}$ $\mathrm{DHE} \leftarrow X_c^{y_s}$

$\mathrm{HS} \leftarrow \mathsf{HKDF}.\mathsf{Extract}(\varnothing, \mathrm{DHE})$
$\mathrm{CHTS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{HS}, \mathrm{Label}_1 \parallel \mathrm{H}_0)$
$\mathrm{SHTS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{HS}, \mathrm{Label}_2 \parallel \mathrm{H}_0)$
$\mathrm{dHS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{HS}, \mathrm{Label}_3 \parallel \mathrm{H}_1)$
$\mathsf{tk}_{chs} \leftarrow \mathsf{DeriveTK}(\mathrm{CHTS})$
$\mathsf{tk}_{shs} \leftarrow \mathsf{DeriveTK}(\mathrm{SHTS})$

`{+EncryptedExtensions }`
`{+CertificateRequest }`*
`{+ServerCertificate:}`$\mathsf{pk}_{\mathcal{S}}$
`{+ServerCertificateVerify:}`
$\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{S}}, \mathrm{Label}_7 \parallel \mathrm{H}_3)$
$\mathsf{fk}_{\mathcal{S}} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{SHTS}, \mathrm{Label}_4 \parallel \mathrm{H}_\epsilon)$
`{+ServerFinished:}` $SF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$

$\mathrm{MS} \leftarrow \mathsf{HKDF}.\mathsf{Extract}(\mathrm{dHS}, \varnothing)$
$\mathrm{CATS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{MS}, \mathrm{Label}_5 \parallel \mathrm{H}_2)$
$\mathrm{SATS} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{MS}, \mathrm{Label}_6 \parallel \mathrm{H}_2)$

`{+ClientCertificate:}`$^*\mathsf{pk}_{\mathcal{C}}$
`{+ClientCertificateVerify:}`*
$\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{C}}, \mathrm{Label}_8 \parallel \mathrm{H}_5)$
$\mathsf{fk}_{\mathcal{C}} \leftarrow \mathsf{HKDF}.\mathsf{Expand}(\mathrm{CHTS}, \mathrm{Label}_4 \parallel \mathrm{H}_\epsilon)$
`{+ClientFinished:}` $CF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6))$

$\mathsf{tk}_{capp} \leftarrow \mathsf{DeriveTK}(\mathrm{CATS})$
$\mathsf{tk}_{sapp} \leftarrow \mathsf{DeriveTK}(\mathrm{SATS})$

abort if $\mathsf{Verify}(\mathsf{pk}_c, \mathrm{Label}_8 \parallel \|\mathrm{H}_5, \mathsf{Sig}) \neq 1$
abort if $CF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6)$

Figure 2. TLS 1.3 handshake with certificate-based authentication

Handshake phase. In this phase, the server, $\mathcal{S}$, learns the same secret session parameters (i.e. session key information) as in standard TLS 1.3, while $\mathcal{C}$ and $\mathcal{V}$ learn shares of the session parameters that only "regular" $\mathcal{C}$ would normally learn. Then, $\mathcal{C}$ and $\mathcal{V}$ engage in the core TLS 1.3 using a series of 2PC functionalities.

We focus on the default mode for establishing a secure TLS 1.3 session using (EC)DH ciphersuites, and using certificate-based authentication between $\mathcal{C}$ and $\mathcal{S}$. In this mode, the handshake starts with $\mathcal{C}$ sending a `ClientHello` (`CH`) message to $\mathcal{S}$. This mes-

Secret	Context Input	Label
CHTS	$H_0 = \mathcal{H}($ClientHello,...,ServerHello $)$	Label$_1$ = “c hs traffic”
SHTS	$H_0 = \mathcal{H}($ClientHello,...,ServerHello $)$	Label$_2$ = “s hs traffic”
dHS	$H_1 = \mathcal{H}(``'')$	Label$_3$ = “derived”
$\mathsf{fk}_{\mathcal{S}}$	$H_\epsilon = \mathcal{H}(``'')$	Label$_4$ = “finished”
$\mathsf{fk}_{\mathcal{C}}$	$H_\epsilon = \mathcal{H}(``'')$	Label$_4$ = “finished”
CATS	$H_2 = \mathcal{H}($ClientHello,...,ServerFinished $)$	Label$_5$ = “c ap traffic”
SATS	$H_2 = \mathcal{H}($ClientHello,...,ServerFinished $)$	Label$_6$ = “s ap traffic”

Table 1. TLS 1.3 handshake and traffic secrets.

Auth message	Context Input	Context String
ServerCertificateVerify	$H_3 = \mathcal{H}($CH,...,SCRT$)$	Label$_7$ = “TLS 1.3, server CertificateVerify”
ServerFinished	$H_4 = \mathcal{H}($CH,...,SCV$)$	
ClientCertificateVerify	$H_5 = \mathcal{H}($CH,...,CCRT$)$	Label$_8$ = “TLS 1.3, client CertificateVerify”
ClientFinished	$H_6 = \mathcal{H}($CH,...,CCV$)$	

Table 2. TLS 1.3 authentication messages and associated hashes.

sage advertises the supported (EC)DH groups and the ephemeral (EC)DH keyshares offered and specified in the supported_groups and key_shares extensions, respectively. The CH message also advertises the signature algorithms supported. It also contains a nonce and a list of supported symmetric-key algorithms (ciphersuites). Note that for DCTLS protocols, the ephemeral keyshares $Z \in EC(\mathbb{F}_c)$ are generated as a combination of additive shares $(z_X \leftarrow\$\ \mathbb{F}_c, Z_X = z_X \cdot G)$ for $X \in \{\mathcal{C}, \mathcal{V}\}$, where $Z = Z_{\mathcal{C}} + Z_{\mathcal{V}} \in EC(\mathbb{F}_c)$.

$\mathcal{S}$ processes the CH message and chooses the cryptographic parameters to be used in the session. If (EC)DH key exchange is in use, $\mathcal{S}$ sends a ServerHello (SH) message containing a key_share extension with the server's (EC)DH key, corresponding to one of the key_shares advertised by $\mathcal{C}$. The SH message also contains a server-generated nonce and the ciphersuite chosen. An ephemeral shared secret is then computed at both ends, which requires $\mathcal{C}$ and $\mathcal{V}$ to engage in a 2PC computation to derive this secret. After this action, all subsequent handshake messages are encrypted using keys derived from this secret. Fortunately, once this derivation is performed, $\mathcal{V}$'s keys can be revealed to $\mathcal{C}$ to perform local encryption/decryption of handshake messages, as these keys are considered independent from the eventual session secret derived at the end of the handshake [10].

$\mathcal{S}$ then sends a certificate chain (in the ServerCertificate message -SCRT-), and a message that contains a proof that the server possesses the private key corresponding to the public key — as advertised in its leaf certificate. This proof is a signature over the handshake transcript and it is sent in the ServerCertificateVerify (SCV) message. $\mathcal{S}$ also sends the ServerFinished (SF) message that provides integrity of the handshake up to this point. It contains a message authentication code (MAC) over the entire transcript, providing key confirmation and binding the server's identity to any computed keys.

Optionally, $\mathcal{S}$ can send a CertificateRequest (CR) message, prior to sending its SCRT message, requesting a certificate from $\mathcal{C}$. At this point, $\mathcal{S}$ can immediately send application data to the unauthenticated client. Upon receiving the server's messages, $\mathcal{C}$ verifies the signature of the SCV message and the MAC of the SF message. If requested, $\mathcal{C}$ must respond with their own authentication messages, ClientCertificate and ClientCertificateVerify to achieve mutual authentication. Finally, $\mathcal{C}$ must confirm their view of the handshake by sending a MAC over the handshake transcript in the ClientFinished (CF) message. The MAC generation must also be computed in 2PC with $\mathcal{V}$.

It is only after this process that the handshake is completed, and $\mathcal{C}$ and $\mathcal{S}$ can derive the keying material required by the subsequent *record layer* to exchange authenticated and encrypted application data. This derivation is again performed in 2PC, and $\mathcal{C}$ and $\mathcal{V}$ essentially both hold shares of all the secret parameters needed to encrypt traffic using the specified encryption ciphersuite. In this work, we specifically target AES-GCM, since over 90% of TLS 1.3 traffic uses this ciphersuite [16]. This means that $\mathcal{C}$ and $\mathcal{V}$ hold shares of the necessary session key and auxiliary data that is used for encrypting traffic using this ciphersuite.

Query execution phase. $\mathcal{C}$ sends a query q (in encrypted form $\hat{\mathsf{q}}$) to $\mathcal{S}$ with help from $\mathcal{V}$. Specifically, since the session keys are secret-shared, $\mathcal{C}$ and $\mathcal{V}$ jointly compute the encryptions of these queries in 2PC. Encrypted responses, $\hat{\mathsf{r}}$, can then be decrypted using a similar procedure to reveal the server response r to $\mathcal{C}$. This is important for running tools in a browser, or any multi-round protocol where subsequent queries depend on previous responses.

Commitment phase. After querying $\mathcal{S}$ and receiving a response r, $\mathcal{C}$ commits to the session by forwarding the ciphertexts to $\mathcal{V}$, and receives $\mathcal{V}$'s session key shares in exchange. Hence, $\mathcal{C}$ can verify the integrity of r, and later prove statements about it in zero-knowledge. The fact that $\mathcal{C}$ sent their commitment before they knew the shares of $\mathcal{V}$ means that $\mathcal{V}$ can trust that such zero-knowledge proofs are generated honestly.

Limitations of previous DCTLS protocols. Existing DCTLS schemes have serious security, performance, and usability limitations. They either only work with old/deprecated TLS versions (1.2 and under) and offer no privacy from the oracle (PageSigner [17]); or rely on trusted hardware (Town Crier [18]), against which various attacks have recently emerged [19]. Another class of oracle schemes assumes cooperation from the server by installing TLS extensions [20], or by changing application-layer logic [21]. These approaches suffer from two fundamental problems: they break legacy compatibility, causing a significant barrier to wide adoption; and only provide conditional exportability as servers have the sole discretion to determine which data can be exported, and can censor export attempts at will.

While DECO [2] promises to solve these problems, its non-modular security design makes it difficult to swap individual pieces of functionality. This has repercussions with respect to the following parameters.

Security. Some of the primitives used by DECO have since been shown to be insecure in certain settings [22], [23], and the security proof only targets TLS 1.2 functionality. General guidance is offered for handling TLS 1.3 sessions, but it is not formally specified.

Privacy. It is not possible to modify the handshake phase to remove explicit authentication of the server to the verifier, due to the non-modular security proof.

Performance. Certain underlying cryptographic tools (such as oblivious transfer protocols) have seen remarkable improvements subsequent to DECO's publication [24], [25]. However, certain parts of the transformations needed to handle the AES-GCM ciphersuite detailed by DECO are underspecified, and naively lead to high costs during 2PC execution.

Usability. No practical implementation of any DCTLS protocol exists (to the best of our knowledge) that can target the TLS 1.3 protocol, in settings where performance matters (e.g. during web browsing).

Overview of DiStefano. Due to the limitations of the current DCTLS protocols, we aim to build a protocol that works for the latest version of TLS, improves privacy guarantees for $\mathcal{C}$ (in the sense that client web browsing history is hidden amongst a crowd of accepted servers), does not require specific hardware or extensions, and can be easily integrated into common applications. Overall, DiStefano achieves the following.

- The creation of a commitment framework that allows for the generation of verifiable, private and anonymous claims over data communicated during TLS 1.3 sessions.
- The creation of proofs of provenance (ring signatures) of authenticated (but private) data communicated in a specific session, with a specific private TLS 1.3 server.
- The creation of a "easy-to-use" tool that comes as part of the TLS library that browsers use. This means that there is no need to use specialized hardware or installing extra extensions.

Overall, we believe that DiStefano is an essential step-forward in establishing that DCTLS protocols *can* be implemented in practice.

3. Cryptographic Preliminaries

3.1. Secure Multi-Party Computation

Multi-party computation (MPC) protocols allow a group of parties $p_1, \ldots, p_n$, each holding private inputs $s_1, \ldots, s_n$, to jointly compute a function $f(s_1, \ldots s_n)$, without leaking any information other than the output of f. Security of these protocols considers an adversary that corrupts at most t out of n parties (n are the number of parties that hold the private inputs), and attempts to learn the private input of honest participants. Two-party secure computation (2PC) refers to a class of these protocols in which $n = 2$ and $t = 1$ [26]. There are two common approaches for 2PC protocols. *Garbled circuits protocols* [27], [28] encode f as a boolean circuit and evaluate an encrypted variant of the circuit across two parties. *Threshold secret-sharing* protocols (e.g. SPDZ [29], [30], or MASCOT [31]), typically operate by first producing some random multiplicative triples (referred to as *Beaver triples* [32]) before additively sharing secret inputs with some extra information.

Garbled circuit protocols are particularly well-suited to secure evaluation of binary circuits, such as AES or SHA-256. The cost of a garbled circuit is normally evaluated in terms of the number of AND gates due to the *Free-XOR* optimisation [33]. In contrast, threshold secret-sharing schemes are typically well-suited for computing arithmetic operations, such as modular exponentiation. We calculate their cost in terms of their number of rounds and bandwidth requirements.

MPC primitives. We use both types of 2PC protocols: we use the maliciously secure authenticated garbling implementation provided by emp [34] for binary operations; and we base our 2PC arithmetic operations on the well-known *oblivious transfer* (OT) primitive.

Definition 1 (Oblivious Transfer (OT)). *An oblivious transfer scheme,* OT*, is a tuple of algorithms:*

- $\mathsf{OT.Gen}(1^\lambda)$*: outputs any key material.*
- $\mathsf{OT.Exec}(m_0, m_1, b)$*: accepts* m_0, m_1 *from* P_1 *and* b *from* P_2*.* P_2 *learns* m_b*, and* P_1 *learns nothing.*

We realise the OT functionality by using the actively secure IKNP [35], [36] extension and the Ferret [24] OT scheme, both provided by emp. The security of these constructions rely on the security of information theoretic MACs and the hardness of the *learning parity with noise* (LPN) problem, respectively, as well as randomness assumptions about hash functions, see e.g. [37].

Using OT as a building block, we realise the remaining 2PC functionality needed by using *multiplicative-to-additive* (MtA) secret sharing schemes.

Definition 2 (MtA)**.** *An MtA scheme,* MtA*, is a tuple of the following algorithms:*

- $\mathsf{MtA.Gen}(1^\lambda)$*: outputs any needed key material.*
- $\mathsf{MtA.Mul}(\alpha,\beta)$*: each* P_i *supplies* a_i*, learning as output* b_i *such that* $\sum b_i = \Pi_i a_i$*.*

A maliciously secure MtA scheme augments this definition with an additional algorithm, $\mathsf{MtA.Check}(a_1,\ldots,b_1,\ldots)$, to check shares consistency.

Existing works in the space [2], [5] realise the MtA functionality with an approach [38] based on Paillier encryption [39]. We deviate from this approach to improve efficiency [40, §5] and to mitigate the need for range proofs [22], [23]. Specifically, we realise the MtA functionality using the schemes introduced in [41] and [42], [43] for rings of characteristic > 2 and for rings of characteristic 2, respectively. The schemes require access to OT functionality and are instantiated with 128 bit statistical and computational security. We note that whilst the security of [42], [43] reduces directly to an NP-hard encoding problem [44], to the best of our knowledge, there is no computational hardness proof for [41].

ECtF. During the *Key Exchange* phase of the handshake of DCTLS, both $\mathcal{V}$ and $\mathcal{C}$ hold additive shares Z_v and Z_c of a shared ECDH key $(x,y) = \text{DHE}$. Given that all key derivation operations are carried out on the x co-ordinate of Z, we use the *elliptic curve to field* (ECtF) functionality [2] to produce additive shares t_v and t_c of the x coordinate, which is an element in $\mathbb{F}_q$. Using these shares as inputs to the subsequent 2PC operations to derive the handshake secrets allows running all computation in a binary circuit, which results in a substantial performance improvement when compared with attempting to combine arithmetic and binary approaches in a garbled circuit. We stress that use of the ECtF functionality enables substantial performance improvements: for example, we estimate that computing just the x co-ordinate of $Z_v + Z_c$ in a garbled circuit would be more expensive than deriving all TLS session secrets, requiring around 1.7M AND gates for an elliptic curve over a field with a 256-bit prime. From a security perspective, we remark that the security of the ECtF functionality reduces the security of the underlying secure multiplication protocol. Thus, we achieve malicious security by instantiating the multiplication with a maliciously secure MtA scheme.

3.2. Ring Signature Schemes

Ring signature schemes were first defined by Rivest, Shamir, and Taurman [45], and allow an individual that is part of a "ring" of n possible signers to generate a signature that is indistinguishable from a signature generated by any of the other members. Notably, ring signatures do not allow the signing to be revealed unless the signer explicitly decides to reveal themselves. We give a formal definition of ring signature schemes and their security properties below.

```
Anon
$\{\mathsf{sk}_i,\mathsf{vk}_i\}_{i\in[n]} \leftarrow\!\!\$\ \Pi.\mathsf{Gen}(1^\lambda)$
$(m,R,i_0,i_1) \leftarrow \mathcal{A}^{\mathcal{O}_\mathsf{S},\mathcal{O}_\mathsf{C}}(\{\mathsf{vk}_i\}_{i\in[n]})$
if $[(i_0,i_1 \notin [n]) \vee (\mathsf{vk}_{i_0},\mathsf{vk}_{i_1} \notin R)]$: abort
$d \leftarrow\!\!\$\ \{0,1\}$
$\sigma \leftarrow \Pi.\mathsf{Sign}(\mathsf{sk}_{i_d}, m, R)$
$d' \leftarrow \mathcal{A}(\sigma)$
if $[d' \overset{?}{=} d]$: return $1$
return $0$

Unf
$\{\mathsf{sk}_i,\mathsf{vk}_i\}_{i\in[n]} \leftarrow\!\!\$\ \Pi.\mathsf{Gen}(1^\lambda)$
$(m^*,R^*,\sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}_\mathsf{S},\mathcal{O}_\mathsf{C}}(R = \{\mathsf{vk}_i\}_{i\in[n]})$
if $[(R^* \not\subseteq R) \vee$
    $(\exists\, i' \in \mathcal{Q}_\mathsf{C}$ s.t. $\mathsf{vk}_{i'} \in R^*) \vee$
    $(m^* \in \mathcal{Q}_\mathsf{S})]$: abort
return $\Pi.\mathsf{Verify}(R^*, \sigma^*, m^*)$
```

Figure 3. Security games for establishing anonymity and unforgeability guarantees of a ring signature scheme Π.

Definition 3 (Ring signatures)**.** *A ring signature scheme,* Π*, is a tuple of the following algorithms:*

- $\Pi.\mathsf{Gen}(1^\lambda)$*: outputs keys* $(\mathsf{sk},\mathsf{vk})$*;*
- $\Pi.\mathsf{Sign}(\mathsf{sk}, m, R = \{\mathsf{vk}_i\}_{i\in[n]})$*: outputs a signature* σ *under key* sk *of a message* m*, with respect to the ring* R*;*
- $\Pi.\mathsf{Verify}(R = \{\mathsf{vk}_i\}_{i\in[n]}, \sigma, m)$*: outputs a bit* $b \in \{0,1\}$*, where* $b = 1$ *indicates successful verification, and* $b = 0$ *indicates failure.*

Firstly, we say that Π is *complete* if, for any set of keys $\{(\mathsf{sk}_i,\mathsf{vk}_i) \leftarrow\!\!\$\ \Pi.\mathsf{Gen}(1^\lambda)\}_{i\in[n]}$, $j \in [n]$, message m, the ring $R = \{\mathsf{vk}_i\}_{i\in[n]}$, and $\sigma \leftarrow \Pi.\mathsf{Sign}(\mathsf{sk}_j, m, R)$, then $1 \leftarrow \Pi.\mathsf{Verify}(R, \sigma, m)$.

Secondly, let Anon and Unf be the security games defined in Fig. 3. We say that Π is *anonymous* (resp. *unforgeable*) if the advantage of a PPT algorithm, $\mathcal{A}$, in either game is negligible.

In both games, the adversary has access to the following oracles.[8]

- $\mathcal{O}_\mathsf{S}$: takes as input an index i, a message m', and a ring R', and returns $\sigma \leftarrow \Pi.\mathsf{Sign}(\mathsf{sk}_i, m', R')$;
- $\mathcal{O}_\mathsf{C}$: takes as input an index i, and returns the randomness used to generate vk_i.

[8] Note that both oracle definitions assume the generation of a global set of key pairs that are used during the security game, and a correspondingly global ring, R, of all valid verification keys.

Furthermore, let $\mathcal{Q}_{\mathsf{S}}$ and $\mathcal{Q}_{\mathsf{C}}$ be the sets of queries sent to $\mathcal{O}_{\mathsf{S}}$ and $\mathcal{O}_{\mathsf{C}}$, respectively.

Instantiations. It is possible to instantiate the required functionality with a specific ring signature scheme known as *ZKAttest* [46]. This scheme generates signatures under ECDSA private keys that preserve anonymity amongst a "ring" of known ECDSA verification keys.

3.3. Commitment Schemes

Definition 4 (Commitment scheme). *A commitment scheme* Γ *is a tuple consisting of the following algorithms:*

- $\Gamma.\mathsf{Gen}(1^\lambda)$*: outputs some secret parameters* sp*;*
- $\Gamma.\mathsf{Commit}(\mathsf{sp}, x)$*: outputs a commitment* c*;*
- $\Gamma.\mathsf{Challenge}(c)$*: outputs a random challenge* t*;*
- $\Gamma.\mathsf{Open}(\mathsf{sp}, c, t, x)$*: outputs a bit* $b \in \{0, 1\}$*.*

An interactive commitment scheme, $\widetilde{\Gamma}$, between a committer, $\mathcal{C}$, and a revealer, $\mathcal{R}$, proceeds as follows:

- $\mathcal{C}$ runs $\mathsf{sp} \leftarrow \Gamma.\mathsf{Gen}(1^\lambda)$, and sends $c \leftarrow \Gamma.\mathsf{Commit}(\mathsf{sp}, x)$ to $\mathcal{R}$;
- $\mathcal{R}$ sends $t \leftarrow \Gamma.\mathsf{Challenge}(c)$ to $\mathcal{C}$;
- $\mathcal{C}$ sends x to $\mathcal{R}$;
- $\mathcal{R}$ computes $b \leftarrow \Gamma.\mathsf{Open}(\mathsf{sp}, c, t, x)$, and outputs $b \overset{?}{=} 1$.

Definition 5 (Binding property). *Given* $\mathsf{sp} \leftarrow \Gamma.\mathsf{Gen}(1^\lambda)$. *We say that* Γ *is a* computationally binding *commitment scheme if, for any PPT algorithm, the following holds:*

$$\Pr\left[0 \leftarrow \Gamma.\mathsf{Open}(\mathsf{sp}, c^*, t^*, x') \;\middle|\; \substack{(x^*, c^*) \leftarrow \mathcal{A}(1^\lambda) \\ t^* \leftarrow \Gamma.\mathsf{Challenge}(c^*) \\ x' \leftarrow \mathcal{A}(1^\lambda); x' \neq x^*}\right] > 1 - \mathsf{negl}(\lambda).$$

We say that Γ *is* perfectly binding *if the same holds for unbounded algorithms, with probability 1.*

Definition 6 (Hiding property). *Let* $\mathsf{sp} \leftarrow \Gamma.\mathsf{Gen}(1^\lambda)$*,* $\{x_b\}_{b \in \{0,1\}} \in \{0, 1\}^2$*, and* $\{c_b \leftarrow \Gamma.\mathsf{Commit}(\mathsf{sp}, x_b)\}$. *We say that* Γ *is a* computationally hiding *commitment scheme if, for any PPT algorithm, the following holds:*

$$\Pr\left[d^* \overset{?}{=} d \;\middle|\; \substack{d \leftarrow \$ \{0,1\} \\ d^* \leftarrow \mathcal{A}(1^\lambda, c_d, (x_0, x_1))}\right] < 1/2 + \mathsf{negl}(\lambda).$$

We say that Γ *is* perfectly hiding *if the same holds for unbounded algorithms, with probability 1/2.*

We will show in Section 4.4 that the commitment phase of DiStefano is a perfectly binding and computationally hiding commitment scheme for TLS 1.3-encrypted data. In other words, the client can commit (to a third-party verifier) to encrypted traffic.

3.4. Authenticated Encryption

An authenticated encryption with associated data (AEAD) scheme considers a keyspace $\mathcal{K}$, a message space $\mathcal{M}$, a ciphertext space $\mathcal{X}$, and a tag space $\mathcal{T}$, and is defined using the following algorithms.

- $k \leftarrow \mathsf{AEAD}\,.\mathsf{keygen}(1^\lambda)$: Outputs a key $k \leftarrow\!\$\, \mathcal{K}$.
- $(C, \tau) \leftarrow \mathsf{AEAD}\,.\mathsf{Enc}(k, m; A)$: For a key $k \in \mathcal{K}$, message $m \in \mathcal{M}$, and associated data $A \in \{0, 1\}^*$, outputs a ciphertext $C \in \mathcal{X}$ and a tag $\tau \in \mathcal{T}$.
- $m \vee \perp \leftarrow \mathsf{AEAD}\,.\mathsf{Dec}(k, C, \tau; A)$: For a key $k \in \mathcal{K}$, ciphertext $C \in \mathcal{M}$, tag $\tau \in \mathcal{T}$, and associated data $A \in \{0, 1\}^*$, outputs a message $m \in \mathcal{M}$ or $\perp$.

Any AEAD scheme must satisfy the following guarantees.

Definition 7 (Correctness). AEAD *is correct if and only if:*

$$\Pr\left[m \leftarrow \mathsf{AEAD}\,.\mathsf{Dec}(k, C, \tau; A) \middle| \substack{k \leftarrow \mathsf{AEAD}\,.\mathsf{keygen}(1^\lambda) \\ (C, \tau) \leftarrow \mathsf{AEAD}\,.\mathsf{Enc}(k, m; A)}\right] = 1$$

holds true.

Definition 8 (Security). *An* AEAD *scheme is secure if it satisfies the IND-CCA notion of security [47].*

It is widely known that the AES-GCM block cipher mode of operation satisfies these guarantees [48], where $\mathcal{K} = \{0, 1\}^\lambda$, $\mathcal{M} = \{0, 1\}^*$, $\mathcal{C} = \{0, 1\}^*$. In other words, it can tolerate messages of arbitrary length and produce ciphertexts accordingly.

4. DiStefano Protocol

In this section, we specify the complete description of each of the phases of the DiStefano protocol.[9] A diagram for the full protocol can be found in Fig. 4. For comparison, we also provide a diagram of TLS 1.3 in Fig. 2. The shorthands used in both diagrams are defined in Table 1 and Table 2. The security analysis of the protocol is handled in Section 6.

4.1. Handshake Phase: HSP

We use the similar overarching mechanism for the handshake phase as described in the original DECO proposal, but focused exclusively on TLS 1.3 with AES-GCM as the AEAD scheme (Section 3.4) and using ECDH for the shared key generation. We also assume that certificates are signed using ECDSA. We note however that our protocol can be adapted to work with any other TLS 1.3-compliant ciphersuites, provided that certain operations can be carried out in 2PC.

At a high-level, we adapt the TLS 1.3 handshake by treating $\mathcal{C}$ and $\mathcal{V}$ as a *single* TLS client from the perspective of a third-party server. To achieve this, we reverse the "traditional" flow of the TLS 1.3 handshake by having $\mathcal{C}$ and $\mathcal{V}$ each prepare an additively shared ephemeral key share *SSK*, as seen in Fig. 4. This operation can be carried out cheaply without 2PC.

$\mathcal{C}$ then sends the `CH` and the `CKS` messages, advertising *SSK* as part of the `key_shares` extension. $\mathcal{S}$

[9] A formal description of each phase as an ideal functionality can be foind in Appendix A.

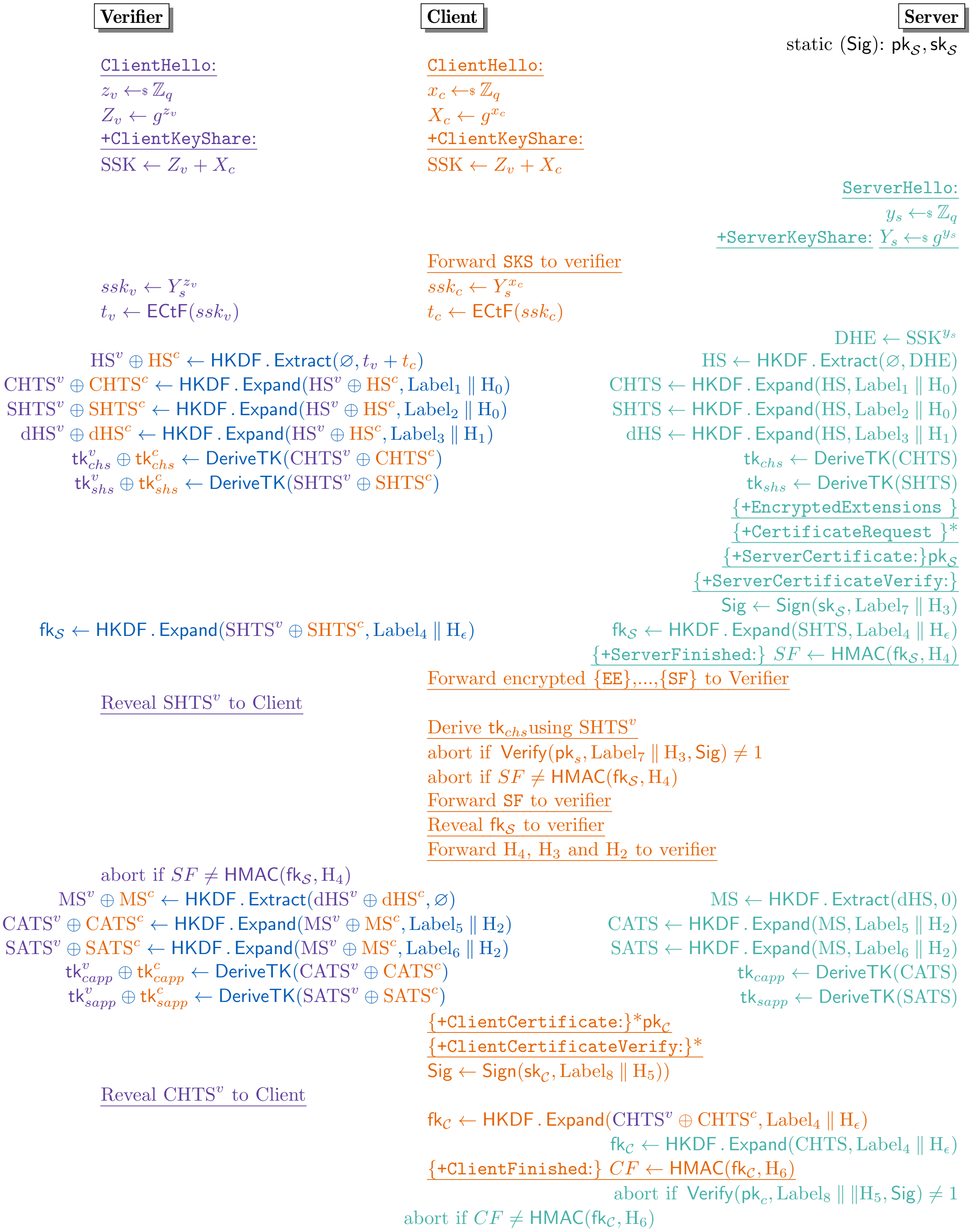
Verifier
Client
Server
static (Sig): $\mathsf{pk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}}$
ClientHello:
$z_v \leftarrow\$ \mathbb{Z}_q$
$Z_v \leftarrow g^{z_v}$
+ClientKeyShare:
$\mathrm{SSK} \leftarrow Z_v + X_c$
ClientHello:
$x_c \leftarrow\$ \mathbb{Z}_q$
$X_c \leftarrow g^{x_c}$
+ClientKeyShare:
$\mathrm{SSK} \leftarrow Z_v + X_c$
ServerHello:
$y_s \leftarrow\$ \mathbb{Z}_q$
+ServerKeyShare: $Y_s \leftarrow\$ g^{y_s}$
Forward SKS to verifier
$ssk_v \leftarrow Y_s^{z_v}$
$t_v \leftarrow \mathsf{ECtF}(ssk_v)$
$ssk_c \leftarrow Y_s^{x_c}$
$t_c \leftarrow \mathsf{ECtF}(ssk_c)$
$\mathrm{DHE} \leftarrow \mathrm{SSK}^{y_s}$
$\mathrm{HS}^v \oplus \mathrm{HS}^c \leftarrow \mathsf{HKDF.Extract}(\varnothing, t_v + t_c)$
$\mathrm{HS} \leftarrow \mathsf{HKDF.Extract}(\varnothing, \mathrm{DHE})$
$\mathrm{CHTS}^v \oplus \mathrm{CHTS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}^v \oplus \mathrm{HS}^c, \mathrm{Label}_1 \| \mathrm{H}_0)$
$\mathrm{CHTS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \mathrm{Label}_1 \| \mathrm{H}_0)$
$\mathrm{SHTS}^v \oplus \mathrm{SHTS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}^v \oplus \mathrm{HS}^c, \mathrm{Label}_2 \| \mathrm{H}_0)$
$\mathrm{SHTS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \mathrm{Label}_2 \| \mathrm{H}_0)$
$\mathrm{dHS}^v \oplus \mathrm{dHS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}^v \oplus \mathrm{HS}^c, \mathrm{Label}_3 \| \mathrm{H}_1)$
$\mathrm{dHS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{HS}, \mathrm{Label}_3 \| \mathrm{H}_1)$
$\mathsf{tk}^v_{chs} \oplus \mathsf{tk}^c_{chs} \leftarrow \mathsf{DeriveTK}(\mathrm{CHTS}^v \oplus \mathrm{CHTS}^c)$
$\mathsf{tk}_{chs} \leftarrow \mathsf{DeriveTK}(\mathrm{CHTS})$
$\mathsf{tk}^v_{shs} \oplus \mathsf{tk}^c_{shs} \leftarrow \mathsf{DeriveTK}(\mathrm{SHTS}^v \oplus \mathrm{SHTS}^c)$
$\mathsf{tk}_{shs} \leftarrow \mathsf{DeriveTK}(\mathrm{SHTS})$
{+EncryptedExtensions }
{+CertificateRequest }*
{+ServerCertificate:}$\mathsf{pk}_{\mathcal{S}}$
{+ServerCertificateVerify:}
$\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{S}}, \mathrm{Label}_7 \| \mathrm{H}_3)$
$\mathsf{fk}_{\mathcal{S}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{SHTS}^v \oplus \mathrm{SHTS}^c, \mathrm{Label}_4 \| \mathrm{H}_\epsilon)$
$\mathsf{fk}_{\mathcal{S}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{SHTS}, \mathrm{Label}_4 \| \mathrm{H}_\epsilon)$
{+ServerFinished:} $SF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$
Forward encrypted {EE},...,{SF} to Verifier
Reveal SHTS^v to Client
Derive tk_{chs}using SHTS^v
abort if $\mathsf{Verify}(\mathsf{pk}_s, \mathrm{Label}_7 \| \mathrm{H}_3, \mathsf{Sig}) \neq 1$
abort if $SF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$
Forward SF to verifier
Reveal $\mathsf{fk}_{\mathcal{S}}$ to verifier
Forward H_4, H_3 and H_2 to verifier
abort if $SF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{S}}, \mathrm{H}_4)$
$\mathrm{MS}^v \oplus \mathrm{MS}^c \leftarrow \mathsf{HKDF.Extract}(\mathrm{dHS}^v \oplus \mathrm{dHS}^c, \varnothing)$
$\mathrm{MS} \leftarrow \mathsf{HKDF.Extract}(\mathrm{dHS}, 0)$
$\mathrm{CATS}^v \oplus \mathrm{CATS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}^v \oplus \mathrm{MS}^c, \mathrm{Label}_5 \| \mathrm{H}_2)$
$\mathrm{CATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \mathrm{Label}_5 \| \mathrm{H}_2)$
$\mathrm{SATS}^v \oplus \mathrm{SATS}^c \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}^v \oplus \mathrm{MS}^c, \mathrm{Label}_6 \| \mathrm{H}_2)$
$\mathrm{SATS} \leftarrow \mathsf{HKDF.Expand}(\mathrm{MS}, \mathrm{Label}_6 \| \mathrm{H}_2)$
$\mathsf{tk}^v_{capp} \oplus \mathsf{tk}^c_{capp} \leftarrow \mathsf{DeriveTK}(\mathrm{CATS}^v \oplus \mathrm{CATS}^c)$
$\mathsf{tk}_{capp} \leftarrow \mathsf{DeriveTK}(\mathrm{CATS})$
$\mathsf{tk}^v_{sapp} \oplus \mathsf{tk}^c_{sapp} \leftarrow \mathsf{DeriveTK}(\mathrm{SATS}^v \oplus \mathrm{SATS}^c)$
$\mathsf{tk}_{sapp} \leftarrow \mathsf{DeriveTK}(\mathrm{SATS})$
{+ClientCertificate:}*$\mathsf{pk}_{\mathcal{C}}$
{+ClientCertificateVerify:}*
$\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathcal{C}}, \mathrm{Label}_8 \| \mathrm{H}_5))$
Reveal CHTS^v to Client
$\mathsf{fk}_{\mathcal{C}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{CHTS}^v \oplus \mathrm{CHTS}^c, \mathrm{Label}_4 \| \mathrm{H}_\epsilon)$
$\mathsf{fk}_{\mathcal{C}} \leftarrow \mathsf{HKDF.Expand}(\mathrm{CHTS}, \mathrm{Label}_4 \| \mathrm{H}_\epsilon)$
{+ClientFinished:} $CF \leftarrow \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6)$
abort if $\mathsf{Verify}(\mathsf{pk}_c, \mathrm{Label}_8 \| \|\mathrm{H}_5, \mathsf{Sig}) \neq 1$
abort if $CF \neq \mathsf{HMAC}(\mathsf{fk}_{\mathcal{C}}, \mathrm{H}_6)$

Figure 4. The DiStefano DCTLS handshake (full 1-RTT handshake protocol). Shorthands correspond to those defined in [10]. Purple represents messages sent or calculated by the verifier, orange represents messages sent or calculated by the client, green represents messages sent or calculated by the server, and blue represents calculations executed by both $\mathcal{C}$ and $\mathcal{V}$, using 2PC.

then processes these messages and, in turn, sends a SH message back to $\mathcal{C}$ containing a freshly sampled ECDH key_share Z_s. At this stage, $\mathcal{S}$ computes the shared ECDH key as $E = x_s \cdot SSK$ and continues to derive all traffic secrets (i.e. $\mathrm{CHTS}, \mathrm{SHTS}, \mathsf{tk}_{shs}, \mathsf{tk}_{chs}$). Once $\mathcal{C}$ and $\mathcal{V}$ receive the SH message, they derive additive shares of the shared ECDH key as $E = x_c \cdot Y_s + x_v \cdot Y_s$. As TLS 1.3 key derivation operates on the x co-ordinate of the shared key, $\mathcal{C}$ and $\mathcal{V}$ convert their additive shares of $E = (E_x, E_y)$ into additive shares $E_x = t_c + t_v$ by running the ECtF functionality. With E_x computed, $\mathcal{C}$ and $\mathcal{V}$ proceed to run the TLS 1.3 handshake key derivation circuit in 2PC, with each party learning shares $\mathrm{HS}^v \oplus \mathrm{HS}^c = \mathsf{HKDF.Extract}(\varnothing, t_v + t_c)$. In practice, this process is carried out inside a garbled circuit that also produces shares of CHTS, SHTS and dHS, as well as the $\mathtt{SF}$ message key $\mathsf{fk}_{\mathcal{S}}$. This key is provided to both $\mathcal{C}$ and $\mathcal{V}$. This circuit comprises of around 800K AND gates, which is similar to DECO's circuit size for TLS 1.2. We delay the derivation of the traffic keys until the next stage, as it provides authenticity guarantees to $\mathcal{V}$.

Authentication phase. $\mathcal{S}$ sends the $\mathtt{CR}$ (if wanted for client authentication), $\mathtt{SCRT}$, $\mathtt{SCV}$ and $\mathtt{SF}$ messages. The $\mathtt{SF}$ message is computed by first deriving a finished key $\mathsf{fk}_{\mathcal{S}}$ from SHTS and then computing a MAC tag SF over a hash of all messages of the handshake up to that point. At this point, $\mathcal{S}$ is able to compute the client application traffic secret, CATS, and the server application traffic secret, SATS. $\mathcal{S}$ can also start sending encrypted application data (encrypted under tk_{sapp}) without waiting for the final flight of $\mathcal{C}$ messages.

$\mathcal{C}$ receives the encrypted messages from $\mathcal{S}$ and, in turn, forwards them (in an encrypted form) to $\mathcal{V}$ alongside a commitment to their share of SHTS. This commitment is necessary to make AES-GCM act as a committing cipher from the perspective of $\mathcal{V}$, which allows $\mathcal{V}$ to disclose their shares of CHTS and SHTS to $\mathcal{C}$ without compromising authenticity guarantees. As $\mathcal{C}$ now knows the entirety of CHTS and SHTS, they are able to locally derive the handshake keys tk_{chs} and tk_{shs}, allowing them to check $\mathcal{S}$'s certificate and $\mathtt{SF}$ messages without the involvement of $\mathcal{V}$. Moreover, as $\mathcal{C}$ now knows tk_{chs} they are also able to respond to the $\mathtt{CR}$ if one exists. $\mathcal{C}$ then forwards an authentic copy of the hashes H_4, H_3 and H_2 to $\mathcal{V}$, allowing them to check the $\mathtt{SF}$ message's authenticity. For performance reasons, we delegate this particular check to the proving stage of the protocol. Notice that $\mathcal{C}$ does not forward the decrypted $\mathtt{SCRT}$ message to $\mathcal{V}$, as this message reveals the identity of the server with which $\mathcal{C}$ is communicating. At this point, though, $\mathcal{C}$ can construct a ring signature based on the server's certificate signature, and send it to $\mathcal{V}$ (it can also perform this operation later in the session). $\mathcal{V}$, then, can check the validity of the ring signature (if present). Similarly, $\mathcal{C}$ can selectively reveal the blocks containing the $\mathtt{SF}$ message, allowing $\mathcal{V}$ to validate the $\mathtt{SF}$. Finally, $\mathcal{C}$ and $\mathcal{V}$ derive the shares of the traffic secrets MS, CATS, SATS and the traffic keys $\mathsf{tk}_{sapp}, \mathsf{tk}_{capp}$ in 2PC. In practice, we instantiate this derivation as a garbled circuit that contains around 700K AND gates. Note that this circuit cannot cheaply be combined with the handshake secret derivation circuit, as deriving the traffic keys requires a hash of the unencrypted handshake transcript: this would require decrypting and hashing large messages inside a garbled circuit, which is rather expensive.

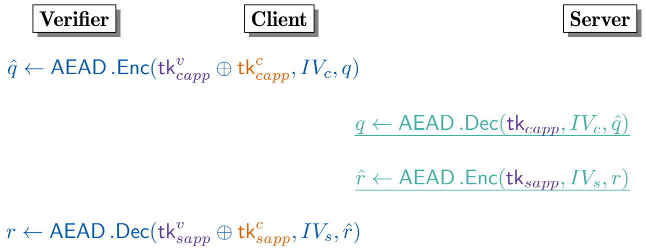
Verifier
Client
Server
$\hat{q} \leftarrow \mathsf{AEAD.Enc}(\mathsf{tk}^v_{capp} \oplus \mathsf{tk}^c_{capp}, IV_c, q)$
$q \leftarrow \mathsf{AEAD.Dec}(\mathsf{tk}_{capp}, IV_c, \hat{q})$
$\hat{r} \leftarrow \mathsf{AEAD.Enc}(\mathsf{tk}_{sapp}, IV_s, r)$
$r \leftarrow \mathsf{AEAD.Dec}(\mathsf{tk}^v_{sapp} \oplus \mathsf{tk}^c_{sapp}, IV_s, \hat{r})$

Figure 5. The DiStefano query execution protocol. Purple represents messages sent or calculated by the verifier, orange represents messages sent or calculated by the client, green represents messages sent or calculated by the server, and blue represents calculations executed by both $\mathcal{C}$ and $\mathcal{V}$, using 2PC.

4.2. Query Execution Phase: QP

Once HSP has completed, $\mathcal{C}$ and $\mathcal{V}$ move into the querying phase Fig. 5. For simplicity, we describe this portion of the protocol in terms of a single round of queries before extending the phase to multiple rounds.

During the query phase of the protocol, $\mathcal{C}$ produces a series of queries $q = q_1, \ldots, q_n$ and jointly encrypts these with $\mathcal{V}$, with both parties learning a vector of ciphertexts $\hat{q}$ as output. Then, $\mathcal{C}$ forwards $\hat{q}$ to $\mathcal{S}$, receiving an encrypted response $\hat{r}$ in exchange. At this stage of the protocol, $\mathcal{C}$ forwards $\hat{r}$ to $\mathcal{V}$ so that both parties may verify the tags on $\hat{r}$: both parties learn a single bit indicating if the tag check passed or not.

In practice, we instantiate this portion of the protocol using the AES-GCM approach described in Section 5. Note that as this approach is simply an instantiation of the DCTLS.QP functionality given in Appendix A, there is no explicit dependence on AES-GCM: any AEAD cipher supported by TLS 1.3 will suffice. We highlight this, and the general security formalisation of the query phase, in Section 6.

Extending the querying phase to multiple rounds is rather straightforward using AES-GCM. We discuss the details of committing to ciphertexts in Section 4.4, but the main idea is that, as each ciphertext block q_i is encrypted with a unique key $\mathsf{AES.Enc}(k, IV + i)$, $\mathcal{C}$ and $\mathcal{V}$ can arbitrarily reveal their shares of $k_i^c + k_i^v = \mathsf{AES.Enc}(k, IV + i)$ at any stage of the querying phase provided an appropriate commitment has been made beforehand. As k_i^c and k_i^v are ephemeral keys, revealing them does not compromise the shares derived during the HSP. The security of this approach directly reduces to the difficulty of recovering an AES key from many known plaintext/ciphertext pairs. This extension permits many useful applications, as $\mathcal{C}$ and $\mathcal{V}$ can now nest commitment rounds inside of the query phase. This may allow for real-world applications to take better advantage of DiStefano.

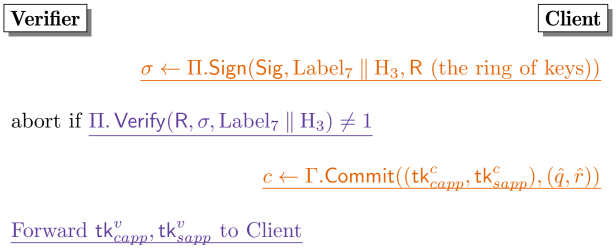
Verifier
Client
$\sigma \leftarrow \Pi.\mathsf{Sign}(\mathsf{Sig}, \mathrm{Label}_7 \parallel \mathrm{H}_3, \mathrm{R}\ \text{(the ring of keys)})$
abort if $\Pi.\mathsf{Verify}(\mathrm{R}, \sigma, \mathrm{Label}_7 \parallel \mathrm{H}_3) \neq 1$
$c \leftarrow \Gamma.\mathsf{Commit}((\mathsf{tk}^c_{capp}, \mathsf{tk}^c_{sapp}), (\hat{q}, \hat{r}))$
Forward $\mathsf{tk}^v_{capp}, \mathsf{tk}^v_{sapp}$ to Client

Figure 6. The DiStefano commitment protocol, assuming a commitment scheme, Γ, and a ring signature scheme, Π, for ECDSA signatures. Purple represents messages sent or calculated by the verifier, orange represents messages sent or calculated by the client.

4.3. Commitment Phase: CP

The objectives of the commitment phase are twofold: i. to assure $\mathcal{V}$ of the authenticity of the server without revealing which server $\mathcal{C}$ is communicating with, ii. to allow $\mathcal{C}$ to learn secret shares held by $\mathcal{V}$ only after they have been committed in a binding way to a specific portion of the TLS data communicated with $\mathcal{S}$.

For the first, $\mathcal{V}$ authenticates (in zero-knowledge) the TLS server that they communicated and established a TLS 1.3 handshake with, as one of N servers from which $\mathcal{V}$ accepts attestations.[10] For the second, $\mathcal{C}$ and the $\mathcal{V}$ take part in a commitment protocol, that allows $\mathcal{C}$ to learn the secret shares held by $\mathcal{V}$, but only after they have committed (in a binding way) to some specific portion of data exchanged during the TLS session with $\mathcal{S}$. Overall, this commitment (that also hides from $\mathcal{V}$ the exact data that was exchanged) would allow subsequent proofs to be communicated. This allows, for example, to prove knowledge of committed data using 2PC, zero-knowledge proofs, or selective opening procedures (see [2] for concrete examples). A visual representation of this phase can be found on Fig. 6.

4.4. Commitment to AES-GCM ciphertexts

We describe how $\mathcal{C}$ forms valid commitments to AES-GCM ciphertexts. Note that although the protocol described here is sufficient for our purposes, more sophisticated approaches may be needed for more general proofs. Furthermore, our eventual construction relies on performance optimisations that are used for running the AES-GCM functionality in 2PC, that are described in Section 5.

Recall that, in DiStefano, both $\mathcal{C}$ and $\mathcal{V}$ learn all AES-GCM ciphertext blocks $C_i = M_i \oplus \mathsf{AES.Enc}(k, IV + i)$ produced by $\mathcal{S}$. As $k = k^c + k^v$ is secret shared, $\mathcal{C}$ and $\mathcal{V}$ check the integrity of the C_is using the approach described in Section 5. Assuming this check is valid, $\mathcal{C}$ locally generates n masks b^c_i and commits to each mask individually. This early commitment is necessary to circumvent the non-committing nature of AES-GCM [49]. After these commitments have been made, $\mathcal{C}$ and $\mathcal{V}$ then call into a garbled circuit, receiving additive shares of $b^c_i + b^v_i = \mathsf{AES.Enc}(k, IV + i)$. In other words, $\mathcal{V}$ learns $\mathsf{AES.Enc}(k, IV + i) + b^c_i$, and $\mathcal{C}$ learns nothing new. Finally, $\mathcal{C}$ learns each k^v_i. As i is variable, this process can be straightforwardly scaled to allow $\mathcal{C}$ to commit to only a subset of these blocks. Moreover, committing to shares in this way allows $\mathcal{C}$ to prove simple statements about individual blocks C_i. For example, $\mathcal{C}$ can reveal any block C_i by revealing their share b^c_i to $\mathcal{V}$, allowing $\mathcal{V}$ to decrypt C_i. This technique is referred to as *selective revealing* in the DECO protocol: we provide more details on the uses of this technique in Appendix E.

Authentication of $\mathcal{S}$ to $\mathcal{V}$. With the previously described technique, $\mathcal{C}$ performs 1-out-of-N authentication of the server to $\mathcal{V}$. Using the HSP, $\mathcal{C}$ commits to the encrypted `SCV` and `SF` messages, and learns the entirety of SHTS. At this stage, $\mathcal{C}$, then, forwards authentic copies of H_2, H_3 and H_4 to $\mathcal{V}$, and proves, using a ring signature scheme (for example, ZKAttest [46], see Appendix B), that `SCV` is a valid signature over H_3 from a trusted server. For their part, $\mathcal{V}$ checks the authenticity of `SF` using the supplied value of H_4 and the validity of the proof using H_3. Note that the authenticity of H_4 and H_3 is guaranteed by the proof of a valid signature [50, §3.1], that in turn authenticates the handshake transcript.

We provide a detailed background on how ZKAttest satisfies the guarantees of the ring signature scheme that we require in Appendix B. Note that it is *imperative* that a higher-level system enforces a public list of $n > 1$ (preferably $n \gg 1$) servers (available with their associated public keys) that have to be accepted. This is to prevent a malicious $\mathcal{V}$ from suggesting a ring of size one, which defeats the purpose of using a ring signature scheme and would allow deanonymisation attacks.

Session Commitment Protocol. Once the authentication protocol in Section 4.4 is carried out, $\mathcal{C}$ can now commit to and reveal certain information about the application traffic they witness. First, we define a commitment scheme (Γ) that can be implicitly constructed using the outputs of QP, using the following algorithms.

- $(\hat{q}_i, \hat{r}_i) \leftarrow \Gamma.\mathsf{Commit}(\mathsf{sp}_{\mathcal{C}}, (\hat{q}, \hat{r}, i))$: For the input i, output the ciphertexts $(\hat{q}_i, \hat{r}_i)$ corresponding to the i^{th} query q_i, and the response $r[i]$ (exchanged during the query phase).
- $\mathsf{sp}_{\mathcal{V}} \leftarrow \Gamma.\mathsf{Challenge}(c)$: Output the secret parameters of $\mathcal{V}$.
- $b \leftarrow \Gamma.\mathsf{Open}((\mathsf{sp}_{\mathcal{C}}, \mathsf{sp}_{\mathcal{V}}), (\hat{q}_i, \hat{r}_i), (q_i, r_i))$: Check that $(\hat{q}_i, \hat{r}_i)$ decrypts to (q_i, r_i), and output $b = 1$ on success, and $b = 0$ otherwise.

In this commitment scheme, the client simply commits to encrypted TLS traffic exchanged during the

[10] This phase could be performed during the handshake protocol. However, for performance reasons, it is preferable to have this proof be communicated in the commitment phase, when online communication is no longer constrained by potential server timeouts during the handshake.

query phase (using 2PC to encrypt and decrypt the traffic). When it comes to opening the encrypted application traffic, the protocol requires $\mathcal{V}$ to send their TLS key secret shares, so that $\mathcal{C}$ can decrypt and then reveal the plaintext values (that were previously encrypted).

It is important to note that in the real DiStefano protocol, $\mathcal{C}$ does not send any unencrypted values to $\mathcal{V}$. Instead, both parties should execute a protocol that proves certain facts about the traffic without revealing anything else. This could be done using a combination of 2PC or zero-knowledge proofs (as is used in the DECO protocol). The formal commitment opening process that we describe above allows for this, since $\mathcal{C}$ can now use the combined secret parameters $(\mathsf{sp}_{\mathcal{C}}, \mathsf{sp}_{\mathcal{V}})$ to prove any statement about the commitment $(\hat{q}_i, \hat{r}_i)$. We prove that Γ is a perfectly binding, and computationally hiding commitment scheme on Appendix D.

4.5. Subsequent Phases

One of the goals of DCTLS protocols we presented in this work is that they allow for facts to be proven about the transmitted encrypted data, which can be realised using zero-knowledge proofs. The DECO protocol provides two explicit mechanisms: i. revealing only a substring of the response while proving its provenance, ii. proving that the revealed substring appears in a context expected by $\mathcal{V}$. The same mechanisms can be applied to DiStefano; but we can also use anonymous credentials or two-party computation (2PC) to generate proofs of the statements on the encrypted data. More details on the techniques are found on Appendix E.

5. Computing AES-GCM in 2PC

AES-GCM is an authenticated encryption with associated data (AEAD) cipher that features prominently inside TLS implementations, with some works reporting that over 90% of all TLS1.3 traffic is encrypted using AES-GCM [16]. We use this algorithm for both encrypting the corresponding handshake messages and any application traffic. We briefly recall how AES-GCM operates before discussing its adaptation for both $\mathcal{C}$ and $\mathcal{V}$ in a 2PC setting (2PC-AES-GCM).

Encryption. Let k and IV refer to an encryption key and initialisation vector, respectively. Given as input a sequence of n appropriately padded plaintext blocks $M = (M_1, \ldots, M_n)$, AES-GCM applies counter-mode encryption to produce the ciphertext blocks $C_i = M_i \oplus \mathsf{AES.Enc}(k, IV + i)$.

To ensure authenticity, AES-GCM also outputs a tag $\tau = \mathsf{Tag}_k(IV, C, A)$ computed over both C and any associated data A as follows:

- Given some vector $\mathbf{x} \in \mathbb{F}_{2^{128}}^m$, we define the polynomial $P_x = \sum_{i=1}^{m} x_i \cdot h^{m-i+1}$ over $\mathbb{F}_{2^{128}}$.
- Assuming that C and A are properly padded, we compute τ as: $\tau(A, C, k, IV) = \mathsf{AES.Enc}(k, IV) \oplus P_{A||C||\mathsf{len}(A)||\mathsf{len}(C)}(h)$ where $h = \mathsf{AES.Enc}(k, 0)$.

Performance improvements. Despite its simplicity, executing AES-GCM encryptions in a multi-party setting can be challenging due to the use of binary and arithmetic operations. For example, whilst AES operations are well-suited for garbled circuits, a single multiplication over $\mathbb{F}_{2^{128}}$ typically requires around 16K AND gates, increasing the cost by nearly a factor of 3. To mitigate this cost, both [2] and [5] recommend computing shares of the powers of h (denoted as $\{h^i\}$) during an offline setup stage, amortising the cost across the entire session. In certain settings, this cost can be reduced further by restricting how many powers of h are used: for example, N-for-1-Auth employs a clever message slicing strategy to minimise the value of i. As this approach may not be supported by all TLS 1.3 servers, we explicitly target the largest possible TLS ciphertext of 16KiB, which corresponds to $i = 1024$.

Assuming that a sharing $(\{h_c^i\}, \{h_v^i\})$ exists, producing tags in 2PC is rather straightforward: tagging n blocks requires two local polynomial evaluations (writing $\tau_c = P_{A||C||\mathsf{len}(A)||\mathsf{len}(C)}(\{h_c^i\})$ and $\tau_v = P_{A||C||\mathsf{len}(A)||\mathsf{len}(C)}(\{h_v^i\})$, respectively) over $\mathbb{F}_{2^{128}}$ and $n + 1$ 2PC evaluations of AES [2], [5]. The final tag is achieved by simply computing $\tau = \tau_c + \tau_v \oplus \mathsf{AES.Enc}(k_c + k_v, IV_c)$. In order to make this most efficient, it is necessary to initially construct a 2PC protocol that evaluates the ciphertext C and outputs to both parties, and then have a subsequent protocol that computes the tag for this ciphertext, based on the local polynomials submitted by the client.

For simplicity's sake, we consider the ideal functionalities for encryption and decryption in 2PC-AES-GCM as given in Algorithm 1 and Algorithm 2, respectively. Notably, to include the aforementioned optimisation, we break the ideal functionality into two step: encryption/decryption and tag creation/verification. In the second step, τ_c and τ_v would be sent by both parties to the ideal functionality: this does not affect the security of the scheme, as neither party has the ability to recover any extra information. On the other hand, our ideal functionality also covers the nonce uniqueness requirement of AES-GCM. We note that in practice these additional checks do not seem to affect the running time by much: for example, our prototype garbled circuit implementation only requires around 768 extra AND gates, representing around a 10% increase over an AES circuit.

5.1. Security

We now argue the security of computing encryptions and decryptions with respect to the ideal functionalities described in Algorithm 1 and Algorithm 2. We implicitly assume that 2PC evaluations of the polynomial P and the AES functionality (using garbled circuits) are secure with respect to malicious adversaries. These security guarantees are assumed in previous work [2], [6], [7], but are not made explicit. We require them later, when

Algorithm 1 2PC-AES-GCM Encrypt

```
**Require:** $k = k_c + k_v, IV_c, IV_v, \{h_c^i\}_{i\in[n]}, \{h_v^i\}_{i\in[n]}$
**Require:** $\mathcal{C}$ inputs a message $m$
**Require:** $IV_c$ and $IV_v$ must not have been supplied for encryption previously.
**Ensure:** $\mathcal{C}$ learns $((C_1, \ldots, C_n), \tau(A, C, k, IV))$.
**Ensure:** $\mathcal{V}$ learns $(C_1, \ldots, C_n)$.
  **if** $IV_c \neq IV_v$ **then**
      **return** Error                    ▷ The IVs must match.
  **end if**
  Parse $m$ as $m_1 \| \ldots \| m_n$          ▷ $m_i$ fits AES blocksize
  $C = (C_i \leftarrow \mathsf{AES.Enc}(k_c + k_v, IV_c + i) \oplus m_i)_{i\in[n]}$
  $\tau_c \leftarrow P_{A||C||\mathsf{len}(A)||\mathsf{len}(C)}(\{h_c^i\})$
  $\tau_v \leftarrow P_{A||C||\mathsf{len}(A)||\mathsf{len}(C)}(\{h_v^i\})$
  $\tau \leftarrow \tau_c \oplus \tau_v \oplus \mathsf{AES.Enc}(k_c + k_v, IV_c)$
  **return** $(C, \tau)$ to $\mathcal{C}$
  **return** $C$ to $\mathcal{V}$
```

Algorithm 2 2PC-AES-GCM Decrypt

```
**Require:** $k = k_c + k_v, IV_c, IV_v, \{h_c^i\}, \{h_v^i\}, A$.
**Require:** Both $\mathcal{C}$ and $\mathcal{V}$ know $A$, $(C_1, \ldots, C_n)$ and $\tau$
**Require:** $IV_c$ and $IV_v$ must not have been supplied for decryption previously.
**Ensure:** $\mathcal{C}$ learns the decrypted message $m$ if $(C_1, \ldots, C_n)$ is a valid encryption wrt $\tau$.
  **if** $IV_c \neq IV_v$ **then**
      **return** Error                    ▷ The IVs must match.
  **end if**
  $\tau_c' \leftarrow P_{A||C||\mathsf{len}(A)||\mathsf{len}(C)}(\{h_c^i\})$
  $\tau_v' \leftarrow P_{A||C||\mathsf{len}(A)||\mathsf{len}(C)}(\{h_v^i\})$
  $\tau' = \tau_c' \oplus \tau_v' \oplus \mathsf{AES.Enc}(k_c + k_v, IV_c)$
  **if** $\tau' \neq \tau$ **then**
      **return** Error                          ▷ Invalid tag
  **end if**
  $m = (C_i \oplus \mathsf{AES.Enc}(k_c + k_v, IV_c + i))_{i\in[n]}$
  **return** m to $\mathcal{C}$
```

proving that the query phase of DiStefano is secure (Section 6.2).

Lemma 9 (Malicious Client)**.** *2PC-AES-GCM is secure in the presence of a malicious adversary that controls $\mathcal{C}$.*

Proof. First, let S be a PPT simulator for the encryption functionality, that simply returns samples C' from the domain of $\mathsf{AES.Enc}$, and $\tau_c \leftarrow\!\!\$ \; \{0,1\}^t$, and returns (C, τ_c) to $\mathcal{C}$. We ultimately argue that the real-world outputs of 2PC-AES-GCM is indistinguishable from this.

Let $\mathsf{S}_{\mathsf{AES}}$ be a simulator for the ideal 2PC evaluation of $\mathsf{AES.Enc}$, and let S_P be a simulator for the ideal evaluation of P. It first sends m to $\mathsf{S}_{\mathsf{AES}}$ and learns $C = (C_1, \ldots, C_n)$. Then it sends C to S_P (along with A) and learns τ. It returns (C, τ) to $\mathcal{C}$. To see that this is indistinguishable from the real-world, we can trivially construct a hybrid argument from the real-world protocol that relies on two steps, replacing real garbled circuit evaluation of each functionality with ideal-world simulation, and argue security based on the maliciously-secure 2PC garbled circuit approach that we use (Section 3.1).

Finally, based on the assumption that AES is a pseudorandom permutation, we can construct a final hybrid step, that replaces $\mathsf{AES.Enc}$ with a random value in the domain.[11].

The case of decryption is much simpler since the client only learns the message if they submit valid inputs to S (by the AEAD security guarantees of AES-GCM). This can be established using the same simulators $\mathsf{S}_{\mathsf{AES}}$ and S_P defined above. □

Lemma 10 (Malicious Verifier)**.** *2PC-AES-GCM is secure in the presence of a malicious adversary that controls $\mathcal{V}$.*

Proof. The proof for a malicious $\mathcal{V}$ follows the same structure as in the case of $\mathcal{C}$, but note that $\mathcal{V}$ is strictly less powerful, because the $\mathcal{V}$ does not submit a message to be encrypted. □

We briefly note that PageSigner follows a slightly different approach than this for computing tags: we decided not to follow their approach, as a "back-of-an-envelope" calculation suggests that it is strictly slower than the aforementioned approach. We discuss this in more detail in Appendix C.

5.2. Optimisation: Multiplicative Sharing of h

DECO gives few details on how to compute shares of the powers of h, other than that they are computed in a 2PC session. We remark that calculating these shares in a garbled circuit is unlikely to be feasible: our adapted version of N-for-1-Auth's share derivation circuit contained around 17M AND gates, and required over 900MiB and 18GiB of network traffic and memory, respectively, just for the pre-processing stage. For comparison, our circuits for TLS 1.3 secret derivation contain around 1.3M AND gates in total, which is approximately a factor of 14 smaller. Thus, using only garbled circuits is unlikely to be feasible.

Several other approaches exist for computing the shares of $\{h^i\}$. For instance, PageSigner reduces computing additive shares of h^i to simply computing shares using MtA computations. Given an initial additive sharing $h = h_c + h_v$, $\mathcal{C}$ and $\mathcal{V}$ iteratively compute additive shares of $\ell_c + \ell_v = h^n = (h_c + h_v)^{n-1}(h_c + h_v)$ for $1 < n \leq 1024$. This approach permits an additional optimisation: as $(x + y)^2 = x^2 + y^2$ over $\mathbb{F}_{2^{128}}$, each party can compute shares of even powers of h locally. Taking this optimisation into account, producing shares in this way costs a total of 1022 MtA operations. However, as computing shares of any odd h^i requires first computing shares of h^{i-2}, the approach seems to require around 500 rounds, which is likely too slow for a WAN setting.

[11] This only holds if the IV is a nonce, see [2, §B.2].

We improve upon this by replacing the additive sharing $h = h_1 + h_2$ with $h = h_1/h_2$ i.e we instead utilise a multiplicative sharing. By instead using multiplicative shares, we can run each MtA computation in parallel, with each P_i supplying $h_i, {h_i}^3, \ldots, {h_i}^{1023}$ as input. This optimisation asymptotically halves the number of MtA operations and reduces the round complexity to a single round. However, this tweak does require a slightly more complicated scheme for computing the initial sharing of h, as we now also must compute a multiplication over $\mathbb{F}_{2^{128}}$, taking the size of the circuit for deriving the initial shares to around 23K AND gates in size. In practice, we reduce the size of this circuit to around 18K AND gates by instead using a carry-less Karatsuba [51] algorithm. Whilst this still represents an increase of around a factor of 3 compared to the additive circuit, the reduction in MtA operations and rounds means that we are able to achieve an end-to-end speed-up of around a factor of 3. We discuss these results in more detail in Section 8.

6. Security Analysis

We now show that the DiStefano protocol (Section 4) is secure, using a novel modular security framework for modelling DCTLS protocols. Our framework splits the various guarantees depending on the phase. First, in Section 6.1, we show that the handshake phase is secure since a *client* ($\mathcal{C}$) and a *TLS server* ($\mathcal{S}$) can negotiate a secure TLS 1.3 session, as described in a modified version of the multi-stage key exchange model of [10]. Second, in Section 6.2, we show that the query phase of DiStefano is secure based on the 2PC-AES-GCM protocol described in Section 5. Third, in Section 6.3, for any given TLS session between $\mathcal{C}$ and $\mathcal{S}$, $\mathcal{C}$ can send commitments $\mathsf{c}_{\hat{\mathsf{q}},\hat{\mathsf{r}}}$ to $\mathcal{V}$, while authenticating $\mathcal{S}$ to $\mathcal{V}$ as one of n possible servers in a TLS 1.3 session S (while preserving the identity of $\mathcal{S}$ private). An overview of how our approach compares to previous work is given in Appendix F.

6.1. Handshake Phase Security

For establishing the security of the handshake phase, we need to show that $\mathcal{C}$ (in cooperation with $\mathcal{V}$) and $\mathcal{S}$ establish a secure TLS 1.3 channel. To do this, we use the multi-stage key exchange model of [10], which follows the Bellare-Rogaway (BR) framework [52] for establishing authenticated key exchange security based on session key indistinguishability, and builds on the multistage model of Fischlin and Günther [53], [54]. This model considers an adversary that: interacts with several concurrent TLS 1.3 sessions between different endpoints (each of which has its own identifier); can intercept, drop, and inject messages between entities; can corrupt endpoints to learn their secret parameters; and can request specific leakage of established keys. The two core security properties that an adversary is attempting to break are known as *Match Security*[12], and *Multi-Stage Security.*[13]

Algorithm 3 2PC-ECtF ideal functionality

```
Require: $ssk_c = Y_s^{x_c}$, $ssk_v = Y_s^{z_v}$
Ensure: Output shares $t_c$ to $\mathcal{C}$, and $t_v$ to $\mathcal{V}$ of the $x$-coordinate of $Z = Y_s^{x_c+z_v}$
```

Algorithm 4 2PC-DeriveTKHS ideal functionality

```
Require: $(t_c, H_0, H_1)$ from $\mathcal{C}$
Require: $(t_v)$ from $\mathcal{V}$
Ensure: For each $w \in \{c, v\}$: return $(\mathrm{HS}^w, \mathrm{CHTS}^w, \mathrm{SHTS}^w, \mathrm{dHS}^w, \mathsf{tk}^w_{chs}, \mathsf{tk}^w_{shs})$ to $\{\mathcal{C}, \mathcal{V}\}$
```

To prove the above security properties, we rely on a similar security framework to that used in Oblivious TLS [6, Section 6], that models the TLS client as a multiparty entity known as a *TLS engine*. The differences in comparison with the original model of [10] are: (1) the handshake traffic keys are leaked to the multi-stage *only* adversary when $\mathcal{C}$ is corrupted; (2) the MAC keys used in `CF` and `SF` messages are leaked to the multistage adversary upon reception of the corresponding messages; (3) the IVs are leaked to the adversary; (4) the adversary has the ability to make the engines abort; (5) the adversary is able to shift the computed secret by an arbitrary scalar Q_ϵ.

One crucial difference in our approach from the TLS engine model of [6] is in criterion (1): we only reveal handshake traffic keys when the *client* is corrupted, and not when the verifier is. It's worth recalling that criterion (5) is permitted (as it is in [6]) since $\mathcal{V}$ can arbitrarily influence the session secret by scalar multiplication. This means that the security of DiStefano is likewise based on the *Shifted PRF ODH assumption* [55]. See [6, Definition 2] for more details. We also require (as in [6]) the additional property that the adversary can only test handshake keys if both $\mathcal{C}$ and $\mathcal{V}$ of a connection are completely honest. Finally, we only allow the adversary to corrupt a single party within any given session.

To summarise, the DiStefano security model essentially provides the adversary with a subset of the capabilities of the adversary in [6]. Note, a potential strengthening of the security model could include the adversary learning the server identity when it corrupts $\mathcal{C}$. However, such information only becomes pertinent during the commitment phase, when we later consider the case of a malicious $\mathcal{V}$. Since we only allow corruption of a single entity in a single session, we do not consider this possibility during the handshake phase of the protocol.

Applying this model to DiStefano. To use the model defined above, we analyse the 2PC interaction between the client and the verifier, and show that a corrupted

[12] That any two sessions with identical identifiers will agree on the same key eventually.

[13] That any tested key is indistinguishable from a random string of the same length.

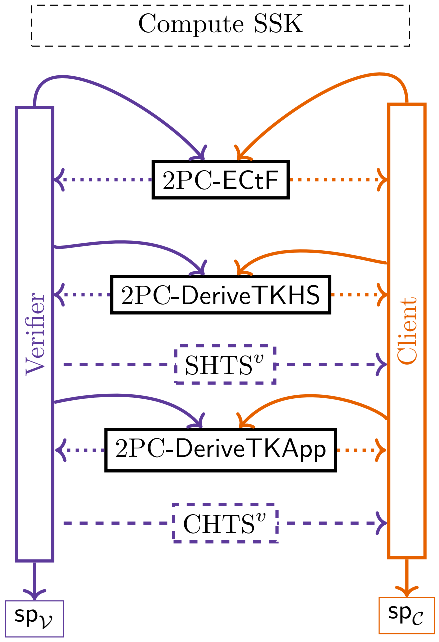

Figure 7. Ordered execution of 2PC functionalities between $\mathcal{C}$ and the $\mathcal{V}$ during the handshake phase of DiStefano. See Fig. 4 for a full description of the protocol.

```
Algorithm 5 2PC-DeriveTKApp ideal functionality
Require: (dHS$_c$) from $\mathcal{C}$
Require: (dHS$_v$) from $\mathcal{V}$
Ensure: For $w \in \{c, v\}$: return ($\mathsf{tk}^w_{capp}, \mathsf{tk}^w_{sapp}$) to $\{\mathcal{C},\mathcal{V}\}$
```

client/verifier can only learn details linked to criteria (1)–(5) above. Fig. 7 gives a summary of the 2PC interactions between $\mathcal{C}$ and $\mathcal{V}$, where Algorithm 3, Algorithm 4, and Algorithm 5 give descriptions of the ideal 2PC functionalities that are used.[14] Our proof is situated in the standard model.

First, before any 2PC takes place, the client and the verifier compute a shared value SSK $= g^{x_c+z_v}$, where x_c and z_c are the secrets of the respective participants. In this portion of the execution, it is possible for either participant to *shift* the session key by a certain scalar value, taken from the scalar field associated with the group that is being used. Criterion (5) captures this capability for an adversary, by allowing them to shift the eventual shared secret by a scalar value once they have corrupted one of the participants.

In each executed 2PC functionality, $\mathcal{C}$ and $\mathcal{V}$ can control their inputs to each function, and produce a value that is used in subsequent stages of the TLS protocol. By using maliciously-secure 2PC garbled circuit protocols, we reduce the ability for either party to cheat down to breaking any of the individual primitives executed within the garbled circuits. Fortunately, each of these primitives is already proven secure individually, and in a non-2PC TLS setting [10]. In other words, using these primitives does not permit any additional capabilities to an adversary that corrupts either party.

Therefore, criteria (1)-(4) are explained in the following. As noted in [6], $\mathcal{V}$ must reveal certain values (SHTS^v and CHTS^v) to allow $\mathcal{C}$ to decrypt handshake traffic before the application session keys are derived. As was shown in [10], revealing this information after committing to server-encrypted ciphertexts is safe, since the eventual application traffic secrets are independent of the handshake-encryption traffic keys. This protects against a malicious client, but means that any adversary that corrupts $\mathcal{C}$ learns all of the intermediate secrets that are used for encrypting and decrypting traffic *during* the handshake. On the other hand, a malicious verifier sending incorrect values will immediately be discovered since $\mathcal{C}$ will no longer be able to decrypt any traffic.

We can now finalise the security of the handshake into the following theorem.

Theorem 11 (Security of handshake phase)**.** *The DiStefano protocol is secure with respect to the ideal handshake phase functionality* (DCTLS.HSP)*, when assuming the following:*

- *a maliciously-secure 2PC-*ECtF *protocol;*
- *a maliciously-secure 2PC-*DeriveTKHS *protocol;*
- *a maliciously-secure 2PC-*DeriveTKApp *protocol;*
- *the hardness of the Shifted PRF ODH problem [6, Definition 2];*
- *the underlying security of the TLS 1.3 protocol [10].*

The proof of this theorem, follows a standard hybrid argument, where at each stage the 2PC protocol is replaced with an ideal functionality that computes the same result. Since each 2PC protocol is executed in sequence, this proof argument follows in the standard model. Once the ideal functionality is used, the rest of the security proof follows from the same properties that guarantee security of the underlying TLS 1.3 handshake protocol. A very similar security proof was given in [6] in the universal composability framework.

As a consequence, the security of DiStefano is confirmed, based on the choices of 2PC protocols that are used. The MPC primitives that we use and implement satisfy malicious security, and are discussed formally in Section 3.1 and Section 4. Our experimental results in Section 8 detail how performance changes depending on the choice of 2PC primitives.

6.2. Query Phase Security

The query phase of DiStefano essentially amounts to considering a 2PC realisation of the record-layer of the TLS 1.3 protocol. In other words, while the server is untouched, we continue to consider the client and the verifier, who work together to encrypt and decrypt packets to and from the server. This is a requirement, since

[14] See Fig. 4 for the full TLS derivation of each value.

Algorithm 6 2PC-RL-Encrypt ideal functionality

```
Require: $(\mathsf{tk}^{c}_{capp}, \mathsf{q}, AD)$ from $\mathcal{C}$
Require: $(\mathsf{tk}^{v}_{capp}, AD)$ from $\mathcal{V}$
  $(\hat{\mathsf{q}}, \tau_{\hat{\mathsf{q}}}) \leftarrow \mathsf{AEAD}.\mathsf{Enc}(\mathsf{tk}_{capp}, \mathsf{q}; AD)$
  return Output $(\hat{\mathsf{q}}, \tau_{\hat{\mathsf{q}}})$ to $\mathcal{C}$
  return Output $\hat{\mathsf{q}}$ to $\mathcal{V}$
```

Algorithm 7 2PC-RL-Decrypt ideal functionality

```
Require: $(\mathsf{tk}^{c}_{sapp}, (\hat{\mathsf{r}}, \tau_{\hat{\mathsf{r}}}), AD)$ from $\mathcal{C}$
Require: $(\mathsf{tk}^{v}_{sapp})$ from $\mathcal{V}$
  return $\mathsf{AEAD}.\mathsf{Dec}(\mathsf{tk}_{sapp}, \hat{\mathsf{r}}, \tau_{\hat{\mathsf{r}}}; AD)$ to $\mathcal{C}$
```

the end of the handshake phase of a DCTLS protocol leaves the client and verifier with shares of the secret session parameters, that need to be combined in order to construct messages.

In effect, a DCTLS protocol must consider two ideal functionalities: 2PC-RL-Encrypt (Algorithm 6), and 2PC-RL-Decrypt (Algorithm 7). In 2PC-RL-Encrypt, the client and the verifier submit their secret parameters, and the client submits a query (e.g. an HTTP request) to submit. The ideal functionality returns an encryption of this query, under a TLS 1.3-compliant AEAD scheme (Section 3.4). In 2PC-RL-Decrypt, the client and the verifier submit the same inputs, and the client submits a ciphertext received from the server, and the ideal functionality returns the decryption of this ciphertext under the same AEAD scheme, or $\perp$ in the event that the ciphertext does not decrypt properly.

We can show that the query phase of DiStefano is secure when AEAD = AES-GCM, assuming the security of the 2PC-AES-GCM protocol (Section 5.1). The proof that the query phase of DiStefano satisfies security with respect to the ideal DCTLS.QP functionality follows once we have protocols that are secure with respect to 2PC-RL-Encrypt and 2PC-RL-Decrypt. The proof that 2PC-AES-GCM satisfies both follows almost immediately from Lemma 9 and Lemma 10, due to the similarity between the ideal functionality for 2PC-RL-Encrypt (2PC-RL-Decrypt) and 2PC-AES-GCM Encrypt (2PC-AES-GCM Decrypt). We state the full theorem below for completeness.

Theorem 12. *The DiStefano protocol is secure with respect to the ideal query phase functionality* (DCTLS.QP), *when assuming a maliciously-secure 2PC-AES-GCM protocol, and the underlying security of the TLS 1.3 protocol [10].*

6.3. Commitment Phase Security

For the commitment phase of DiStefano, we split the requirement into a number of sub-properties: session privacy (SPriv); 1-out-of-n session authentication (SAuth^{1}_{n}); and session unforgeability (SUnf). In each model, we first assume that secure handshake and query phases have been computed, using the ideal functionalities (DCTLS.HSP, DCTLS.QP) (Appendix A). Recall that we only consider adversarial corruption of a single party in any situation, and therefore for any post-handshake security game, we consider only handshake phase corruptions concerning the same party.

In each of the security models (Fig. 8), we consider a (potentially dishonest) $\mathcal{C}$ that starts by sending a commitment, $\mathsf{c}_{\hat{\mathsf{q}},\hat{\mathsf{r}}}$, to a specific session, S. In SPriv, the honest client $\mathcal{C}$ constructs and sends a proof, $\sigma_{\mathcal{S},\mathsf{sid},\mathsf{L}}$, that $\mathsf{c}_{\hat{\mathsf{q}},\hat{\mathsf{r}}}$ is a commitment to a TLS session established with $\mathcal{S} \in \mathsf{L}$ (L is the set of accepted servers). The adversarial verifier, $\mathcal{A}_{\mathcal{V}}$, succeeds if it identifies the identity of $\mathcal{S}$ (it can point which server in the set L that $\mathcal{C}$ is communicating with). In SAuth^{1}_{n}, we consider an adversarial client, $\mathcal{A}_{\mathcal{C}}$, where the communication in the security game is the same, except that $\mathcal{A}_{\mathcal{C}}$ succeeds if the commitment $\mathsf{c}_{\hat{\mathsf{q}},\hat{\mathsf{r}}}$ corresponds to a session S established with a server $\mathcal{S}' \notin \mathsf{L}$. Finally, in SUnf, $\mathcal{V}$ reveals their secret session data to $\mathcal{A}_{\mathcal{C}}$, and $\mathcal{A}_{\mathcal{C}}$ succeeds if it can open $\mathsf{c}_{\hat{\mathsf{q}},\hat{\mathsf{r}}}$ to a different session S'. Overall, we show that each of the properties follows, assuming a sufficiently binding and hiding commitment scheme, and a ring signature scheme for ECDSA TLS certificates for implementing the proof that $\mathcal{S} \in \mathsf{L}$ (e.g. ZKAttest [46]).

Session privacy. For protecting the privacy of sessions during the commitment phase, i.e. that the client commitment does not reveal any information about the session to a malicious $\mathcal{V}$, we show that DiStefano satisfies security in the SPriv security game (Fig. 8).

Lemma 13. *Let Γ be a computationally hiding commitment scheme for a* DCTLS *scheme, and let Π be a ring signature scheme that satisfies anonymity for ECDSA TLS certificates. Then, for all PPT algorithms $\mathcal{A}$, we have that:*

$$\mathsf{Adv}^{\mathsf{spriv}}_{\mathcal{A},\mathsf{DCTLS},\Gamma}(\lambda) < \mathsf{negl}(\lambda).$$

Proof. We construct our proof of security as a two-step hybrid proof. In the first step, Π is modified to always sign using the secret key of server $\mathcal{S}^{0}$, regardless of the bit d. In the second step, the commitment scheme is modified to always commit to traffic exchanged with $\mathcal{S}^{0}$, regardless of the choice of bit d. We can see that steps above can be arbitrarily changed to always commit to traffic exchanged with $\mathcal{S}^{1}$, therefore, we will speak only about the $\mathcal{S}^{0}$, without loss of generality.

First, note that once both hybrid steps have been executed, the adversary $\mathcal{A}_{\mathcal{V}}$ has no advantage in guessing the bit d, since they always receive session commitments and ring signatures with respect to the traffic received from a single server. Therefore, we simply have to show that the real execution of SPriv is indistinguishable from this case to show that DCTLS satisfies SPriv security.

The distinguishing probability between the two views in the first hybrid step can be bounded by the anonymity property of Π. In other words, if there is an adversary $\mathcal{A}$

```
SPriv
  $\mathsf{L} \leftarrow \mathcal{A}_{\mathcal{V}}(1^{\lambda}, 1^{N})$
  $(\mathsf{sk}_{\Pi}, \mathsf{vk}_{\Pi}) \leftarrow \Pi.\mathsf{setup}(1^{\lambda})$
  $(\mathsf{pp}^0, \mathsf{sp}^0_{\mathcal{C}}, \mathsf{sp}^0_{\mathcal{S}}, \mathsf{sp}^0_{\mathcal{V}}) \leftarrow \mathsf{DCTLS.HSP}(1^{\lambda}, \mathcal{C}, \mathcal{S}^0, \mathcal{A}_{\mathcal{V}})$
  $(\mathsf{pp}^1, \mathsf{sp}^1_{\mathcal{C}}, \mathsf{sp}^1_{\mathcal{S}}, \mathsf{sp}^1_{\mathcal{V}}) \leftarrow \mathsf{DCTLS.HSP}(1^{\lambda}, \mathcal{C}, \mathcal{S}^1, \mathcal{A}_{\mathcal{V}})$
  if $[(\mathcal{S}^0 \notin \mathsf{L}) \vee (\mathcal{S}^1 \notin \mathsf{L})]$: return $0$
  $d \leftarrow\!\!\$\ \{0,1\}$
  $\mathsf{q} \leftarrow \mathcal{A}_{\mathcal{V}}(\mathsf{L})$
  $(\hat{\mathsf{q}}, \hat{\mathsf{r}}) \leftarrow \mathsf{DCTLS.QP}(\mathsf{pp}^d, \mathsf{sp}^d_{\mathcal{C}}, \mathsf{sp}^d_{\mathcal{S}}, \mathsf{sp}^d_{\mathcal{V}}, \mathsf{q})$
  $\sigma \leftarrow \Pi.\mathsf{Sign}(\mathsf{sk}_{\Pi}, \mathsf{pp}^d, \mathsf{sp}^d_{\mathcal{C}}, \mathsf{L})$
  $\mathsf{c}^d \leftarrow \Gamma.\mathsf{Commit}(\mathsf{sp}^d_{\mathcal{C}}, \hat{\mathsf{q}}, \hat{\mathsf{r}})$
  $d' \leftarrow \mathcal{A}_{\mathcal{V}}(\{\mathsf{pp}^d, \mathsf{sp}^d_{\mathcal{V}}\}_{d \in \{0,1\}}, \mathsf{vk}_{\Pi}, \mathsf{q}, \mathsf{c}^d, \sigma, \mathsf{L})$
  if $[d' \overset{?}{=} d]$: return $1$
  return $0$

$\mathsf{SAuth}^1_n$
  $\mathsf{L} \leftarrow \mathcal{V}(1^{\lambda}, 1^{N})$
  $(\mathsf{sk}_{\Pi}, \mathsf{vk}_{\Pi}) \leftarrow \Pi.\mathsf{setup}(1^{\lambda})$
  $(\mathsf{pp}, \mathsf{sp}_{\mathcal{C}}, \perp, \mathsf{sp}_{\mathcal{V}}) \leftarrow \mathsf{DCTLS.HSP}(1^{\lambda}, \mathcal{A}_{\mathcal{C}}, \mathcal{S}, \mathcal{V})$
  if $[\mathcal{S} \in \mathsf{L}]$: return $0$
  $\sigma \leftarrow \mathcal{A}_{\mathcal{C}}(\mathsf{pp}, \mathsf{sp}_{\mathcal{C}}, \mathsf{sk}_{\Pi}, \mathsf{vk}_{\Pi}, \mathsf{L})$
  return $\Pi.\mathsf{Verify}(\mathsf{vk}_{\Pi}, \mathsf{pp}, \mathsf{sp}_{\mathcal{V}}, \sigma, \mathsf{L})$

SUnf
  $(\mathsf{pp}, \mathsf{sp}_{\mathcal{C}}, \mathsf{sp}_{\mathcal{S}}, \mathsf{sp}_{\mathcal{V}}) \leftarrow \mathsf{DCTLS.HSP}(1^{\lambda}, \mathcal{A}_{\mathcal{C}}, \mathcal{S}, \mathcal{V})$
  $\mathsf{q} \leftarrow \mathcal{A}_{\mathcal{C}}(1^{\lambda}, \mathsf{pp}, \mathsf{sp}_{\mathcal{C}})$
  $(\hat{\mathsf{q}}, \hat{\mathsf{r}}) \leftarrow \mathsf{DCTLS.QP}(\mathsf{pp}, \mathsf{sp}_{\mathcal{C}}, \mathsf{sp}_{\mathcal{S}}, \mathsf{sp}_{\mathcal{V}}, \mathsf{q})$
  $\mathsf{c} \leftarrow \mathcal{A}_{\mathcal{C}}(\mathsf{sp}_{\mathcal{C}}, \hat{\mathsf{q}}, \hat{\mathsf{r}})$
  $\mathsf{q}' \leftarrow \mathcal{A}_{\mathcal{C}}(\mathsf{sp}_{\mathcal{C}}, \mathsf{sp}_{\mathcal{V}}, \mathsf{q}, \hat{\mathsf{q}}, \hat{\mathsf{r}}, \mathsf{c})$
  if $[(\Gamma.\mathsf{Open}(\mathsf{sp}_{\mathcal{V}}, \mathsf{q}', \mathsf{c})) \wedge (\mathsf{q}' \neq \mathsf{q})]$: return $1$
  return $0$
```

Figure 8. Security games for the commitment protocol.

that distinguish between the two steps, then there is an adversary $\mathcal{B}$ that can break the Anon security game of Π (Fig. 3). This follows since, in the case when $d = 1$ the only difference is the fact that σ is always computed over the certificate of $\mathcal{S}^0$. Therefore, $\mathcal{B}$ can simply forward the message to be signed during the TLS execution to their challenger, and receive back the signature σ. Then, they can send this signature back to $\mathcal{A}$ and output whatever $\mathcal{A}$ outputs. If $\mathcal{A}$ has non-negligible advantage in distinguishing between the two steps, then so will $\mathcal{B}$.

The distinguishing probability between the two views in the second hybrid step can be bounded by the fact that the session commitment is generated only for $\mathcal{S}^0$. As such, any adversary $\mathcal{B}$ against the computational hiding property of Γ forward their challenge commitment to $\mathcal{A}$ in the same as before, and win with the same advantage as $\mathcal{A}$. This completes the proof. □

1-out-of-n server authentication. We show that DiStefano ensures that a malicious $\mathcal{C}$ must authenticate $\mathcal{S}$ to $\mathcal{V}$, out of a set L possible n accepted servers (where L is specified by $\mathcal{V}$) using the SAuth^1_n security game (Fig. 8).

Lemma 14. *Let Π be a ring signature scheme that satisfies unforgeability for ECDSA TLS certificates. Then, for all PPT algorithms $\mathcal{A}$, we have that:*

$$\mathsf{Adv}^{\mathsf{sauth}^1_n}_{\mathcal{A},\mathsf{DCTLS},\Pi}(\lambda) < \mathsf{negl}(\lambda).$$

Proof. Suppose that $\mathcal{C}$ could win the SAuth^1_n game with non-negligible advantage. Then, an adversary $\mathcal{B}$ that is attempting to forge ring signatures for Π can simply output whatever signature $\mathcal{A}_{\mathcal{C}}$ outputs as their answer to the Unf security game (Fig. 3). If $\mathcal{A}_{\mathcal{C}}$ creates a valid forgery, then the unforgeability of Π is violated. □

Session unforgeability. We show that a malicious $\mathcal{C}$ cannot open commitments to sessions that were not previously committed to, by showing that DiStefano satisfies security in the SUnf security game (Fig. 8).

Lemma 15. *Let Γ be a perfectly binding commitment scheme for a* DCTLS *scheme. Then, for all PPT algorithms $\mathcal{A}$, we have that:*

$$\mathsf{Adv}^{\mathsf{spriv}}_{\mathcal{A},\mathsf{DCTLS},\Gamma}(\lambda) < \mathsf{negl}(\lambda).$$

Proof. It is clear to see that an adversary attempting to break the perfect binding property of Γ can utilise the adversary $\mathcal{A}_{\mathcal{V}}$ against SUnf to establish a valid opening based on an uncommitted value. □

7. Implementation

In order to enable easy integration with other cryptographic libraries and browsers, we implemented a prototype of DiStefano in C++.[15] This implementation contains around 14K lines of code, tests and documentation. We developed this implementation using C++ best practices, and we hope that this effort is useful for other researchers who wish to extend our work.

Concretely, our implementation of DiStefano uses BoringSSL for TLS functionality and emp for all MPC functionality. BoringSSL is the only cryptographic library supported by Chromium-based Internet browsers, that make up a dominant share of all Internet browser usage. As far as we are aware, our implementation contains primitives and circuits that are not available elsewhere. Our implementation also contains a modified version of N-for-1-Auth's circuit generation to produce the relevant garbled circuits. We further reduce the online cost of N-for-1-Auth's secret sharing scheme by

[15] Code: https://anonymous.4open.science/r/tls-at-6823

using a pre-determined splitting scheme for specific secrets: we refer the reader to our implementation[16].

Our implementation of DiStefano's DCTLS protocol is fully integrated inside BoringSSL's TLS 1.3 handshake. In order to achieve this integration without heavily modifying BoringSSL, we augmented the library's internal SSL data structures with a series of function pointers. These changes comprise of around 150 lines of code. As these changes are small, our implementation of DiStefano can be easily updated to more recent versions of BoringSSL. Additionally, using function pointers makes all functionality *generic*, allowing for further modifications. In practice, the overhead of using function pointers should be small compared to the needed MPC functionality.

On the other hand, our changes to emp were far simpler. Specifically, in order to better model constrained devices, we removed the multi-threaded pre-processing from emp's authenticated garbling scheme. This decision increases the pre-processing time of our circuits by around a factor of 10 without increasing the online time: we discuss this in more detail in Section 8. For completeness, we provide a full listing of the changes made to third party libraries alongside our prototype.

8. Experimental Analysis

We evaluated the performance of DiStefano in a LAN setting. To better reflect a real-world environment, we use a consumer-grade device (a Macbook air M1 with 8GB of RAM) for $\mathcal{C}$ and a server-grade device (an Intel Xeon Gold 6138 with 32GB of RAM) for $\mathcal{V}$ with all communication between these nodes using TLS 1.3. All evaluations were carried out using a single thread over a 1Gbps network with a round-trip time of around 16ms. Timings and bandwidth measurements are computed as the mean of 50 samples, and are represented in milliseconds and mebibytes respectively (1 MiB is 2^{20} bytes). Each measurement is given to 4 significant figures.

Before presenting our results, we will like to point out that comparisons between our results and others in the literature [2], [5] should be made carefully. In the case of DECO, the implementation used is not publicly available, and, as a result, we were unable to reproduce any of their results on our hardware. Moreover, as our implementation is single-threaded, we are unable to take advantage of the multi-threaded pre-processing provided by emp. Given that [34, §7] reports an order of magnitude increase in bandwidth due to multi-threading, it is perhaps not surprising that our offline times are an order of magnitude higher than those presented in DECO. However, our online timings are comparable with DECO, and parallelising the pre-processing stage would likely mitigate any discrepancies[17]. In any case, as the pre-processing can be carried out before DCTLS, we do not consider this increased time as a major issue.

On the other hand, it is also difficult to compare our timings to PageSigner. The original implementation of it is written entirely in Javascript, preventing the usage of dedicated hardware resources. Given that our implementation is instead written in a natively compiled language, we might reasonably expect DiStefano to be faster than PageSigner. Moreover, PageSigner also follows a semi-honest security model and solely targets TLS 1.2: both of these are incompatible with DiStefano.

To make the results in this section easier to interpret, we evaluate the runtimes and bandwidth costs of each piece of DiStefano in isolation. We stress that evaluating and optimising the individual primitives used in this work is rather delicate, and better results may be achieved via small tweaks to our implementation. We highlight these decisions where appropriate, and discuss circumstances in which these changes might be useful.

Table 3 gives results for each individual circuit used in DiStefano. Each circuit is evaluated without amortisations, i.e. these timings do not take advantage of the amortised pre-processing available inside emp. Indeed, as the most expensive operation of these circuits will only be used once per session, we do not expect that employing amortisation will yield a substantial speed-up. However, employing amortisations for common operations, e.g. AES-GCM tagging and verification, may lead to faster running times than presented here (see [34, §7] for concrete speed-ups). We also compare the offline time using the original implementation of authenticated garbling (LeakyDeltaOT [34]) that utilises FerretCOT. Our results imply that FerretCOT performs better than the original OT for large circuit sizes in both bandwidth and running time. However, for smaller circuits it appears that the original implementation is faster at the cost of requiring more bandwidth. Given that the pre-processing times are proportional to the size of the circuits, we can see that our results appear to be predominantly network bound. The results also highlight that our Karatsuba-based circuit achieves modest gains in both bandwidth and time over the naive circuit.

Table 4 shows the results for each arithmetic primitive used in DiStefano. Given that all running times and bandwidth counts are rather low, we do not expect this portion of DiStefano to represent a bottleneck even on constrained networks. We can also see that the tweak introduced in Section 5.2 reduces the running time by a factor of around 3, whilst also halving the required bandwidth for the multiplication (this ignores bandwidth used by shared setup). This also represents an improvement of around 4 orders of magnitude over using a garbled circuit.

[16] We stress that this approach is less flexible than N-for-1-Auth's approach. For example, our approach only supports 2 parties, whereas N-for-1-Auth supports arbitrarily many.

[17] Notably, [2] does not mention if the pre-processing used multiple threads or not.

Table 3. Garbled Circuit timings and bandwidth.

Circuit	OT protocol	Offline (ms)	Online (ms)	Bandwidth (MB)
AES-GCM share (K)	LeakyDeltaOT	2340	34.92	21.04
AES-GCM share (K)	FerretCOT	2683	59.48	9.009
AES-GCM share (N)	LeakyDeltaOT	2678	39.09	25.63
AES-GCM share (N)	FerretCOT	2853	61.36	10.35
AES-GCM Tag	LeakyDeltaOT	1019	22.30	7.604
AES-GCM Tag	FerretCOT	2336	22.22	5.010
AES-GCM Verify	LeakyDeltaOT	1032	21.16	7.746
AES-GCM Verify	FerretCOT	2277	21.24	5.130
TLS 1.3 HS (P256)	LeakyDeltaOT	51470	93.16	558.7
TLS 1.3 HS (P256)	FerretCOT	19847	88.90	173.4
TLS 1.3 HS (P384)	LeakyDeltaOT	51610	95.38	560.1
TLS 1.3 HS (P384)	FerretCOT	19940	89.88	173.8
TLS 1.3 TS	LeakyDeltaOT	51450	95.21	523.2
TLS 1.3 TS	FerretCOT	18820	99.25	162.6
AES Commit	LeakyDeltaOT	1070	15.67	7.583
AES Commit	FerretCOT	2303	15.86	5.084
2PC-GCM (256B)	LeakyDeltaOT	11120	39.54	117.6
2PC-GCM (256B)	FerretCOT	5495	39.42	37.17
2PC-GCM (512B)	LeakyDeltaOT	21790	59.02	234.9
2PC-GCM (512B)	FerretCOT	8752	58.82	71.4
2PC-GCM (1KB)	LeakyDeltaOT	34180	97.93	367.7
2PC-GCM (1KB)	FerretCOT	12950	97.18	114.53
2PC-GCM (2KB)	LeakyDeltaOT	67750	176.57	734.9
2PC-GCM (2KB)	FerretCOT	25200	170	226.8

Table 4. Primitive timings and bandwidth.

Primitive	Time (ms)	Bandwidth (MB)
ECtF (P256)	286.3	0.384
ECtF (P384)	335.5	0.648
ECtF (P521)	421.4	1.22
MtA (P256)	33.67	0.075
MtA (P384)	40.65	0.127
MtA (P521)	55.83	0.241
AES-GCM powers (multiplicative)	1694	0.049
AES-GCM powers (additive)	5926	0.080
AES-GCM powers (GC)	—	900

Preliminary timings indicate that our implementation of DiStefano is competitive with the timings reported in DECO, with the DCTLS portion taking around 500ms to complete for a 256-bit secret. We intend to provide timings for a WAN setting in the next version of this report, along with a slightly updated implementation.

9. Discussion

In this section, we cover some of the related work in constructing DCTLS-like protocols, some applications of using commitments based on TLS.encrypted traffic, and limitations of both the DiStefano design and of DCTLS protocols more generally. We finish by discussing possible browser-based integrations of DiStefano.

9.1. Related Work

As noted in Section 2, DiStefano is an instance of a DCTLS protocol. Other alternatives exist, but all have limitations as noted in Section 2.2. The DECO and PageSigner protocols, for example, only (formally) work for TLS 1.2 and under, and provide limited privacy. TownCrier [18] has similar problems, and requires using trusted computing functionality. Recently, the PECO protocol [56] was proposed, which informally extends the DECO protocol to support TLS 1.3, but provides no formal guarantees nor implementation of it.

The *N-for-1-Auth* protocol [57] allows a user to authenticate to N servers independently by doing the work of only authenticating to one. An N-for-1 authentication system consists of many servers and users. Each user has a number of authentication factors they can use to authenticate. The user holds a secret s that they wish to distribute among the N servers. The protocol consists of two phases: i. *Enrollment*, when the user wants to store s on the servers, the user provides the servers with a number of authentication factors, which the servers verify using authentication protocols: these protocols use a mechanism called “TLS-in-SMPC” that allows N servers to jointly act as a TLS client endpoint to communicate with a TLS server (which can be, for example, a TLS email server. A single server from the N ones cannot decrypt any TLS traffic), and, after authenticating with these factors, the client secret-shares s and distributes the shares across the servers; ii. *Authentication*, where the user runs the N-for-1-Auth protocols for the authentication factors and, once it is authenticated, the N servers can perform computation over s for the user, which is application-specific (it can be key-recovery, for example).

The *Oblivious TLS* protocol [6] allows for any TLS endpoint to obviously interact with another TLS endpoint, without the knowledge that it is interacting with a multi-party computation instance. It consists of the following phases: i. *Multi-Party Key Exchange*, which is the key exchange phase of the TLS handshake ran in an MPC manner by performing an exponentiation between a known public key and a secret exponent, where the output remains secret; ii. *Threshold Signing*, which is the authentication phase of the TLS handshake done by having the TLS transcript signed with EdDSA Schnorr-based signatures in a threshold protocol; iii. *Record Layer* which is ran by using authenticated encryption, based on AES-GCM, inside MPC.

Recent work on developing zero-knowledge middleboxes for TLS 1.3 traffic [58] has many similarities with techniques used in DCTLS protocols. However, in their setting, they consider the third-party to be an on-path proxy that receives encrypted traffic (similar to the proxy model of DECO [2]). Furthermore, the client is only required to produce commitments to their own traffic, rather than the traffic that is received from the server. The considered application involves examples such as those that require corporate governance of Internet browsing to be enforced by middleboxes, which would naturally be thwarted in a setting where all client traffic is encrypted.

Table 5. Comparison of functionality provided by DCTLS-like protocols.

Protocol	TLS 1.3	Attestations	1-out-of-n auth
DECO/TownCrier [2], [18]	✗	✓	✗
N-for-1 [7]	✓	✗	✗
Oblivious TLS [6]	✓	✗	✗
ZKMiddleboxes [58]	✓	$\mathcal{C} \rightarrow \mathcal{S}$ only	✗
DiStefano	✓	✓	✓

Table 6. Results for running KeyUpdate in 2PC.

Circuit	OT protocol	Offline time (ms)	Online time (ms)	Bandwidth (MB)
KeyUpdate	LeakyDeltaOT	10540	29.95	98.54
KeyUpdate	FerretCOT	7960	31.96	31.61

Table 5 compares the functionality provided in DiStefano with other DCTLS-like protocols.

Concurrent work. In work concurrent to ours, Xie et al. [59] proposed a series of optimisations to the MPC protocols used inside DECO, primarily targeting the TLS 1.2 setting. Whilst most of these improvements are orthogonal to our work, one particularly interesting optimisation is a faster approach for deriving TLS traffic secrets inside garbled circuits. Interestingly, this approach is somewhat reminiscent of the highly optimised CBC-HMAC protocol proposed in DECO [2, §4.2.1] for computing tags in 2PC. Whilst we have not yet incorporated this particular improvement into our work, we believe that adapting our circuits to take advantage of this change is likely to be straightforward. We intend to incorporate these changes and benchmark the improved timings in the next version of this work.

9.2. Applications

DiStefano can be used to commit to encrypted TLS 1.3 data. As noted in [2], such commitments can be used as the basis of zero-knowledge proofs (or *attestations*) for showing that certain facts are present in such traffic. However, note that such attestations could also be constructed via different methods, using cooperative decryption of certain ciphertext blocks, or more generic 2PC techniques. The DECO protocol provides examples of applications that they can prove certain statements for, including proof of confidential financial information, and proof of age. We note further that TLS sessions could be as the basis for more generic user credentials, that prove arbitrary facts about a user. For a more complete summary, see [2].

9.3. Limitations

We conclude this section by addressing both the implementation and real-world limitations of our work. In the first case, our implementation of DiStefano does not support key rotation via KeyUpdate messages or full 0-RTT mode. In practice, these limitations are not major: in both cases these issues can be circumvented by simply re-running the HSP [18]. We also provide no concrete instantiation of the zero-knowlege primitives that can be used to create attestations, but they should follow the guidelines stated in Section 4.5. Note that said proofs should only attest of the relevant information needed and should not harm user's privacy.

DCTLS protocols must assume a particular characteristic of its users for proofs to be *meaningful*: it requires that they are honest in the data that they transmit. Suppose that Alice wishes to provide proof of their age to a particular website. Alice logs in a Government agency website and then runs DiStefano to produce a proof of age. However, this process assumes that the account Alice logs into is theirs, which may not be the case e.g. Alice may have used a stolen account for attestation or somehow created a fake account. In this setting, there are no guarantees given to the third-party website unless it assumes that Alice is honest.

There could also be the possibility that DCTLS could be used in coercive situations by becoming actively harmful to vulnerable or at risk people due to the lack of human involvement. Thus, we would like to emphasise that deployment of tools such as DiStefano must be considered carefully. Furthermore, DCTLS can also be subject of different legal and compliance issues in regards to being considered as a form of webscraping. The DECO paper goes on detail about cases as such.

9.4. Browser Integration

DiStefano can be integrated into a browser that uses BoringSSL, e.g. Google Chrome/Brave, easily. As our changes to BoringSSL itself are rather minimal, it would be possible to simply describe our changes as a series of deltas in a version control system. These deltas can then be applied during the process of building the browser based on build flags.[19] We leave the completion and deployment of such library as future work.

10. Conclusion

We build DiStefano, a DCTLS protocol that allows for generation of private commitments to encrypted TLS 1.3 data. We use a modular security framework that preserves TLS 1.3 security properties, and guarantees 1-out-of-n privacy for client browsing patterns. We provide an open-source integration into the BoringSSL library, that demonstrates the efficiency of DiStefano.[20] The flexibility, security, and usability of DiStefano makes it an immediate candidate for many real-world applications.

[18] For completeness, we benchmarked the cost of running the KeyUpdate operation in a garbled circuit, see Table 6.

[19] Indeed, such a system is already used for the Brave Browser.

[20] https://anonymous.4open.science/r/tls-at-6823

Acknowledgements

We would like to thank Xiao Wang for his help with the various aspects of EMP and for sharing an initial integration of Ferret; Yashvanth Kondi and Peter Scholl for useful conversations; Benjamin Livshits for discussing interesting applications for this work. Large parts of this work were completed while all of the authors were affiliated with Brave Software. Joe Rowell was supported by EPSRC grant EP/P009301/1.

References

[1] E. Rescorla, "The transport layer security (tls) protocol version 1.3," Internet Requests for Comments, RFC Editor, RFC 8446, August 2018.

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

[3] M. Rosenberg, J. White, C. Garman, and I. Miers, "`zk-creds`: Flexible anonymous credentials from zkSNARKs and existing identity infrastructure," Cryptology ePrint Archive, Report 2022/878, 2022, https://eprint.iacr.org/2022/878.

[4] A. Guy, M. Sporny, D. Reed, and M. Sabadello, "Decentralized identifiers (DIDs) v1.0," W3C, W3C Recommendation, Jul. 2022, https://www.w3.org/TR/2022/REC-did-core-20220719/.

[5] P. team, "Pagesigner: One-click website auditing," Website, accessed 04/04/2023. [Online]. Available: https://old.tlsnotary.org/pagesigner

[6] D. Abram, I. Damgård, P. Scholl, and S. Trieflinger, "Oblivious TLS via multi-party computation," in *CT-RSA 2021*, ser. LNCS, K. G. Paterson, Ed., vol. 12704. Springer, Heidelberg, May 2021, pp. 51–74.

[7] W. Chen, R. Deng, and R. A. Popa, "N-for-1 auth: N-wise decentralized authentication via one authentication," Cryptology ePrint Archive, Report 2021/342, 2021, https://eprint.iacr.org/2021/342.

[8] Cloudflare, "Tls 1.2 vs. tls 1.3 vs. quic: Distribution of secure traffic by protocol," 2023, accessed 11/04/2023. [Online]. Available: https://radar.cloudflare.com/adoption-and-usage#tls-1-2-vs-tls-1-3-vs-quic

[9] H. Lee, D. Kim, and Y. Kwon, "Tls 1.3 in practice:how tls 1.3 contributes to the internet," in *Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 70–79. [Online]. Available: https://doi.org/10.1145/3442381.3450057

[10] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol," *Journal of Cryptology*, vol. 34, no. 4, p. 37, Oct. 2021.

[11] I. Google, "Boringssl," https://boringssl.googlesource.com/boringssl/. [Online]. Available: https://boringssl.googlesource.com/boringssl/

[12] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 5–17.

[13] K. Arai and S. Matsuo, "Formal verification of TLS 1.3 full handshake protocol using proverif (Draft-11)," IETF TLS mailing list, 2016. [Online]. Available: https://mailarchive.ietf.org/arch/msg/tls/NXGYUUXCD2b9WwBRWbvrccjjdyI

[14] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: Breaking TLS using SSLv2," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, August 2016, pp. 689–706. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram

[15] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 535–552.

[16] R. Holz, J. Hiller, J. Amann, A. Razaghpanah, T. Jost, N. Vallina-Rodriguez, and O. Hohlfeld, "Tracking the deployment of tls 1.3 on the web: A story of experimentation and centralization," *SIGCOMM Comput. Commun. Rev.*, vol. 50, no. 3, p. 3–15, jul 2020. [Online]. Available: https://doi.org/10.1145/3411740.3411742

[17] T. team, "Tlsnotary: Proof of data authenticity," Website, accessed 04/04/2023. [Online]. Available: https://tlsnotary.github.io/landing-page/

[18] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 270–282. [Online]. Available: https://doi.org/10.1145/2976749.2978326

[19] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

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

[21] A. Backman, J. Richer, and M. Sporny, "Signing http messages," IETF draft, accessed 14/11/2022. [Online]. Available: https://www.ietf.org/archive/id/draft-ietf-httpbis-message-signatures-04.html

[22] D. Tymokhanov and O. Shlomovits, "Alpha-rays: Key extraction attacks on threshold ecdsa implementations," Cryptology ePrint Archive, Paper 2021/1621, 2021, https://eprint.iacr.org/2021/1621. [Online]. Available: https://eprint.iacr.org/2021/1621

[23] N. Makriyannis and U. Peled, "A note on the security of gg18," 2021, https://info.fireblocks.com/hubfs/A_Note_on_the_Security_of_GG.pdf. [Online]. Available: https://info.fireblocks.com/hubfs/A_Note_on_the_Security_of_GG.pdf

[24] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, "Ferret: Fast extension for correlated OT with small communication," in *ACM CCS 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM Press, Nov. 2020, pp. 1607–1626.

[25] M. Rosulek and L. Roy, "Three halves make a whole? Beating the half-gates lower bound for garbled circuits," in *CRYPTO 2021, Part I*, ser. LNCS, T. Malkin and C. Peikert, Eds., vol. 12825. Virtual Event: Springer, Heidelberg, Aug. 2021, pp. 94–124.

[26] Y. Lindell and B. Pinkas, "Secure multiparty computation for privacy-preserving data mining," Cryptology ePrint Archive, Report 2008/197, 2008, https://eprint.iacr.org/2008/197.

[27] A. C. Yao, "Protocols for secure computations," in *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, 1982, pp. 160–164.

[28] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity and a methodology of cryptographic protocol design (extended abstract)," in *27th FOCS*. IEEE Computer Society Press, Oct. 1986, pp. 174–187.

[29] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *CRYPTO 2012*, ser. LNCS, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, Heidelberg, Aug. 2012, pp. 643–662.

[30] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *ACM CCS 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM Press, Nov. 2020, pp. 1575–1590.

[31] M. Keller, E. Orsini, and P. Scholl, "MASCOT: Faster malicious arithmetic secure computation with oblivious transfer," in *ACM CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM Press, Oct. 2016, pp. 830–842.

[32] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *CRYPTO'91*, ser. LNCS, J. Feigenbaum, Ed., vol. 576. Springer, Heidelberg, Aug. 1992, pp. 420–432.

[33] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *ICALP 2008, Part II*, ser. LNCS, L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, Eds., vol. 5126. Springer, Heidelberg, Jul. 2008, pp. 486–498.

[34] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 21–37.

[35] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," in *CRYPTO 2003*, ser. LNCS, D. Boneh, Ed., vol. 2729. Springer, Heidelberg, Aug. 2003, pp. 145–161.

[36] M. Keller, E. Orsini, and P. Scholl, "Actively secure OT extension with optimal overhead," in *CRYPTO 2015, Part I*, ser. LNCS, R. Gennaro and M. J. B. Robshaw, Eds., vol. 9215. Springer, Heidelberg, Aug. 2015, pp. 724–741.

[37] C. Guo, J. Katz, X. Wang, and Y. Yu, "Efficient and secure multiparty computation from fixed-key block ciphers," in *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2020, pp. 825–841.

[38] R. Gennaro and S. Goldfeder, "Fast multiparty threshold ECDSA with fast trustless setup," in *ACM CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM Press, Oct. 2018, pp. 1179–1194.

[39] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT'99*, ser. LNCS, J. Stern, Ed., vol. 1592. Springer, Heidelberg, May 1999, pp. 223–238.

[40] H. Xue, M. H. Au, X. Xie, T. H. Yuen, and H. Cui, "Efficient online-friendly two-party ECDSA signature," in *ACM CCS 2021*, G. Vigna and E. Shi, Eds. ACM Press, Nov. 2021, pp. 558–573.

[41] I. Haitner, N. Makriyannis, S. Ranellucci, and E. Tsfadia, "Highly efficient OT-based multiplication protocols," in *EUROCRYPT 2022, Part I*, ser. LNCS, O. Dunkelman and S. Dziembowski, Eds., vol. 13275. Springer, Heidelberg, May / Jun. 2022, pp. 180–209.

[42] J. Doerner, Y. Kondi, E. Lee, and a. shelat, "Secure two-party threshold ECDSA from ECDSA assumptions," in *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2018, pp. 980–997.

[43] ——, "Threshold ECDSA from ECDSA assumptions: The multiparty case," in *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019, pp. 1051–1066.

[44] R. Impagliazzo and M. Naor, "Efficient cryptographic schemes provably as secure as subset sum," *Journal of Cryptology*, vol. 9, no. 4, pp. 199–216, Sep. 1996.

[45] R. L. Rivest, A. Shamir, and Y. Tauman, "How to leak a secret," in *ASIACRYPT 2001*, ser. LNCS, C. Boyd, Ed., vol. 2248. Springer, Heidelberg, Dec. 2001, pp. 552–565.

[46] A. Faz-Hernández, W. Ladd, and D. Maram, "ZKAttest: Ring and group signatures for existing ECDSA keys," in *SAC 2021*, ser. LNCS, R. AlTawy and A. Hülsing, Eds., vol. 13203. Springer, Heidelberg, Sep. / Oct. 2022, pp. 68–83.

[47] M. Naor and M. Yung, "Public-key cryptosystems provably secure against chosen ciphertext attacks," in *22nd ACM STOC*. ACM Press, May 1990, pp. 427–437.

[48] T. Iwata, K. Ohashi, and K. Minematsu, "Breaking and repairing GCM security proofs," in *CRYPTO 2012*, ser. LNCS, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, Heidelberg, Aug. 2012, pp. 31–49.

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

[50] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol candidates," in *ACM CCS 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM Press, Oct. 2015, pp. 1197–1210.

[51] S. Gueron and M. E. Konavis, "Intel® carry-less multiplication instruction and its usage for computing the gcm mode," 2014, accessed 14/03/2023. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/clmul-wp-rev-2-02-2014-04-20.pdf

[52] M. Bellare and P. Rogaway, "Entity authentication and key distribution," in *CRYPTO'93*, ser. LNCS, D. R. Stinson, Ed., vol. 773. Springer, Heidelberg, Aug. 1994, pp. 232–249.

[53] M. Fischlin and F. Günther, "Multi-stage key exchange and the case of Google's QUIC protocol," in *ACM CCS 2014*, G.-J. Ahn, M. Yung, and N. Li, Eds. ACM Press, Nov. 2014, pp. 1193–1204.

[54] F. Günther, "Modeling advanced security aspects of key exchange and secure channel protocols," *it - Information Technology*, vol. 62, no. 5-6, pp. 287–293, 2020.

[55] J. Brendel, M. Fischlin, F. Günther, and C. Janson, "PRF-ODH: Relations, instantiations, and impossibility results," in *CRYPTO 2017, Part III*, ser. LNCS, J. Katz and H. Shacham, Eds., vol. 10403. Springer, Heidelberg, Aug. 2017, pp. 651–681.

[56] M. B. Santos, "Peco: methods to enhance the privacy of deco protocol," Cryptology ePrint Archive, Paper 2022/1774, 2022, https://eprint.iacr.org/2022/1774. [Online]. Available: https://eprint.iacr.org/2022/1774

[57] W. Chen, R. Deng, and R. A. Popa, "N-for-1 auth: N-wise decentralized authentication via one authentication," Cryptology ePrint Archive, Paper 2021/342, 2021, https://eprint.iacr.org/2021/342. [Online]. Available: https://eprint.iacr.org/2021/342

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

[59] X. Xie, K. Yang, X. Wang, and Y. Yu, “Lightweight authentication of web data via garble-then-prove,” Cryptology ePrint Archive, Paper 2023/964, 2023, https://eprint.iacr.org/2023/964. [Online]. Available: https://eprint.iacr.org/2023/964

[60] J. Groth and M. Kohlweiss, “One-out-of-many proofs: Or how to leak a secret and spend a coin,” Cryptology ePrint Archive, Report 2014/764, 2014, https://eprint.iacr.org/2014/764.

Appendix A. Formal description of DCTLS protocols

The three phases (HSP, QP, CP) of a generic three-party TLS (DCTLS) protocol can be modified as formal algorithmic processes (or ideal functionalities) executed between $\mathcal{C}$, $\mathcal{S}$, and $\mathcal{V}$. These processes are described below.

$(\mathsf{pp}, \mathsf{sp}_{\mathcal{C}}, \mathsf{sp}_{\mathcal{S}}, \mathsf{sp}_{\mathcal{V}}) \leftarrow \mathsf{DCTLS.HSP}(1^{\lambda})$: The handshake phase takes as input a security parameter, and computes a TLS handshake between $\mathcal{S}$, and an effective client that consists of both $\mathcal{C}$ and $\mathcal{V}$. The public/secret parameters $(\mathsf{pp}, \mathsf{sp}_{\mathcal{S}})$ learnt by $\mathcal{S}$ are the same as in a standard TLS handshake. The secret parameters learned by $\mathcal{C}$ $(\mathsf{sp}_{\mathcal{C}})$ and $\mathcal{V}$ $(\mathsf{sp}_{\mathcal{V}})$ are shares of the secret parameters learnt by a standard TLS client [10], so that neither party can compute encrypted traffic alone.

$(\mathsf{r}, \hat{\mathsf{q}}, \hat{\mathsf{r}}) \leftarrow \mathsf{DCTLS.QP}(\mathsf{pp}, \mathsf{sp}_{\mathcal{C}}, \mathsf{sp}_{\mathcal{S}}, \mathsf{sp}_{\mathcal{V}}, \mathsf{q})$: The query phase takes the public and secret parameters of each party as input, along with a query, q, that is to be sent to $\mathcal{S}$. This phase requires $\mathcal{S}$ to construct a response, r, to q and return it to $\mathcal{C}$. The phase outputs both q and r, and also vectors of TLS ciphertexts ($\hat{\mathsf{q}}$ and $\hat{\mathsf{r}}$) that encrypt the client queries and the server responses. $\hat{\mathsf{q}}$ and $\hat{\mathsf{r}}$ are vectors containing blocks of the TLS ciphertext encrypting q and r, respectively.

$b \leftarrow \mathsf{DCTLS.CP}(\mathsf{pp}, \mathsf{sp}_{\mathcal{C}}, \mathsf{sp}_{\mathcal{V}}, \mathsf{q}, \mathsf{r}, \hat{\mathsf{q}}, \hat{\mathsf{r}}, (i, j))$: The commitment phase outputs a bit b, where $b = 1$ if $\mathcal{C}$ constructs a valid opening of $\hat{\mathsf{q}}[i]$ and $\hat{\mathsf{r}}[j]$ with respect to the unencrypted q and r. Broadly speaking, $\mathcal{C}$ sends to $\mathcal{V}$ the TLS-encrypted ciphertexts, before $\mathcal{V}$ sends $\mathsf{sp}_{\mathcal{V}}$ to $\mathcal{C}$, and then $\mathcal{C}$ opens the commitments. Note that a valid opening could be proving in zero-knowledge that $\hat{\mathsf{r}}[j]$ encrypts a value in a given range, or using 2PC to decrypt the block directly.

Appendix B. Background on ZKAttest

With ZKAttest, $\mathcal{S}$ can create a proof of knowledge of a ECDSA Signature. We explain this by first recalling how ECDSA works. Let $s_k \in \mathbb{F}_q$ be a ECDSA signing key, m the message to sign, and let $P_k = s_k \cdot G$ be the corresponding public verification key. In order to sign a message m, the signer first samples a scalar value $k \in \mathbb{F}_q$ and uses this to produce a random point $(x_1, y_1) = kG$, with the restriction that $r = x_1 \mod n \neq 0$. The signer then truncates a hash of m to n bits (i.e. $z = H(m)\,[0 : n-1]$) and computes $s = \frac{z + r \cdot G}{k} \mod n$, outputting (r, s) as the signature on m. To verify this signature, the receiving party checks that all inputs are valid, before computing $(u_1 = zs^{-1} \mod n)$ and $(u_2 = rs^{-1} \mod n)$, and finally $R = (R_x, R_y) = (u_1 \cdot G) + (u_2 \cdot P_k)$, with validation passing if $R_x = r \mod n$ and failing otherwise.

The authors of ZKAttest noticed that the ECDSA verification functionality is unsuited to create zero-knowledge proofs, since it is unclear how to prove that $R_x = r \mod n$ without revealing R_x to $\mathcal{V}$ [46]. Revealing R_x can actually lead to a fairly straightforward de-anonymisation attack, as knowledge of R_x alongside knowledge of zs^{-1} allows recovery of P_k. Notice that there are only finitely many choices of R_y for a given R_x: in this case, knowledge of R_x alongside knowledge of z and s allows $\mathcal{V}$ to compute $(R - zs^{-1} \mod n) \cdot G = (rs^{-1} \mod n) \cdot P_k$, which allows recovery of P_k via modular inversion.

To circumvent this attack, the authors instead modify the verification equation, revealing R directly to the verifier and only revealing a commitment to $b = s/r$ instead of r and s directly. In this case, verifying an ECDSA signature is the same as proving that $bR - zr^{-1} \cdot G = P_k$. As z and r can be revealed without revealing s, and as R appears as a uniformly random value, this proof works in zero-knowledge.

Given a list of public keys that $\mathcal{S}$ holds $(P_{k1}, P_{k2}, \ldots, P_{kn})$ as a ring, $\mathcal{S}$ takes the private key corresponding to their key P_{ki}, commits to it, signs the TLS 1.3 transcript ($\text{Label}_7 \parallel \text{H}_3$), creates a proof of signature under committed key (by using the ZKAttest scalar multiplication proof and the point addition proof to prove $bR - (zr)^{-1} \cdot G = P_{ki}$), and then proves that the commitment is to one of the keys in the list via Groth-Kohlweiss proofs [60]. $\mathcal{V}$ gets the revealed H_3 and the proof (which states that there is a signature that verifies under a public key that is on the list of public keys that a $\mathcal{S}$ maintains), and it is able to verify such proof.

Appendix C. PageSigner and AES-GCM

In this section, we compare our approach to computing AES-GCM tags to the approach employed by PageSigner [5]. For a background on AES-GCM tags, see Section 5. In a 2PC setting, we assume that both k and the powers of $h = h_c + h_v$ are additively shared by both parties, with C, IV and A acting as public inputs.

Assuming that C is a single block without any associated data (i.e. $C = C_1$), we have $\tau = (h_c^{m-2} + h_v^{m-2}) \cdot (h_c^1 + h_v^1) \cdot C_1 = (h_c^{m-1} + h_v^{m-1} + h_c^{m-2} \cdot h_v^1 + h_v^{m-2} \cdot h_c^{m-2}) \cdot C_1$. As the first of these terms can be computed locally, the cost of computing τ can be reduced to computing $(h_v^{m-2} \cdot h_c + h_c^{m-2} \cdot h_v) \cdot C_1$ in 2PC. This approach can actually be written as a variant of our approach, as the left hand-side is fixed for a particular sharing of h. However, PageSigner instead repeats this process each time a tag is computed. Interestingly, it turns out that simply computing a sharing of $h_v^2 h_c$ and $h_c^2 h_v$ is sufficient to tag blocks of arbitrary length, lowering the cost of tagging to just two OT-based multiplications.

From a performance perspective, a "back-of-an-envelope" calculation shows that this approach is strictly less performant than the approach used by us. Intuitively, this is because our approach allows all polynomial evaluation to be done locally, which is not the case for PageSigner: whilst both approaches require computing an initial sharing of h and its powers, PageSigner's approach also requires computing two OT-based multiplications per tagging. Concretely, instantiating these multiplications using the maliciously secure scheme presented in [42] with 128-bits of statistical security would require 2048 oblivious transfers of 128-bits for the multiplication alone, requiring around 32KiB of bandwidth per tag. In contrast, our scheme only requires transferring around 64 bytes per tagging operation. In other words, our scheme requires around 500× less bandwidth per tagging operation than the approach employed by PageSigner.

Appendix D. Commitment scheme security

We prove that Γ is a perfectly binding and computationally hiding commitment scheme.

Lemma 16 (Perfectly binding). *The commitment scheme Γ is perfectly binding.*

Proof. Recall that the decryption key $r = r_i^k + r_i^c$ for each block C_i is secret-shared across both $\mathcal{C}$ and $\mathcal{V}$. The authenticity of C_i is assured to both parties by checking the tag of C_i in 2PC. Then, as $\mathcal{C}$ commits to both C_i and r_i^c before learning r_i^v, AES-GCM acts as a committing AEAD scheme from the perspective of $\mathcal{V}$. (C_i, k_i^c) acts as a perfectly binding commitment to M_i. □

Lemma 17 (Computationally hiding). *The commitment scheme Γ is computationally hiding.*

Proof. The client commitment is series of AES-GCM encrypted ciphertexts, for which $\mathcal{V}$ only holds the key shares $(\mathsf{tk}_{capp}^v, \mathsf{tk}_{sapp}^v)$. Note that the full keys are defined as $(\mathsf{tk}_{capp}, \mathsf{tk}_{sapp}) = (\mathsf{tk}_{capp}^c \oplus \mathsf{tk}_{capp}^v, \mathsf{tk}_{sapp}^c \oplus \mathsf{tk}_{sapp}^v)$. By the semantic security of the encryption scheme, we know that if the client generates two separate commitments using the same secret parameters, the encryptions of both will be indistinguishable with anything other than negligible probability. Therefore, the condition required in Definition 6 follows for all bounded adversaries. □

Appendix E. Proofs over encrypted data

DiStefano can be used to provide statments in zero knowledge about encrypted data transmitted during a TLS 1.3 session. Specically, it can provide proofs that an specific substring appears on said data which, in turn means, that the confidentiality of the data remains and only what is needed is revealed.

E.1. Revealing a substring

We briefly discuss how DiStefano can be used to implement two specific optimisations presented by DECO: "Selective Opening", which allows $\mathcal{C}$ to reveal that a certain substring is present in a plaintext M, and "Selective Redacting", which allows $\mathcal{C}$ to reveal the entirety of M other than some selection of omitted substrings.

Using our AES-GCM protocol, both approaches are easily achievable. Suppose that $\mathcal{C}$ is committing to some set of ciphertexts $C_1, \ldots, C_n$ for the purpose of proving a statement. Since $\mathcal{C}$ is required to commit to their additive shares of the decryption keys k_i^c before learning V's key shares, selectively opening C_i simply requires revealing k_i^c to $\mathcal{V}$. Similarly, $\mathcal{C}$ can selectively reveal any combination of ciphertexts by simply revealing those individual keys. In practice, revealing each block is rather cheap, requiring only 128-bits of bandwidth. In addition, this scheme can be adapted to deal with substrings inside a single block C: rather than revealing k_i^c directly, $\mathcal{C}$ and $\mathcal{V}$ instead decrypt C in a garbled circuit with the output masked by a mask ρ that is chosen by $\mathcal{C}$. We remark that this approach is somewhat fragile: for any soundness to hold, we would also require that $\mathcal{C}$ is only allowed to modify certain portions of the output plaintext. We view this difficulty as orthogonal to this work: this would require more extensive zero-knowledge proofs.

Appendix F. Comparison with prior security models

DECO protocol [2]. In [2], the authors consider an all-encompassing simulation-based (i.e. real-/ideal-world) security model, that includes the client proving knowledge of facts associated with the commitment $\mathsf{c}_{\hat{\mathsf{q}},\hat{\mathsf{r}}}$. This approach has a number of downsides. As stated previously, it does not apply when handshakes are performed using TLS 1.3, and formal guidance is not given on how to adapt their security proof to this case. Moreover, their ideal-world functionality does not establish a number of security properties that have later been shown to be guaranteed by the TLS 1.3 protocol [10]. Furthermore, the identity of the TLS server is known to $\mathcal{V}$, which contradicts one of the security properties that we aim to achieve. In summary, their proof is not modular, and so making changes to their model to incorporate these changes would be a significant task.

Oblivious TLS [6] and N-for-1-Auth [7]. In both [6] and [7], $\mathcal{C}$ and/or $\mathcal{S}$ are built as a series of entities, combining to execute the given role in the TLS handshake as an MPC functionality or *TLS engine.*

In [6], the authors show that the security of the TLS exchange in the multi-stage key exchange security model of [10] can be maintained, with some tweaks that allow the TLS engines to reveal all secret material, when a single party is corrupted. In [7], a security model that is similar to the approach of [2] is achieved, by modelling the handshake and record-layer protocols as an ideal functionality, and using the real-/ideal-world paradigm to prove security.

Note that both protocols assume that all parties learn the same information (except for a single party that relays the eventual TLS messages between the engine and the other entity). Moreover, their model does not take into account security guarantees that are only made explicit when the client attempts to attest to data that was exchanged in the TLS connection. However, the security model of Oblivious TLS is useful to us, as it allows for proving that a multi-party client can achieve a TLS 1.3 handshake with the security guarantees established in [10]. We adapt this to work in our scenario, so that we can provide similar guarantees.