

DiStefano: Decentralized Infrastructure for Sharing Trusted Encrypted Facts and Nothing More Private and Efficient Commitments for TLS-encrypted Data

Sofia Celi*, Alex Davidson†, Hamed Haddadi*¶, Gonçalo Pestana‡ and Joe Rowel§

* 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, maliciously-secure framework for generating private commitments over TLS-encrypted web traffic, for a designated third-party. DiStefano provides many improvements over previous TLS commitment systems, including: a modular protocol specific to TLS 1.3, support for arbitrary verifiable claims over encrypted data, inherent ring privacy for client browsing history, and various optimisations to ensure fast online performance of the TLS 1.3 session. We build a permissive open-source implementation of DiStefano integrated into the BoringSSL cryptographic library (used by Chromium-based Internet browsers). We show that DiStefano is practical for committing to facts in arbitrary TLS traffic, requiring < 1 s and ≤ 5 kB to execute the online phase in a LAN setting.

1. Introduction

The Transport-Layer Security (TLS) protocol [51] provides encrypted and authenticated channels between clients and servers on the Internet. Such channels commonly transmit *trusted information* about users behind clients such as proofs of age [70], social security statuses [54], and accepted purchase information. While various applications would benefit from learning such data points, doing so represents an obvious privacy concern [9], [16], [18], [44], [57], [68]. Exporting such information as anonymous credentials is non-trivial since the information resides in an encrypted and authenticated channel. Meanwhile, both legislation (such as GDPR [23]) and standards bodies (such as W3C [32]) have made usage of privacy-preserving data credentials a priority.

Designated-Commitment TLS (DCTLS) protocols¹ provide modified TLS handshakes that allow exporting certain claims over the TLS channel to a designated *verifier*. The protocols perform handshakes that secret-share private session data amongst a client and a verifier, and compute the handshake and record-layer phases in two-party computation (2PC). Examples of

DCTLS protocols include DECO [70], TLSNotary/PageSigner [59]², TownCrier [69], Garble-then-Prove [64], and Janus [43]. Similar techniques are also used to produce zero-knowledge middleboxes [28] (for proving that client traffic adheres to corporate browsing policies, for example), and for devising multi-party TLS clients/servers. Prominent examples of the latter are Oblivious TLS [1] and MPCAuth [14].

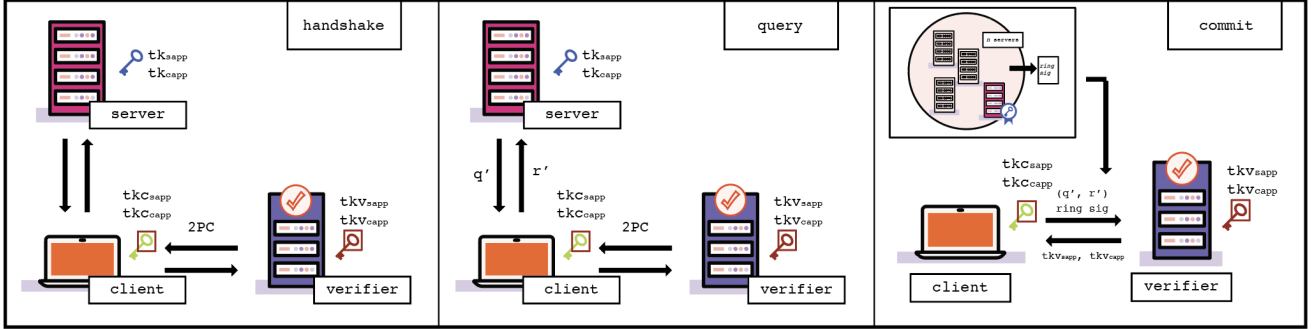
Unfortunately, while previous works claim practicality, all such DCTLS protocols appear insufficient for wide-scale usage. First, no protocol explicitly provides secure support for TLS 1.3. Support for TLS 1.3 surpassed that of 1.2 around December 2020 [45] and, according to Cloudflare Radar [15], TLS 1.3 now accounts for 63% of secure network traffic as opposed to 8.7% for TLS 1.2, so it is imperative that protocols support this version. When DCTLS protocols do support TLS 1.3, the security analysis is lacking and/or efficiency concerns that surround implementing TLS 1.3 ciphersuites and protocol steps in (maliciously-secure) 2PC are overlooked. Existing security arguments also lack in *agility*, meaning that they only apply for a static protocol, ciphersuite and 2PC primitives. This is a critical concern: already primitives used by DECO have been shown to be insecure [47], [61]. Client privacy is also neglected as the protocols reveal the server that clients communicate with to the verifier, revealing their browsing history. Second, from a usability perspective, no fully-featured open-source implementation of a DCTLS protocol exists that achieves strong security guarantees, or much less one that interoperates with common Internet browsing tools.

Our work. We design DiStefano (as seen in Fig. 1), a DCTLS protocol that securely generates private commitments over TLS 1.3 data. Security is proven using a novel standalone model that permits cryptographic agility, by allowing for the swapping various of schemes depending on the desired ciphersuite. DiStefano is pro-

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

2. We refer exclusively here to original PageSigner as TLSNotary does not appear to have a fixed cryptographic design.

Figure 1. An overview of the DiStefano protocol. In the **handshake** and **query** phases, the TLS client performs the TLS 1.3 handshake and record-layer protocols in conjunction with the verifier using 2PC to secret-share traffic keys and other session data, for establishing a secure session with the server (secret-shared keys are represented with a square over the key). In the **commitment** phase, the client authenticates the server to the verifier using a ring signature, and commits to some encrypted session data, before receiving the verifier’s secret TLS session shares.



vided as a permissive open-source implementation³ integrated into the widely-used BoringSSL library,⁴ where 2PC functionality is provided by emp [62]. With respect to client’s privacy, DiStefano guarantees *ring* privacy by using ring signatures produced over TLS certificates. Finally, the commitments generated by DiStefano can be used to produce any type of verifiable private claim: non-interactively using zero-knowledge proofs and interactively using 2PC. Note that, in this work, we prefer to build a modular framework for solving the core functionality, and leave the implementation of the subsequent proving stage up to the implementer (see Section 4.4). To ensure high performance, a number of optimisations were made to the cryptographic functionality and software implementation for DiStefano. The online portions of the handshake and record-layer phases can be executed in 500 ms and 190 ms, and with 5 kB and around 4 kB of bandwidth, respectively, for 2 kB of communication.

Formal contributions. Our formal contributions follow:

- A private Delegated-Commitment TLS 1.3 (DCTLS) protocol, DiStefano (Section 4), with a modular, standalone security framework that proves security in the presence of malicious adversaries (Section 6).
- Novel optimisations that allow running secure 2PC TLS 1.3 clients with higher efficiency (Section 5).
- An open-source, Chromium-compliant implementation integrated into BoringSSL.
- Experimental analysis that shows that DiStefano is practically efficient for committing to various sizes of Internet traffic (Section 7).

2. Background

3. <https://github.com/brave-experiments/DiStefano>

4. This library is used by most Chromium-based Internet browsers, that make up a dominant share of all browser usage.

2.1. General Notation

Vectors are denoted by lower-case bold letters. Given s , $\text{len}(s)$ denotes the length of s . The symbol $[m]$ indicates the set $1, 2, \dots, m$. We write $a \leftarrow b$ to assign the value of b to a , and $a \leftarrow \$S$ (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 . We are primarily concerned with the smallest field, \mathbb{F}_2 , where 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 $EC(\mathbb{F}_q)$.

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

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

We say that Δ is secure with respect to Game , iff $\text{Adv}_{\mathcal{A}, \Delta}^{\text{game}}(\lambda) \leq \text{negl}(\lambda)$, for some negligible function $\text{negl}(\lambda)$ 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 B): a (\mathcal{V} -assisted) handshake phase, a (\mathcal{V} -assisted) query execution phase, and a commitment phase. Previous

work, such as in DECO and tools like PageSigner, provide explicit attestation functionality for proving facts about the committed TLS session (using zero-knowledge proofs). 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 mostly incomplete) extension to TLS 1.3. 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 [2]–[4], [8]. Thus, in this work, we focus on TLS 1.3, and argue that previous protocols cannot be easily extended to handle both TLS 1.2 and 1.3, due to substantial protocol-level differences. We provide an overview of the standard TLS 1.3 handshake, and the standard notation defined in [22], in Appendix J.

Description of DCTLS phases. In Fig. 1, we give an overview of the stages of DCTLS for establishing commitments to TLS 1.3 encrypted traffic between \mathcal{C} and \mathcal{S} to be sent to a designated \mathcal{V} . 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 [51]. The following is an informal description of TLS 1.3 (1-RTT with certificate-based authentication) when extended to support DCTLS-like protocols.

Handshake phase. In this phase, \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 a regular \mathcal{C} would normally learn. This requires \mathcal{C} and \mathcal{V} to engage in the core TLS 1.3 protocol 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 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 message advertises the supported (EC)DH groups and the ephemeral (EC)DH keyshares 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 $Z = Z_C + Z_V \in EC(\mathbb{F}_c)$.

\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 \mathcal{S} -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. 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 [22].

\mathcal{S} then sends a certificate chain (in the `ServerCertificate` message -SCRT-), and a message that contains a proof that they possess the private key corresponding to the public key advertised in the 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 \mathcal{S} ’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 \mathcal{C} . Upon receiving \mathcal{S} ’s messages, \mathcal{C} verifies the signature of the SCV message and the MAC of SF. If requested, \mathcal{C} responds 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} .

Now, the handshake is completed, and \mathcal{C} and \mathcal{S} can derive the key material required by the subsequent *record layer* to exchange authenticated and encrypted application data. This derivation is performed in 2PC, and \mathcal{C} and \mathcal{V} 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 [35].

Record Layer (query execution) phase. \mathcal{C} sends a query q (in encrypted form \hat{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{r} , can then be decrypted using a similar procedure to reveal \mathcal{S} ’s 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. The fact that \mathcal{C} sends commitments before they receive \mathcal{V} ’s shares means that \mathcal{V} can trust subsequent attestations over the commitments.

Limitations of this approach. 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 [60]); or rely on trusted hardware (Town Crier [69]), against which various attacks exist [11]. Another class of oracle schemes assumes cooperation from \mathcal{S} by installing TLS extensions [52], or by changing application-layer logic [5]. These approaches suffer from two fundamental problems: they break legacy compatibility, causing a significant barrier to wide adoption; and only provide conditional exportability as \mathcal{S} has the sole discretion to determine which data can be exported, and can censor export attempts. While DECO [70] promises to solve these problems, its non-modular security design makes it impossible to swap individual pieces of functionality (without rewriting the entire security proof). These limitations have the following repercussions.

Security. Some primitives used by DECO have since been shown to be insecure [47], [61], and the security proof only targets TLS 1.2. General guidance is offered for handling TLS 1.3, but it is not formally specified. More worryingly, the security argument is all-encompassing (this is common in other constructions too [58], [69]). This significantly harms *cryptographic agility*, since *any* change to the primitives, protocol, or ciphersuites that are used theoretically dictates that an entirely new proof should be written. Lack of cryptographic agility has been shown to be a significant source of cryptographic vulnerabilities in real-world systems [49].

Privacy. Explicit authentication of \mathcal{S} to \mathcal{V} during the handshake is mandated, due to the non-modular security proof, which is harmful for client browsing privacy.

Performance. Certain underlying cryptographic tools (such as oblivious transfer protocols) have seen remarkable improvements subsequent to DECO’s publication [55], [66]. 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. Recent DCTLS protocols [13], [43], [56], [64] are either aimed entirely at TLS 1.2 [64], are entirely theoretical [56], or use semi-honest 2PC to achieve reasonable performance [13], [43] in a LAN setting. We note that the use of semi-honest 2PC must be applied carefully to prevent loss of security, and (to the best of our knowledge) no TLS 1.3 attestation mechanism has yet been proposed that provides an appropriate level of security. We discuss this, and the fact that it may lead to potential attacks, further in Appendix I. Moreover, even when semi-honest 2PC is used, performance is lacking and public implementations are rare. For example, the recently proposed Janus protocol [43] is accompanied by a reference implementation, with a reported handshake time of around 0.6 s in a LAN setting with around 1.7 GB

of traffic. By contrast, our implementation achieves malicious security guarantees in around the same time, whilst exchanging much less data (around 220MB of offline data, and 5KB of online). Thus, it does not appear that the use of malicious 2PC is a bottleneck for current protocols.

2.3. Overview of DiStefano

Due to the limitations of the previous DCTLS protocols, we aim to build a protocol that works for TLS 1.3, improves browser privacy guarantees for \mathcal{C} , does not require specific hardware or extensions, and can be easily integrated into common applications. Overall, DiStefano achieves the following.

- The creation of a maliciously-secure framework that generates binding and hiding commitments over data communicated during TLS 1.3 sessions.
- Cryptographic optimisations that ensure practical running costs, and experimental analysis showing that DiStefano is ready for real workflows.
- A publicly-available implementation integrated into the TLS library that browsers use, with no need for specialized hardware or installing extra extensions.

We believe that DiStefano is an essential step-forward for showing that DCTLS *can* be implemented in practice.

Overview of required optimisations. Our implementation of DiStefano requires several optimisations to achieve its performance. We reduce the number of rounds required to derive AES-GCM secret shares (cf. Section 5) by a factor of around 500 compared to prior art (PageSigner), and reduce the required bandwidth by around a factor of three. Moreover, we carefully combine multiple sub-circuits used in the TLS handshake to reduce the number of re-computed secrets and circuit invocations (cf. Section 4.1). We stress that a large amount of engineering time was spent on low-level tweaks: we consider this a contribution in its own right and we hope this effort allows future researchers to adapt our code easily.

3. Secure Multi-Party Computation

Two-party secure computation (2PC) protocols allow parties p_1 and p_2 to jointly compute generic functions $f(s_1, s_2)$ over their private inputs s_1 and s_2 . The security of the protocols ensures that nothing of each input is revealed to the other party, except for what f naturally reveals [46]. There are two common approaches for 2PC protocols. *Garbled circuits protocols* [26], [67] encode f as a boolean circuit and evaluate an encrypted variant of the circuit across two parties. *Threshold secret-sharing* protocols (e.g. SPDZ [17], [39], or MASCOT [41]), typically operate by first producing some random multiplicative triples (referred to as

Beaver triples [6]) 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 [42]. 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 [63] 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, consists of the following algorithms:*

- $\text{OT.Gen}(1^\lambda)$: *outputs any key material.*
- $\text{OT.Exec}(m_0, m_1, b)$: *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 via the actively secure IKNP [37], [40] extension and the Ferret [66] OT scheme. Both rely on the security of information theoretic MACs, the *learning parity with noise* (LPN) assumption, and on randomness assumptions about hash functions, see [31].

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, consists of the following algorithms:*

- $\text{MtA.Gen}(1^\lambda)$: *outputs any needed key material.*
- $\text{MtA.Mul}(\alpha, \beta)$: *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 expands this definition with an additional algorithm, $\text{MtA.Check}(a_1, \dots, b_1, \dots)$, to check shares consistency. Existing works [43], [59], [70] realise MtA with an approach [25] based on Paillier encryption [50]. We deviate from this approach to improve efficiency [65, §5], and to mitigate the need for range proofs [47], [61] (necessary for achieving malicious security). We realise the MtA functionality using the schemes introduced in [34] and [19], [20] for rings of characteristic > 2 and 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 [19], [20] reduces directly to an NP-hard encoding problem [36], to the best of our knowledge, there is no computational hardness proof for [34].

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 coordinate of Z , we use the *elliptic curve to field* (ECtF) functionality [70] 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 compared with attempting to combine arithmetic and binary approaches in a garbled circuit. We stress that use of the ECtF functionality improves performance: 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. We achieve malicious security by instantiating the multiplication with a maliciously-secure MtA scheme.

4. DiStefano Protocol

In this section, we fully describe each of the phases of the DiStefano protocol (formal ideal functionalities are given in Appendix B). A diagram of the full protocol is found in Fig. 2. For comparison, we also provide a diagram of TLS 1.3 and a summary of the shorthands that are taken from [22] in Appendix J. The security analysis is handled in Section 6 and Appendix C.

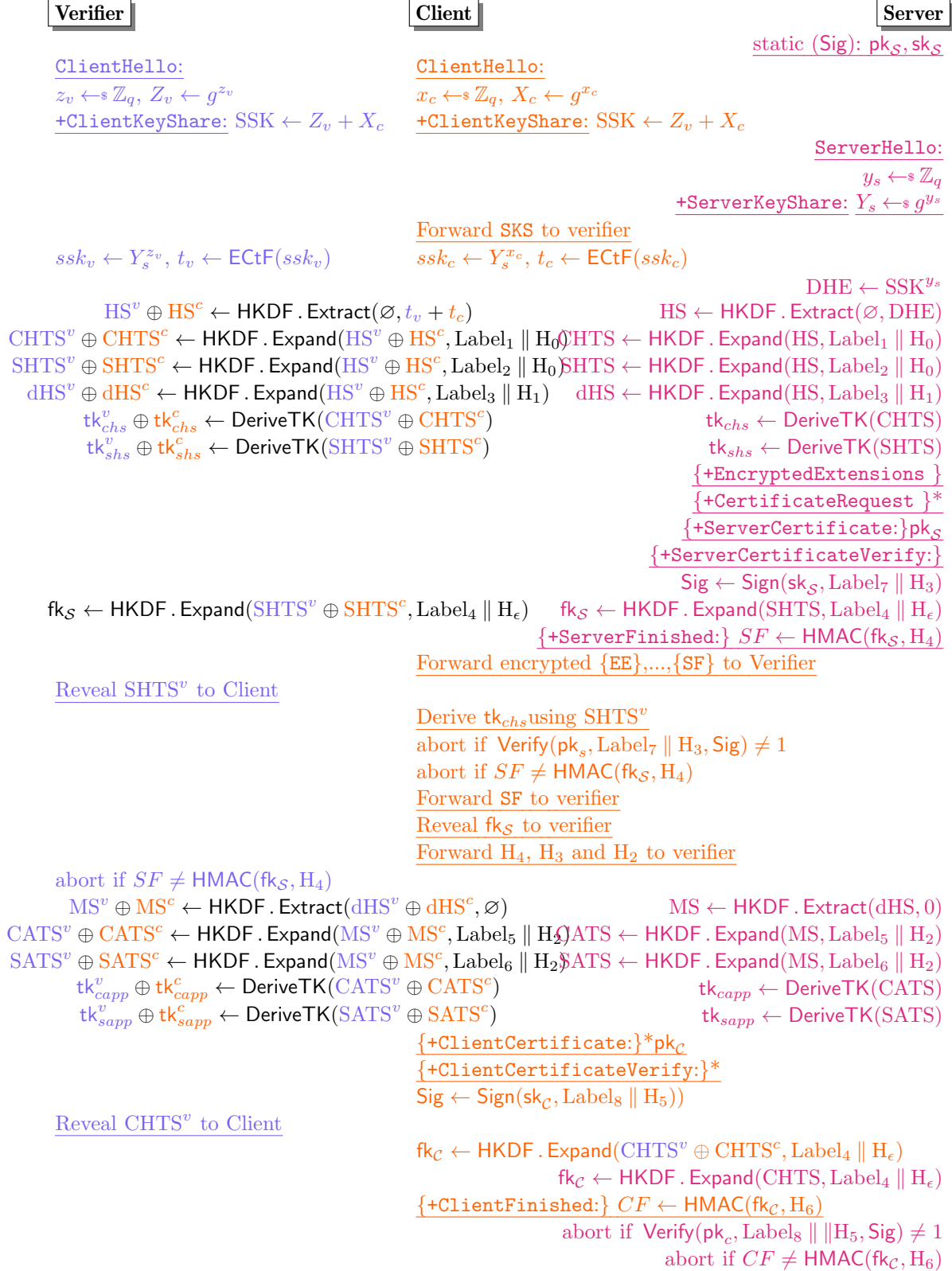
4.1. Handshake Phase: HSP

We use the similar overarching mechanism for the handshake phase as described in Section 2.2, but focused exclusively on TLS 1.3 with AES-GCM as the AEAD scheme (Section A.2), using ECDH for the shared key generation, and using ECDSA certificates. The 2PC ideal functionalities that we use are defined in Algorithms 1 to 3 (Section C.1). However, the protocol can be adapted to work with any other TLS 1.3-compliant ciphersuites that are compatible with 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 \mathcal{S} . For 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. 2. This can be computed without 2PC.

\mathcal{C} then sends the CH and the CKS messages, advertising SSK as part of the `key_shares` extension. \mathcal{S} 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. CHTS, SHTS, $\text{tk}_{sh,s}$, $\text{tk}_{ch,s}$). Once \mathcal{C} and \mathcal{V} receive the SH message, they derive additive shares of the shared ECDH key as $E = x_c \cdot Y_s + x_v \cdot Y_s$.

Figure 2. The DiStefano 1-RTT handshake protocol. Shorthands correspond to those defined in [22]. **Purple** represents messages sent or calculated by \mathcal{V} , **orange** by the client, **pink** by the server, and **black** for 2PC calculations between the client and verifier. Messages with an asterisk (*) are optional, and those within braces ({}) are encrypted.

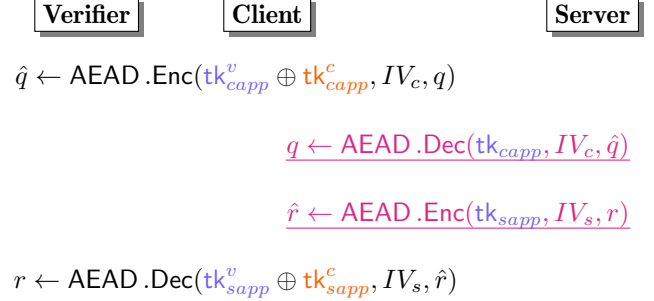


As TLS 1.3 key derivation operates on the x co-ordinate of the shared key, \mathcal{C} and \mathcal{V} convert their additive shares of $E = (E_x, E_y)$ into additive shares $E_x = t_c + t_v$ by running the ECtF functionality. With E_x computed, \mathcal{C} and \mathcal{V} proceed to run the TLS 1.3 handshake key derivation circuit in 2PC, with each party learning shares $HS^v \oplus HS^c = \text{HKDF.Extract}(\emptyset, t_v + t_c)$. In practice, this process is carried out inside a garbled circuit that produces shares of CHTS, SHTS and dHS, as well as the SF message key fk_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, as it provides authenticity guarantees to \mathcal{V} .

Authentication phase. \mathcal{S} sends the CR (if wanted), SCRT, SCV and SF messages. The SF message is computed by first deriving a finished key fk_S from SHTS and then computing a MAC tag SF over a hash of all the previous handshake messages. 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}) while waiting for the final flight of \mathcal{C} messages.

\mathcal{C} receives the encrypted messages from \mathcal{S} and, in turn, forwards them (encrypted) 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 SF messages without the involvement of \mathcal{V} . Moreover, as \mathcal{C} now knows tk_{chs} they are also able to respond to the CR if one exists. \mathcal{C} then forwards an authentic copy of the hashes H_2 , H_3 , and H_4 to \mathcal{V} , allowing them to check the SF message’s authenticity. Notice that \mathcal{C} does not forward the decrypted SCRT message to \mathcal{V} , as this message reveals the identity of the server. At this point, though, \mathcal{C} can construct a ring signature (schemes such as [27]) 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 SF message, allowing \mathcal{V} to validate the SF. Finally, \mathcal{C} and \mathcal{V} derive the shares of the traffic secrets MS, CATS, SATS and the traffic keys tk_{sapp} , 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 expensive.

Figure 3. The DiStefano query execution protocol. Purple represents messages sent or calculated by \mathcal{V} , orange by \mathcal{C} , pink by \mathcal{S} , and black for 2PC between \mathcal{C} and \mathcal{V} .



4.2. Query Execution Phase: QP

Once HSP has completed, \mathcal{C} and \mathcal{V} move into the query phase (Fig. 3). 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 $q = q_1, \dots, q_n$ 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.

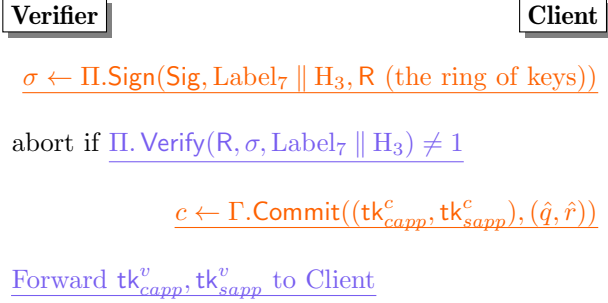
In practice, we instantiate this portion of the protocol using the AES-GCM approach described in Section 5. There is no explicit dependence on AES-GCM: any AEAD cipher supported by TLS 1.3 will suffice. We highlight this, and the general security formalisation of the query phase, in Section 6.

Extending the query phase to multiple rounds is straightforward using AES-GCM. We discuss the details of committing to ciphertexts in Section 5.1, but the main idea is that, as each ciphertext block q_i is encrypted with a unique key $\text{AES.Enc}(k, IV + i)$, \mathcal{C} and \mathcal{V} can arbitrarily reveal their shares of $k_i^c + k_i^v = \text{AES.Enc}(k, IV + i)$ at any stage of the query 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 permits many useful applications, as \mathcal{C} and \mathcal{V} can now nest commitment rounds inside of the query phase.

4.3. Commitment Phase: CP

The objectives of the commitment phase (Fig. 4) are twofold: i) to assure \mathcal{V} of the authenticity of \mathcal{S} without revealing which server \mathcal{C} is communicating with; and ii) to allow \mathcal{C} to learn secret shares held by \mathcal{V} only after

Figure 4. 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 \mathcal{V} , and orange by \mathcal{C} .



they have produced binding commitments to a specific portion of the TLSsession with \mathcal{S} .

For validating the authenticity of the server, \mathcal{V} ring-authenticates (as mentioned previously, using a ring signature or similar) the TLS server that they communicated with, as one of N servers from which \mathcal{V} accepts attestations.⁵ After ring authentication, \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.

- $(\hat{q}_i, \hat{r}_i) \leftarrow \Gamma.\text{Commit}(\text{sp}_C, (\hat{q}, \hat{r}, i))$: 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 .
- $\text{sp}_V \leftarrow \Gamma.\text{Challenge}(c)$: Output the secret parameters of \mathcal{V} .
- $b \leftarrow \Gamma.\text{Open}((\text{sp}_C, \text{sp}_V), (\hat{q}_i, \hat{r}_i), (q_i, r_i))$: 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 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). We prove that Γ is a perfectly binding, and computationally hiding commitment scheme in Appendix F.

4.4. Subsequent Phases

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 DCTLS commitments, without revealing anything else. This could be done using zero-knowledge proofs, selective opening strategies (as is used

5. This could be performed during the handshake phase. For performance reasons, it is preferable to communicate in the commitment phase, when online communication is no longer constrained by potential handshake time-outs.

in DECO), or subsequent 2PC. The formal commitment opening process that we described previously can be used for this, since \mathcal{C} can now use the combined secret parameters $(\text{sp}_C, \text{sp}_V)$ to prove any statement about the commitment (\hat{q}_i, \hat{r}_i) . See Appendix G for more details.

5. AES-GCM Specifics

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 [35]. We use this algorithm for both encrypting the corresponding handshake messages and any application traffic. Here we describe how to commit to ciphertexts in AES-GCM (Section 5.1), as well as optimisations that make it more amenable to 2PC evaluation of the encryption and decryption procedures (Section 5.2).

5.1. Commitment to AES-GCM ciphertexts

Recall that, in DiStefano, both \mathcal{C} and \mathcal{V} learn all AES-GCM ciphertext blocks $C_i = M_i \oplus \text{AES}.\text{Enc}(k, IV + i)$ produced by \mathcal{S} . As $k = k^c + k^v$ is secret shared, \mathcal{C} and \mathcal{V} check the integrity of the C_i 's using the approach described in Section 5. Assuming this check is valid, \mathcal{C} locally generates n masks b_i^c and commits to each mask individually. This early commitment is necessary to circumvent the non-committing nature of AES-GCM [29]. After these commitments have been made, \mathcal{C} and \mathcal{V} then call into a garbled circuit, receiving additive shares of $b_i^c + b_i^v = \text{AES}.\text{Enc}(k, IV + i)$. In other words, \mathcal{V} learns $\text{AES}.\text{Enc}(k, IV + i) + b_i^c$, and \mathcal{C} learns nothing new. Finally, \mathcal{C} learns each k_i^v . 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_i^c to \mathcal{V} , allowing \mathcal{V} to decrypt C_i . This technique is referred to as *selective opening* in the DECO protocol: we provide more details on the uses of this technique in Appendix G. We prove the security of this commitment scheme in Appendix F.

5.2. 2PC optimisations

In this section, we discuss some optimisations that are necessary for ensuring high performance of AES-GCM encryption/decryption during online TLS operations. The ideal functionalities that we use to describe AES-GCM in 2PC, along with the security proofs of these optimisations, are given in Appendix D. We briefly recall how standard AES-GCM operates.

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

$M = (M_1, \dots, M_n)$, AES-GCM applies counter-mode encryption to produce the ciphertext blocks $C_i = M_i \oplus \text{AES.Enc}(k, IV + i)$. To ensure authenticity, AES-GCM also outputs a tag $\tau = \text{Tag}_k(IV, C, A)$ 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: $\tau(A, C, k, IV) = \text{AES.Enc}(k, IV) \oplus P_{A||C||\text{len}(A)||\text{len}(C)}(h)$ where $h = \text{AES.Enc}(k, 0)$.

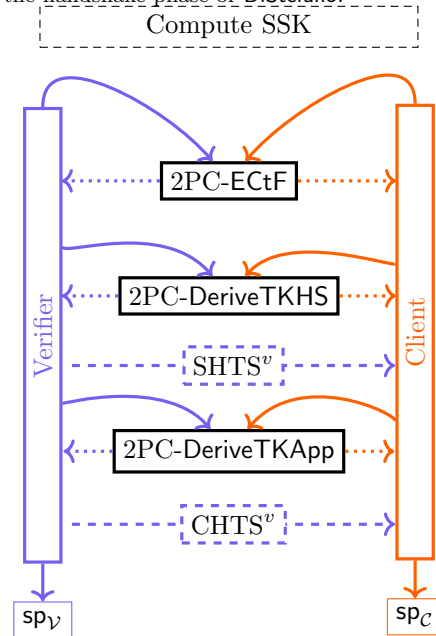
Efficiency. Despite its simplicity, executing AES-GCM encryptions in a multi-party setting can be challenging due to the use of binary and arithmetic operations. For example, whilst AES operations are well-suited for garbled circuits, a single multiplication over $\mathbb{F}_{2^{128}}$ typically requires around 16K AND gates, increasing the cost by nearly a factor of 3. To mitigate this cost, both [70] and [59] 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, MPCAuth 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 $\tau_c = P_{A||C||\text{len}(A)||\text{len}(C)}(\{h_c^i\})$ and $\tau_v = P_{A||C||\text{len}(A)||\text{len}(C)}(\{h_v^i\})$, respectively) over $\mathbb{F}_{2^{128}}$ and $n + 1$ 2PC evaluations of AES [59], [70]. The final tag is achieved by simply computing $\tau = \tau_c + \tau_v \oplus \text{AES.Enc}(k_c + k_v, IV_c)$. 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.

Our optimisations. 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 MPCAuth’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 $h = h_c + h_v$, C and V iteratively compute additive

Figure 5. Ordered execution of 2PC exchange between C and the V during the handshake phase of DiStefano.



shares of $l_c + l_v = h^n = (h_c + h_v)^{n-1}(h_c + h_v)$ 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.

We improve upon this by replacing the additive sharing $h = h_1 + h_2$ with $h = h_1/h_2$, i.e. using a multiplicative sharing. By using multiplicative shares, we can run each MtA computation in parallel, with each P_i supplying $h_i, h_i^3, \dots, h_i^{1023}$ 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 [30] 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 7.

6. Security Analysis

Previous DCTLS protocols use all-encompassing ideal functionalities and Universally-Composable (UC)

security proofs [12], proving that the entire flow from handshake to attestation is secure. This is problematic for cryptographic agility, as it means that any modification to the TLS ciphersuite, 2PC functionality, or protocol extensions would necessitate a complete rewrite of the proof. Such agility is critical for building flexible secure systems, that can be modified easily if our understanding of cryptographic primitives and systems change [49].

We reimagine the security model for DCTLS protocols in two ways. First, we move the security analysis to the *standalone model*. UC security proofs are particularly useful when building atomic protocol primitives, that may be used in arbitrary composition with other primitives. Since DCTLS is a high-level protocol that is likely to be used as a single application, we believe that the standalone model captures a sufficiently natural security requirement, without the added complexity needed to ensure UC security. Second, our analysis breaks the protocol down into its three phases: the handshake phase, the query phase, and the commitment phase; and proves that each is secure independently. We give a short overview of how we model security for each phase of the protocol in the following. Due to space limitations, the full standalone security model is covered in Appendix C.

Handshake phase. We model the handshake phase similarly to Oblivious TLS [1]. Essentially, this model proves that we can satisfy the original guarantees proven about TLS 1.3 [22] (i.e. related to *Match* and *Multi-Stage* security) even when executing certain functionalities in 2PC. One key difference is that the presence of the verifier ensures that potential TLS 1.3 adversaries can alter the derivation of secrets in the client, and thus we similarly base security on a modified *Shifted PRF ODH* assumption, see [1, Definition 2] for more details. While Oblivious TLS opts for a UC-security proof, we use a standalone depiction for simplicity reasons, since each 2PC functionality is used in sequence. The general protocol execution is given in Fig. 5.

Query execution phase. The query phase of DiStefano essentially amounts to considering a 2PC realisation of the record-layer of the TLS 1.3 protocol. We define 2PC ideal functionalities (Algorithms 4 and 5) that abstracts the core encryption and decryption functionality for application traffic. We eventually show that we can prove security of this phase based on the 2PC realisation of the AES-GCM functionalities that we formalise in Appendix D (Algorithms 6 and 7). These functionalities implement the functionality and optimisations described in Section 5. To prove security of alternative ciphersuites, it is simply a matter of implementing the 2PC ideal functionalities using different primitives.

Commitment phase. We separate the commitment phase from the previous two phases, and use a game-based security model (Section C.3). These game-based security notions (7) evaluate the capacity of the protocol

to satisfy: *session privacy* (SPriv), that ensures that committed sessions are indistinguishable; *ring authentication* (SAuth_n¹), that the client is forced to successfully ring authenticate the TLS server amongst N possible a priori-chosen servers; and *session unforgeability* (SUnf), that ensures that the client cannot arbitrarily forge sessions that did not occur. In the end, we show that each of these properties is proven with respect to DiStefano, by the binding and hiding commitment scheme devised from AES-GCM (Appendix F), and a sufficient ring signature scheme for ECDSA certificates [27] (Section A.3).

7. Experimental Analysis

Implementation. In order to enable easy integration with other cryptographic libraries and browsers, we implemented a prototype of DiStefano in C++.⁶ 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. 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. 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 MPCAuth’s circuit generation to produce the relevant garbled circuits. We further reduce the online cost of MPCAuth’s secret sharing scheme by using a pre-determined splitting scheme for specific secrets.⁷ We provide a full listing of the changes made to third party libraries alongside our prototype.

Results. 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 8 GB of RAM) for \mathcal{C} and a server-grade device (an Intel Xeon Gold 6138 with 32 GB of RAM) for \mathcal{V} . All communication used in TLS 1.3 was carried out using a single thread over a 1 Gbps network with a round-trip time of around 16 ms. 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).

Table 1 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. 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.

6. <https://github.com/brave-experiments/DiStefano>

7. We stress that this approach is less flexible than MPCAuth’s approach. For example, our approach only supports 2 parties, whereas MPCAuth supports arbitrarily many.

Table 1. GARBLED CIRCUIT TIMINGS AND BANDWIDTH.

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

Each garbled circuit is reported in terms of offline/online times (ms) and bandwidth (MB) costs. “K” means Karatsuba and “N” means Naive. “LD” refers to “LeakyDeltaOT” and “FC” means “FerretCOT”.

AES-GCM tagging and verification may lead to faster running times (see [63, §7] for concrete speed-ups). We also compare the offline time using the original implementation of authenticated garbling (LeakyDeltaOT [63]) that uses 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 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 2 shows the results for each arithmetic primitive used. Given that all running times and bandwidth counts are rather low, we do not expect this 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 all represents an improvement of around 4 orders of magnitude over using a garbled circuit.

Preliminary timings indicate that our implementation of DiStefano is competitive with DECO, with the DCTLS portion taking around 500 ms to complete for a 256-bit secret. These conclusions are consistent across

Table 2. PRIMITIVE TIMINGS AND BANDWIDTH.

Primitive	Time (ms)	Bandwidth (MB)
ECtF (<i>P256</i>)	336.1	0.415
ECtF (<i>P384</i>)	335.5	0.648
ECtF (<i>P521</i>)	421.4	1.22
MtA (<i>P256</i>)	33.67	0.075
MtA (<i>P384</i>)	40.65	0.127
MtA (<i>P521</i>)	55.83	0.241
AES-GCM powers (mul.)	1694	0.049
AES-GCM powers (add.)	5926	0.080
AES-GCM powers (GC)	—	900

Table 3. E2E TIMINGS AND BANDWIDTH FOR DCTLS

Process	Time (ms)	Bandwidth (MB)
\mathcal{C} and \mathcal{S} Key Share	1.3167	1.335e-04
\mathcal{C} and \mathcal{V} execute ECtF	0.008083	9.53674e-07
Circuit Preprocessing	6280.08	220.484
\mathcal{S} sends cert.	0.011375	3.14713e-05
Derive traffic secrets	33.1389	0.0276108
Derive GCM shares	136.573	0.0488291

both the individual and E2E timings (cf. Table 3).

Comparisons with prior work. We would like to point out that comparisons between our results and others in the literature [59], [70] should be made carefully. In the case of DECO, their implementation is not publicly available, and we were unable to reproduce any of their results. Moreover, as our implementation is single-threaded, we are unable to take advantage of emp’s multi-threaded pre-processing. Given that [63, §7] reports an order of magnitude increase in bandwidth due to multi-threading, it is not surprising that our offline times are an order of magnitude higher. However, our online timings are comparable with DECO, and parallelising the pre-processing stage would likely mitigate any discrepancies⁸. As pre-processing can be carried out before, we do not consider this a major issue.

It is also difficult to compare our timings to PageSigner. Their original implementation is written entirely in Javascript, preventing the usage of dedicated hardware resources. Given that our implementation is instead written in C++, we might expect DiStefano to be faster. PageSigner also follows a semi-honest security model and targets TLS 1.2: these are incompatible with DiStefano.

8. Discussion

8.1. 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. We summarise the comparison in Table 4, and discuss further below.

⁸ Notably, [70] does not mention if the pre-processing used multiple threads.

The DECO and PageSigner protocols, for example, only (formally) work for TLS 1.2 and under, and provide limited privacy. TownCrier [69] 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 *MPCAuth* protocol [58] 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. In the *enrollment* phase, 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. In the *authentication* phase, the user runs the MPCAuth 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 (such as key recovery, for instance).

The *Oblivious TLS* protocol [1] 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; and 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 [28] has many similarities with techniques used in DCTLS protocols. However, the verifier is considered to be an on-path proxy that both receives and forwards encrypted traffic between the parties (similar to the proxy model of DECO [70]). Furthermore, the client is only required to produce commitments to their own traffic, rather than the traffic that is received from the server. Applications involve those that include corporate oversight and enactment of Internet browsing policies to be enforced by middleboxes, which are naturally thwarted in a setting where all client traffic is sent encrypted over TLS.

Table 4. COMPARISON OF DCTLS-LIKE PROTOCOLS.

Protocol	TLS 1.3	Attest	Ring auth
DECO-like [69], [70]	✗	✓	✗
MPCAuth [14]	✓	✗	✗
Oblivious TLS [1]	✓	✗	✗
ZKMiddleboxes [28]	✓	$C \rightarrow S$	✗
DiStefano	✓	✓	✓

Table 5. RESULTS FOR RUNNING *KeyUpdate* IN 2PC.

Circuit	OT	Offline (ms)	Online (ms)	Bandwidth (MB)
KeyUpdate	LD-OT	10540	29.95	98.54
KeyUpdate	FC-OT	7960	31.96	31.61

Concurrent work. In work concurrent to ours, Xie et al. [64] proposed a series of optimisations to the MPC protocols used inside DECO, targeting TLS 1.2. 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. This approach is somewhat reminiscent of the highly optimised CBC-HMAC protocol proposed in DECO [70, §4.2.1] for computing tags in 2PC. We remark that incorporating this particular optimisation into our secret derivation process seems non-trivial, and we discuss these difficulties in Appendix H.

8.2. Applications

DiStefano can be used to commit to encrypted TLS 1.3 data. As noted in [70], 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, 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 that they can prove certain statements for, including proof of confidential financial information, and proof of age. It should be noted that TLS sessions could serve as the basis for more generic user credentials, proving arbitrary facts about a user. For a more complete summary, see [70].

8.3. Limitations

Our implementation of DiStefano does not support key rotation via *KeyUpdate* messages or full 0-RTT mode, but this limitation is not major: it can be circumvented by simply re-running the HSP.⁹ We also provide no concrete instantiation of the zero-knowledge primitives that can be used to create attestations, but they should follow the guidelines stated in Section 4.4. Said proofs

9. For completeness, we benchmarked the cost of running the *KeyUpdate* operation in a garbled circuit, see Table 5.

must also be mindful of user privacy concerns: if proof circuits explicitly target server-specific HTML formats, this will undo the ring authentication privacy guarantees.

DCTLS protocols must assume user commitments are *meaningful*, that the TLS server stores only *correct* data. Suppose that Alice wishes to provide a proof of their age from a particular Government agency website. Alice logs in to the website, and then runs DiStefano to produce a commitment to their age based on the data present there. This process assumes that the account Alice logs in to is their own, which may not be the case, e.g. if the account is stolen or fake. \mathcal{V} should only accept commitments from servers that it trusts to correctly store user data.

Finally, DCTLS protocols could become actively harmful tools for monitoring or censoring client traffic in certain applications, especially those without 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.

8.4. 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, which can then be applied during the process of building the browser based on build flags.¹⁰ We leave the completion and deployment as future work.

9. Conclusion

We build DiStefano, a DCTLS protocol that generates private commitments to encrypted TLS 1.3 data. We use a modular, standalone security framework that provides malicious security, and guarantees ring privacy for client browsing patterns. We provide an open-source integration in BoringSSL, and demonstrate the online efficiency of DiStefano for believable workloads.¹¹ The flexibility, security, and usability of DiStefano makes it an immediate candidate for real-world applications and Internet browser integration.

Acknowledgement

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] Damiano Abram, Ivan Damgård, Peter Scholl, and Sven Trieflinger. Oblivious TLS via multi-party computation. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 51–74. Springer, Heidelberg, May 2021.
- [2] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 5–17, New York, NY, USA, 2015. Association for Computing Machinery.
- [3] Kenichi Arai and Shinriqichiro Matsuo. Formal verification of TLS 1.3 full handshake protocol using proverif (Draft-11). IETF TLS mailing list, 2016.
- [4] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, Austin, TX, August 2016. USENIX Association.
- [5] A. Backman, J. Richer, and M. Sporny. Signing http messages. IETF draft. Accessed 14/11/2022.
- [6] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.
- [7] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Heidelberg, August 1994.
- [8] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, 2015.
- [9] Pandora Blake. Age verification for online porn: more harm than good? *Porn Studies*, 6(2):228–237, 2019.
- [10] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. PRF-ODH: Relations, instantiations, and impossibility results. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 651–681. Springer, Heidelberg, August 2017.
- [11] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wensisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, August 2018. USENIX Association.
- [12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
- [13] Kwan Yin Chan, Handong Cui, and Tsz Hon Yuen. Dido: Data provenance from restricted tls 1.3 websites. *Cryptology ePrint Archive*, Paper 2023/1056, 2023. <https://eprint.iacr.org/2023/1056>.

10. Indeed, such a system is already used for the Brave Browser.

11. <https://github.com/brave-experiments/DiStefano>

- [14] Weikeng Chen, Ryan Deng, and Raluca Ada 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>.
- [15] Cloudflare. Tls 1.2 vs. tls 1.3 vs. quic: Distribution of secure traffic by protocol, 2023. Accessed 11/04/2023.
- [16] John T Cross. Age verification in the 21st century: Swiping away your privacy. *J. Marshall J. Computer & Info. L.*, 23:363, 2004.
- [17] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.
- [18] Jonathan J Darrow and Stephen D Lichtenstein. Do you really need my social security number-data collection practices in the digital age. *NCJL & Tech.*, 10:1, 2008.
- [19] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy*, pages 980–997. IEEE Computer Society Press, May 2018.
- [20] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *2019 IEEE Symposium on Security and Privacy*, pages 1051–1066. IEEE Computer Society Press, May 2019.
- [21] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1197–1210. ACM Press, October 2015.
- [22] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology*, 34(4):37, October 2021.
- [23] European Commission. GDPR: Right to Portability, Art. 20. <https://gdpr-info.eu/art-20-gdpr/>. Accessed 5th September 2023., 2014.
- [24] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google’s QUIC protocol. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1193–1204. ACM Press, November 2014.
- [25] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1179–1194. ACM Press, October 2018.
- [26] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design (extended abstract). In *27th FOCS*, pages 174–187. IEEE Computer Society Press, October 1986.
- [27] Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 253–280. Springer, Heidelberg, April 2015.
- [28] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 4255–4272. USENIX Association, August 2022.
- [29] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 66–97. Springer, Heidelberg, August 2017.
- [30] Shay Gueron and Michael E. Konavis. Intel® carry-less multiplication instruction and its usage for computing the gcm mode, 2014. Accessed 14/03/2023.
- [31] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy*, pages 825–841. IEEE Computer Society Press, May 2020.
- [32] Amy Guy, Manu Sporny, Drummond Reed, and Markus Sabadello. Decentralized identifiers (DIDs) v1.0. W3C recommendation, W3C, July 2022. <https://www.w3.org/TR/2022/REC-did-core-20220719/>.
- [33] Felix Günther. Modeling advanced security aspects of key exchange and secure channel protocols. *it - Information Technology*, 62(5-6):287–293, 2020.
- [34] Iftach Haitner, Nikolaos Makriyannis, Samuel Ranellucci, and Eliad Tsfadia. Highly efficient OT-based multiplication protocols. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 180–209. Springer, Heidelberg, May / June 2022.
- [35] Ralph Holz, Jens Hiller, Johanna Amann, Abbas Razaghpanah, Thomas Jost, Narseo Vallina-Rodriguez, and Oliver Hohlfeld. Tracking the deployment of tls 1.3 on the web: A story of experimentation and centralization. *SIGCOMM Comput. Commun. Rev.*, 50(3):3–15, jul 2020.
- [36] Russell Impagliazzo and Moni Naor. Efficient cryptographic schemes provably as secure as subset sum. *Journal of Cryptology*, 9(4):199–216, September 1996.
- [37] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.
- [38] Tetsu Iwata, Keisuke Ohashi, and Kazuhiko Minematsu. Breaking and repairing GCM security proofs. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 31–49. Springer, Heidelberg, August 2012.
- [39] Marcel Keller. MP-SPDZ: A versatile framework for multiparty computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020.
- [40] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.
- [41] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MAS-COT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.
- [42] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfssdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.
- [43] Jan Lauinger, Jens Ernstberger, Andreas Finkenzeller, and Sebastian Steinhorst. Janus: Fast privacy-preserving data provenance for tls 1.3. Cryptology ePrint Archive, Paper 2023/1377, 2023. <https://eprint.iacr.org/2023/1377>.
- [44] Rick S Lear and Jefferson D Reynolds. Your social security number or your life: Disclosure of personal identification information by military personnel and the compromise of privacy and national security. *BU Int’l LJ*, 21:1, 2003.

- [45] Hyunwoo Lee, Doowon Kim, and Yonghwi Kwon. Tls 1.3 in practice: how tls 1.3 contributes to the internet. In *Proceedings of the Web Conference 2021*, WWW '21, page 70–79, New York, NY, USA, 2021. Association for Computing Machinery.
- [46] Yehuda Lindell and Benny Pinkas. Secure multiparty computation for privacy-preserving data mining. *Cryptology ePrint Archive*, Report 2008/197, 2008. <https://eprint.iacr.org/2008/197>.
- [47] Nikolaos Makriyannis and Udi Peled. A note on the security of gg18, 2021. https://info.fireblocks.com/hubfs/A_Note_on_the_Security_of_GG.pdf.
- [48] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *22nd ACM STOC*, pages 427–437. ACM Press, May 1990.
- [49] David Ott, Kenny Paterson, and Dennis Moreau. Where is the research on cryptographic transition and agility? *Commun. ACM*, 66(4):29–32, mar 2023.
- [50] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.
- [51] E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor, August 2018.
- [52] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, and Srdjan Capkun. TLS-N: Non-repudiation over TLS enable ubiquitous content signing. In *NDSS 2018*. The Internet Society, February 2018.
- [53] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 552–565. Springer, Heidelberg, December 2001.
- [54] Michael Rosenberg, Jacob White, Christina Garman, and Ian 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>.
- [55] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 94–124, Virtual Event, August 2021. Springer, Heidelberg.
- [56] Manuel B. Santos. Peco: methods to enhance the privacy of deco protocol. *Cryptology ePrint Archive*, Paper 2022/1774, 2022. <https://eprint.iacr.org/2022/1774>.
- [57] Berin Szoka and Adam D Thierer. Coppa 2.0: The new battle over privacy, age verification, online safety & free speech. *Progress & Freedom Foundation Progress on Point Paper No*, 16, 2009.
- [58] Sijun Tan, Weikeng Chen, Ryan Deng, and Raluca Ada Popa. Mpcauth: Multi-factor authentication for distributed-trust systems. *Cryptology ePrint Archive*, Paper 2021/342, 2023. <https://eprint.iacr.org/2021/342>.
- [59] PageSigner team. Pagesigner: One-click website auditing. Website. Accessed 04/04/2023.
- [60] TLSNotary team. Tlsnotary: Proof of data authenticity. Website. Accessed 04/04/2023.
- [61] Dmytro Tymokhanov and Omer Shlomovits. Alpha-rays: Key extraction attacks on threshold ecDSA implementations. *Cryptology ePrint Archive*, Paper 2021/1621, 2021. <https://eprint.iacr.org/2021/1621>.
- [62] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. <https://github.com/emp-toolkit>, 2016.
- [63] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingam, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 21–37. ACM Press, October / November 2017.
- [64] Xiang Xie, Kang Yang, Xiao Wang, and Yu Yu. Lightweight authentication of web data via garble-then-prove. *Cryptology ePrint Archive*, Paper 2023/964, 2023. <https://eprint.iacr.org/2023/964>.
- [65] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. Efficient online-friendly two-party ECDSA signature. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 558–573. ACM Press, November 2021.
- [66] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1607–1626. ACM Press, November 2020.
- [67] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.
- [68] Majid Yar. Protecting children from internet pornography? a critical assessment of statutory age verification and its enforcement in the uk. *Policing: An International Journal*, 43(1):183–197, 2020.
- [69] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 270–282, New York, NY, USA, 2016. Association for Computing Machinery.
- [70] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1919–1938. ACM Press, November 2020.

Appendix A. Additional Cryptographic Preliminaries

We provide some additional cryptographic preliminaries that are required for arguing the security of our system.

A.1. Commitment Schemes

Definition 3 (Commitment scheme). *A commitment scheme Γ is a tuple consisting of the following algorithms:*

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

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

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

Definition 4 (Binding property). Given $\text{sp} \leftarrow \Gamma.\text{Gen}(1^\lambda)$. We say that Γ is a computationally binding commitment scheme if, for any PPT algorithm, the following holds:

$$\Pr \left[0 \leftarrow \Gamma.\text{Open}(\text{sp}, c^*, t^*, x') \mid \begin{array}{l} (x^*, c^*) \leftarrow \mathcal{A}(1^\lambda) \\ t^* \leftarrow \Gamma.\text{Challenge}(c^*) \\ x' \leftarrow \mathcal{A}(1^\lambda); x' \neq x^* \end{array} \right] > 1 - \text{negl}(\lambda).$$

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

Definition 5 (Hiding property). Let $\text{sp} \leftarrow \Gamma.\text{Gen}(1^\lambda)$, $\{x_b\}_{b \in \{0,1\}} \in \{0,1\}^2$, and $\{c_b \leftarrow \Gamma.\text{Commit}(\text{sp}, x_b)\}$. We say that Γ is a computationally hiding commitment scheme if, for any PPT algorithm, the following holds:

$$\Pr \left[d^* \stackrel{?}{=} d \mid \begin{array}{l} d \leftarrow \mathcal{A}(\{0,1\}) \\ d^* \leftarrow \mathcal{A}(1^\lambda, c_d, (x_0, x_1)) \end{array} \right] < 1/2 + \text{negl}(\lambda).$$

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

We show in Appendix F that AES-GCM ciphertext commitment scheme in Section 5.1 is perfectly binding and computationally hiding for TLS 1.3 encrypted data. A high-level overview of the commitment phase based on this scheme is given in Section 4.3.

A.2. 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.

- $k \leftarrow \text{AEAD}.\text{keygen}(1^\lambda)$: Outputs a key $k \leftarrow \mathcal{K}$.
- $(C, \tau) \leftarrow \text{AEAD}.\text{Enc}(k, m; A)$: 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 $\tau \in \mathcal{T}$.
- $m \vee \perp \leftarrow \text{AEAD}.\text{Dec}(k, C, \tau; A)$: 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 6 (Correctness). AEAD is correct if and only if the following holds true.

$$\Pr \left[m \leftarrow \text{AEAD}.\text{Dec}(k, C, \tau; A) \mid \begin{array}{l} k \leftarrow \text{AEAD}.\text{keygen}(1^\lambda) \\ (C, \tau) \leftarrow \text{AEAD}.\text{Enc}(k, m; A) \end{array} \right] = 1$$

Definition 7 (Security). An AEAD scheme is secure if it satisfies the IND-CCA notion of security [48].

It is widely known that the AES-GCM block cipher mode of operation satisfies these guarantees [38], 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.

Figure 6. Security games for establishing anonymity and unforgeability guarantees of a ring signature scheme Π .

Anon	
1 :	$\{\text{sk}_i, \text{vk}_i\}_{i \in [n]} \leftarrow \Pi.\text{Gen}(1^\lambda)$
2 :	$(m, R, i_0, i_1) \leftarrow \mathcal{A}^{\mathcal{O}_S, \mathcal{O}_C}(\{\text{vk}_i\}_{i \in [n]})$
3 :	if $[(i_0, i_1 \notin [n]) \vee (\text{vk}_{i_0}, \text{vk}_{i_1} \notin R)]$: abort
4 :	$d \leftarrow \mathcal{A}(\{0,1\})$
5 :	$\sigma \leftarrow \Pi.\text{Sign}(\text{sk}_{i_d}, m, R)$
6 :	$d' \leftarrow \mathcal{A}(\sigma)$
7 :	if $[d' \stackrel{?}{=} d]$: return 1
8 :	return 0
Unf	
1 :	$\{\text{sk}_i, \text{vk}_i\}_{i \in [n]} \leftarrow \Pi.\text{Gen}(1^\lambda)$
2 :	$(m^*, R^*, \sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}_S, \mathcal{O}_C}(R = \{\text{vk}_i\}_{i \in [n]})$
3 :	if $[(R^* \not\subseteq R) \vee$
4 :	$(\exists i' \in \mathcal{Q}_C \text{ s.t. } \text{vk}_{i'} \in R^*) \vee$
5 :	$(m^* \in \mathcal{Q}_S)]$: abort
6 :	return $\Pi.\text{Verify}(R^*, \sigma^*, m^*)$

A.3. Ring Signature Schemes

Ring signature schemes were first defined by Rivest, Shamir, and Tauman [53], 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.

Definition 8 (Ring signatures). A ring signature scheme, Π , is a tuple of the following algorithms:

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

First, we say that Π is *complete* if, for any set of keys $\{(\text{sk}_i, \text{vk}_i) \leftarrow \Pi.\text{Gen}(1^\lambda)\}_{i \in [n]}$, $j \in [n]$, message m , the ring $R = \{\text{vk}_i\}_{i \in [n]}$, and $\sigma \leftarrow \Pi.\text{Sign}(\text{sk}_j, m, R)$, then $1 \leftarrow \Pi.\text{Verify}(R, \sigma, m)$. Second, let Anon and Unf be the security games defined in Fig. 6. 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:

- \mathcal{O}_S : takes as input an index i , a message m' , and a ring R' , and returns $\sigma \leftarrow \Pi.\text{Sign}(\text{sk}_i, m', R')$;

- \mathcal{O}_C : takes as input an index i , and returns the randomness used to generate vk_i .

Furthermore, let \mathcal{Q}_S and \mathcal{Q}_C be the sets of queries sent to \mathcal{O}_S and \mathcal{O}_C , respectively.¹²

Instantiations. It is possible to instantiate the required functionality with a specific ring signature scheme that generates signatures under ECDSA private keys that preserve anonymity amongst a “ring” of known ECDSA verification keys (e.g. see [27]).

Appendix B. DCTLS formal description

The three phases (HSP, QP, CP) of a generic three-party TLS (DCTLS) protocol are formally described (in terms of their inputs and outputs) below.

- $(pp, sp_C, sp_S, sp_V) \leftarrow \text{DCTLS.HSP}(1^\lambda)$: 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 (pp, sp_S) learnt by \mathcal{S} are the same as in a standard TLS handshake. The secret parameters learned by \mathcal{C} (sp_C) and \mathcal{V} (sp_V) are shares of the secret parameters learnt by a standard TLS client [22], so that neither party can compute encrypted traffic alone.
- $(r, \hat{q}, \hat{r}) \leftarrow \text{DCTLS.QP}(pp, sp_C, sp_S, sp_V, q)$: 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{q} and \hat{r}) that encrypt the client queries and the server responses. \hat{q} and \hat{r} are vectors containing blocks of the TLS ciphertext encrypting q and r , respectively.
- $b \leftarrow \text{DCTLS.CP}(pp, sp_C, sp_V, q, r, \hat{q}, \hat{r}, (i, j))$: The commitment phase outputs a bit b , where $b = 1$ if \mathcal{C} constructs a valid opening of \hat{q}_i and \hat{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 sp_V to \mathcal{C} , and then \mathcal{C} opens the commitments. Note that a valid opening could be proving in zero-knowledge that \hat{r}_j encrypts a value in a given range, or using 2PC to decrypt the block directly.

Appendix C. DCTLS Standalone Security Model

C.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

12. 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.

\mathcal{S} establish a secure TLS 1.3 channel. To do this, we use the multi-stage key exchange model of [22], which follows the Bellare-Rogaway (BR) framework [7] for establishing authenticated key exchange security based on session key indistinguishability, and builds on the multistage model of Fischlin and Günther [24], [33]. 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*¹³ and *Multi-Stage Security*.¹⁴

To prove the above security properties, we rely on a similar security framework to that used in Oblivious TLS [1, Section 6], that models \mathcal{C} as a multi-party entity known as a *TLS engine*. The differences in comparison with the original model of [22] are: (1) the handshake traffic keys are leaked to the multi-stage adversary *only* when \mathcal{C} is corrupted; (2) the MAC keys used in CF and SF messages are leaked to the multi-stage 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 [1] 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 [1]) 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* [10]. See [1, Definition 2] for more details. We also require (as in [1]) 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 [1]. Note that a potential strengthening of the security model could include the adversary learning \mathcal{S} ’s 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 \mathcal{C} and \mathcal{V} , and show that a corrupted client/verifier can

13. That any two sessions with identical identifiers will agree on the same key eventually.

14. That any tested key is indistinguishable from a random string of the same length.

Algorithm 1 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 2 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** $(HS^w, CHTS^w, SHTS^w, dHS^w, tk_{chs}^w, tk_{shs}^w)$ to $\{\mathcal{C}, \mathcal{V}\}$

only learn details linked to criteria (1)–(5) above. Fig. 5 gives a summary of the 2PC interactions between \mathcal{C} and \mathcal{V} , where Algorithm 1, Algorithm 2, and Algorithm 3 give descriptions of the ideal 2PC functionalities that are used.¹⁵ Our proof is situated in the standard model.

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 “cheat-down” ability for either party 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 [22]. 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 [1], \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 [22], 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.

¹⁵ See Fig. 2 for the full TLS derivation of each value.

Algorithm 3 2PC-DeriveTKApp ideal functionality

Require: (dHS_c) from \mathcal{C} **Require:** (dHS_v) from \mathcal{V} **Ensure:** For $w \in \{c, v\}$: **return** $(tk_{capp}^w, tk_{sapp}^w)$ to $\{\mathcal{C}, \mathcal{V}\}$

Algorithm 4 2PC-RL-Encrypt ideal functionality

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

Theorem 9 (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 [1, Definition 2];
- the underlying security of the TLS 1.3 protocol [22].

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 [1] 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 and Section 4. Our experimental results in Section 7 detail how performance changes depending on the choice of 2PC primitives.

C.2. Query Phase Security

As in the handshake phase, while the server is left untouched, we continue to consider the client and the verifier as one that works together to encrypt and decrypt packets to and from \mathcal{S} . This is a requirement, since 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, the query execution phase considers two ideal functionalities: 2PC-RL-Encrypt (Algorithm 4), and 2PC-RL-Decrypt (Algorithm 5). In 2PC-RL-Encrypt, the

Algorithm 5 2PC-RL-Decrypt ideal functionality

Require: $(\text{tk}_{sapp}^c, (\hat{r}, \tau_{\hat{r}}), AD)$ from \mathcal{C}

Require: (tk_{sapp}^v) from \mathcal{V}

return $\text{AEAD}.\text{Dec}(\text{tk}_{sapp}, \hat{r}, \tau_{\hat{r}}; AD)$ to \mathcal{C}

client and the verifier submit their secret parameters, and the client submits a query (e.g. an HTTP request). The ideal functionality returns an encryption of this query, under a TLS 1.3-compliant AEAD scheme (Section A.2). 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 $\text{AEAD} = \text{AES-GCM}$, assuming the security of the 2PC-AES-GCM protocol (Appendix D). 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 14 and Lemma 15, 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 10. *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 [22].*

C.3. Commitment Phase Security

For the commitment phase of DiStefano, we split the requirement into a number of sub-properties: session privacy (SPriv), ring session authentication (SAuth_n¹), 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 B). 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. 7), we consider a (potentially dishonest) \mathcal{C} that starts by sending a commitment, $c_{\hat{q}, \hat{r}}$, to a specific session, S . In SPriv, the honest client \mathcal{C} constructs and sends a proof, $\sigma_{S, \text{sid}, L}$, that $c_{\hat{q}, \hat{r}}$ is a commitment to a TLS session established with $S \in L$ (L is the set of accepted servers). The adversarial verifier, $\mathcal{A}_{\mathcal{V}}$, succeeds if it identifies the identity of S (it can point which server in the set L that \mathcal{C} is communicating with). In SAuth_n¹, 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 $c_{\hat{q}, \hat{r}}$ corresponds to a session S established with a server $S' \notin L$. 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 $c_{\hat{q}, \hat{r}}$ 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-based TLS certificates for showing that $S \in L$, e.g. [27].

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. 7).

Lemma 11. *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:*

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

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 S^0 , regardless of the bit d . In the second step, the commitment scheme is modified to always commit to traffic exchanged with 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 S^1 , therefore, we will speak only about the S^0 , without loss of generality.

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} that distinguish between the two steps, then there is an adversary \mathcal{B} that can break the Anon security game of Π (Fig. 6). This follows since, in the case when $d = 1$ the only difference is the fact that σ is always computed over the certificate of 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 S^0 . As such, any adversary \mathcal{B} against the computational hiding property of Γ forward their challenge commitment to \mathcal{A}

Figure 7. Security games for the commitment protocol.

SPriv	SAuth _n ¹
1: $L \leftarrow \mathcal{A}_V(1^\lambda, 1^N)$	1: $L \leftarrow \mathcal{V}(1^\lambda, 1^N)$
2: $(\text{sk}_\Pi, \text{vk}_\Pi) \leftarrow \Pi.\text{setup}(1^\lambda)$	2: $(\text{sk}_\Pi, \text{vk}_\Pi) \leftarrow \Pi.\text{setup}(1^\lambda)$
3: $(\text{pp}^0, \text{sp}_C^0, \text{sp}_S^0, \text{sp}_V^0) \leftarrow \text{DCTLS.HSP}(1^\lambda, \mathcal{C}, \mathcal{S}^0, \mathcal{A}_V)$	3: $(\text{pp}, \text{sp}_C, \perp, \text{sp}_V) \leftarrow \text{DCTLS.HSP}(1^\lambda, \mathcal{A}_C, \mathcal{S}, \mathcal{V})$
4: $(\text{pp}^1, \text{sp}_C^1, \text{sp}_S^1, \text{sp}_V^1) \leftarrow \text{DCTLS.HSP}(1^\lambda, \mathcal{C}, \mathcal{S}^1, \mathcal{A}_V)$	4: if $[\mathcal{S} \in L]$: return 0
5: if $[(\mathcal{S}^0 \notin L) \vee (\mathcal{S}^1 \notin L)]$: return 0	5: $\sigma \leftarrow \mathcal{A}_C(\text{pp}, \text{sp}_C, \text{sk}_\Pi, \text{vk}_\Pi, L)$
6: $d \leftarrow \mathbb{S}\{0, 1\}$	6: return $\Pi.\text{Verify}(\text{vk}_\Pi, \text{pp}, \text{sp}_V, \sigma, L)$
7: $q \leftarrow \mathcal{A}_V(L)$	SUnf
8: $(\hat{q}, \hat{r}) \leftarrow \text{DCTLS.QP}(\text{pp}^d, \text{sp}_C^d, \text{sp}_S^d, \text{sp}_V^d, q)$	1: $(\text{pp}, \text{sp}_C, \text{sp}_S, \text{sp}_V) \leftarrow \text{DCTLS.HSP}(1^\lambda, \mathcal{A}_C, \mathcal{S}, \mathcal{V})$
9: $\sigma \leftarrow \Pi.\text{Sign}(\text{sk}_\Pi, \text{pp}^d, \text{sp}_C^d, L)$	2: $q \leftarrow \mathcal{A}_C(1^\lambda, \text{pp}, \text{sp}_C)$
10: $c^d \leftarrow \Gamma.\text{Commit}(\text{sp}_C^d, \hat{q}, \hat{r})$	3: $(\hat{q}, \hat{r}) \leftarrow \text{DCTLS.QP}(\text{pp}, \text{sp}_C, \text{sp}_S, \text{sp}_V, q)$
11: $d' \leftarrow \mathcal{A}_V(\{\text{pp}^d, \text{sp}_V^d\}_{d \in \{0,1\}}, \text{vk}_\Pi, q, c^d, \sigma, L)$	4: $c \leftarrow \mathcal{A}_C(\text{sp}_C, \hat{q}, \hat{r})$
12: if $[d' \stackrel{?}{=} d]$: return 1	5: $q' \leftarrow \mathcal{A}_C(\text{sp}_C, \text{sp}_V, q, \hat{q}, \hat{r}, c)$
13: return 0	6: if $[(\Gamma.\text{Open}(\text{sp}_V, q', c)) \wedge (q' \neq q)]$: return 1
	7: return 0

in the same as before, and win with the same advantage as \mathcal{A} . This completes the proof. \square

Ring 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 SAAuth_n¹ security game (Fig. 7).

Lemma 12. *Let Π be a ring signature scheme that satisfies unforgeability for ECDSA TLS certificates. Then, for all PPT algorithms \mathcal{A} , we have that:*

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

Proof. Suppose that \mathcal{C} could win the SAAuth_n¹ 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}_C outputs as their answer to the Unf security game (Fig. 6). If \mathcal{A}_C creates a valid forgery, then the unforgeability of Π is violated. \square

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. 7).

Lemma 13. *Let Γ be a perfectly binding commitment scheme for a DCTLS scheme. Then, for all PPT algorithms \mathcal{A} , we have that:*

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

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

Algorithm 6 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 $((C_1, \dots, C_n), \tau(A, C, k, IV))$.

Ensure: \mathcal{V} learns (C_1, \dots, C_n) .

if $IV_c \neq IV_v$ **then**

return Error

\triangleright The IVs must match.

end if

Parse m as $m_1 \parallel \dots \parallel m_n$ $\triangleright m_i$ fits AES blocksize

$C = (C_i \leftarrow \text{AES.Enc}(k_c + k_v, IV_c + i) \oplus m_i)_{i \in [n]}$

$\tau_c \leftarrow P_{A \parallel C \parallel \text{len}(A) \parallel \text{len}(C)}(\{h_c^i\})$

$\tau_v \leftarrow P_{A \parallel C \parallel \text{len}(A) \parallel \text{len}(C)}(\{h_v^i\})$

$\tau \leftarrow \tau_c \oplus \tau_v \oplus \text{AES.Enc}(k_c + k_v, IV_c)$

return (C, τ) to \mathcal{C}

return C to \mathcal{V}

Appendix D.

Security of AES-GCM optimisation

Ideal functionalities. We consider the ideal functionalities for encryption and decryption in 2PC-AES-GCM as given in Algorithm 6 and Algorithm 7, respectively. 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.

Security argument. We now argue the security of computing encryptions and decryptions with respect to the ideal functionalities described in Algorithm 6 and

Algorithm 7 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 $A, (C_1, \dots, C_n)$ and τ
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, \dots, C_n) is a valid encryption wrt τ .
if $IV_c \neq IV_v$ **then**
 return Error ▷ The IVs must match.
end if
 $\tau'_c \leftarrow P_{A||C||\text{len}(A)||\text{len}(C)}(\{h_c^i\})$
 $\tau'_v \leftarrow P_{A||C||\text{len}(A)||\text{len}(C)}(\{h_v^i\})$
 $\tau' = \tau'_c \oplus \tau'_v \oplus \text{AES.Enc}(k_c + k_v, IV_c)$
if $\tau' \neq \tau$ **then**
 return Error ▷ Invalid tag
end if
 $m = (C_i \oplus \text{AES.Enc}(k_c + k_v, IV_c + i))_{i \in [n]}$
return m to \mathcal{C}

Algorithm 7. 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 [1], [14], [70], but are not made explicit. We require them when proving that the query phase of DiStefano is secure (Section C.2).

Lemma 14 (Malicious Client). *2PC-AES-GCM is secure in the presence of a malicious adversary that controls \mathcal{C} .*

Proof. Let S be a PPT simulator for the encryption functionality, that simply returns samples C' from the domain of 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 are indistinguishable from this.

Let S_{AES} be a simulator for the ideal 2PC evaluation of AES.Enc , and let S_P be a simulator for the ideal evaluation of P . It first sends m to S_{AES} and learns $C = (C_1, \dots, C_n)$. 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).

Finally, based on the assumption that AES is a pseudorandom permutation, we can construct a final hybrid step, that replaces AES.Enc with a random value in the domain.¹⁶

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 S_{AES} and S_P defined above. \square

16. This only holds if the IV is a nonce, see [70, §B.2].

Lemma 15 (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. \square

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 E.

Appendix E. PageSigner and AES-GCM

In this section, we compare our approach to computing AES-GCM tags to the approach employed by PageSigner [59]. 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 $\tau = (h_c^{m-2} + h_v^{m-2}) \cdot (h_c^1 + h_v^1) \cdot C_1 = (h_c^{m-1} + h_v^{m-1} + h_c^{m-2} \cdot h_v^1 + h_v^{m-2} \cdot h_c^1) \cdot C_1$. 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 efficient than the one that we adopt in Section 5. Intuitively, this is because our approach allows all polynomial evaluation to be done locally, even while both approaches require computing an initial sharing of h and its powers, PageSigner’s approach explicitly requires computing two OT-based multiplications per tagging. Concretely, instantiating these multiplications using the maliciously-secure scheme presented in [19] 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 $500\times$ less bandwidth per tagging operation than the approach employed by PageSigner.

Appendix F. Commitment scheme security

We prove that the commitment scheme Γ from Section 5.1 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 . \square

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 $(\text{tk}_{capp}^v, \text{tk}_{sapp}^v)$. Note that the full keys are defined as $(\text{tk}_{capp}, \text{tk}_{sapp}) = (\text{tk}_{capp}^c \oplus \text{tk}_{capp}^v, \text{tk}_{sapp}^c \oplus \text{tk}_{sapp}^v)$. 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 5 follows for all bounded adversaries. \square

Appendix G. Proofs over encrypted data

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

Revealing a substring. We briefly show how DiStefano can 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, \dots, 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 H. Incompatibility with Garble-then-prove

In this section we highlight why we have not incorporated the recent improvements presented in [64] into our work. We stress that there are no fundamental incompatibilities between DiStefano and the improvements made by [64]; instead, the difficulties are solely implementation driven. We consider producing a tool that allows us to incorporate these changes to be a pressing, but orthogonal, open problem.

An optimised 2PC-HMAC. We briefly recall the secret derivation optimisation presented in [64]. For the purposes of exposition we shall first show how to efficiently compute the HMAC function in a 2PC setting before incorporating this into TLS1.3 secret derivation.

Let $H = \text{SHA256}$. Recall that the $\text{HMAC}_{\text{SHA256}}$ of a variable length message m with a key k is

$$\text{HMAC}_H(k, m) = H((k \oplus \text{opad}) || H((k \oplus \text{ipad}) || m)).$$

From this we can see that naively computing HMAC_H in 2PC is likely to be expensive, as any generic circuit would be required to compute the hash of m in 2PC. Given that the cost of computing such a hash is proportional to the length of m , such a circuit would likely perform poorly on long messages.

This situation was considered in the context of CBC-HMAC by [70]. In this setting, \mathcal{P} and \mathcal{V} wish to compute the CBC-HMAC of a message m that is known entirely by \mathcal{P} under a shared key k . In order to implement this functionality efficiently, the authors of [70] take advantage of the underlying structure of SHA256. Namely, suppose that m_1 and m_2 are two correctly sized blocks. Then

$$\text{SHA256}(m_1, m_2) = f_H(f_H(IV, m_1), m_2)$$

where IV is the initialisation vector and f_H is the compression function of SHA256. If we now return to the HMAC computation, it is clear that the inner-most call is $f_H(IV, k \oplus \text{ipad})$. Given that f_H is assumed to be a one way function, revealing $s_0 = f_H(IV, k \oplus \text{ipad})$ does not reveal anything about k to either party; thus, we can realise HMAC more efficiently by simply revealing s_0 to \mathcal{P} , allowing them to compute the hash of m locally i.e outside of 2PC. This means that the HMAC

computation of an arbitrarily long message only requires a few SHA256 calls in 2PC, rather than potentially many when realising HMAC generically.

This idea was recently adapted to the context of TLS 1.2 secret derivation by the authors of [64]. Briefly, the authors propose revealing the value $s_0 = f_H(IV_0, pms \oplus \text{ipad})$ to both parties, allowing some of the f_H calls to be realised locally. In addition, the authors of [64] also propose re-using previously garbled values across multiple circuits, allowing even fewer f_H calls to be carried out in 2PC. Concretely, this optimisation reduces the number of needed f_H calls in 2PC from 18 to only 6.

Implementation difficulties. We now highlight why this approach seems difficult to realise in DiStefano. At a high-level, the main issue is that our current garbled circuit library (`emp`) does not support either accepting new input or outputting partial values as the circuit runs. Indeed, if `emp` were to support this feature, then realising the optimisation presented would be easy. However, the fact that `emp` does not support this feature means that we would have to realise this optimisation by chaining multiple circuits together. In this model, each circuit computes a sub-portion of the secret derivation procedure and outputs some intermediate results to each party.

On the one hand, this approach would allow each party to locally compute some of the calls to f_H , yielding the claimed speed-ups. However, this would require each circuit to either recompute any shared input, and to validate that the same value of s_0 is used across multiple circuits. In the case of the former, we would need to recompute `pms` in each individual circuit, and we would also need to check that the same value of s_0 is supplied by both parties. Put simply, this decomposition would reduce some of the potential speed-ups from [64]. Moreover, decomposing secret derivation into multiple circuits would allow a malicious party to alter their inputs at each stage, potentially causing selective failure. Whilst we stress that we cannot see an obvious way to use this to expose a vulnerability in the security of DiStefano, it would contradict elements of the security model that we consider.

Nonetheless, none of these reasons invalidate the approach taken in [64], and we believe that a similar idea can be applied even with the current version of `emp`. For example, notice that our secret derivation circuit recomputes the compression function $f_H(\text{HS} \oplus \text{ipad})$ a total of three times during secret derivation; re-using this value inside the same circuit would be more efficient than our current approach. A similar approach can also be applied to the derivation of traffic secrets. However, we have not implemented this approach, as we believe that it is unlikely to be competitive with [64] in practice. Put differently, we believe that the completion of a tool that allows for the [64] optimisation to be realised securely inside DiStefano should be the priority for future work.

Appendix I.

Potential attacks on Janus and DECO’s TLS 1.3 variant

In this section we present an attack on an *interpretation* of the TLS 1.3 variant of DECO and the recently presented Janus [43] TLS attestation mechanism. Notably, this attack allows either a malicious client to produce false attestations, or for a malicious proxy to fully decrypt all traffic. We stress that our interpretation of DECO’s TLS 1.3 variant may be different from exactly what the authors intended, due to lack of specificity, and thus the attack presented may not be directly applicable. However, the attack presented here applies to any TLS attestation protocol that does not carefully handle the sharing (and disclosure) of traffic secrets.

Background: the `SF` message. The attack presented in this section exploits a flaw in how both Janus and DECO’s TLS 1.3 variant handle session authentication between the client and the verifier. In order to explain this flaw, we briefly recall the TLS 1.3 server authentication mechanism used in TLS 1.3 (see [22] for a more thorough explanation).

First, assume that \mathcal{S} and \mathcal{C} are establishing a TLS 1.3 session; and that \mathcal{S} and \mathcal{C} have exchanged both the hello messages (`ClientHello` and `CH`) and derived authentic copies of the handshake secrets `SHTS` and `CHTS`. At this stage, \mathcal{S} authenticates itself to \mathcal{C} as follows:

- 1) To ensure authenticity, \mathcal{S} sends its certificate `SCRT` to \mathcal{C} , followed by a signature on the hash of the session transcript (`SCV`). Upon receipt of these messages, \mathcal{C} checks the certificate and uses the corresponding public key to verify `SCV`. Notably, \mathcal{C} is able to check `SCV` because \mathcal{C} has a full view of the transcript.
- 2) To ensure integrity, \mathcal{S} uses `SHTS` to derive the *finished key* fk_s . \mathcal{S} then computes the *server finished* (`SF`) message, by computing a HMAC on the hash of the session transcript using fk_s . \mathcal{S} then sends `SF` to \mathcal{C} , who locally validates `SF` by deriving their own fk_s . Again, \mathcal{C} is only able to check the `SF` because they have a full and complete view of the transcript.

In the regular (i.e two-party) variant of TLS 1.3 this approach has been shown to provide very high security guarantees: the pioneering works of [21], [22] have shown that the TLS 1.3 handshake protocol establishes session keys with strong security properties under standard cryptographic assumptions. Interestingly, the TLS 1.3 handshake protocol actually achieves a stronger level of security than may be fully necessary in some settings. For example, it has been argued that the `SF` message alone authenticates the transcript [21], as the `SF` message is an authenticated HMAC of the transcript sent so far. Put differently, the `SCV` message does not provide any additional integrity guarantees for the derived keying material compared to just checking the `SF` message.

TLS 1.3 in DECO and potential avenues for attack. The authors of [70] initially proposed using a variant of the above idea to accelerate certificate checking in DECO’s TLS 1.3 variant. Namely, the authors of DECO suggest, as the traffic keys are independent of the handshake keys, that the \mathcal{C} and \mathcal{V} can simply derive all secrets in a single circuit, with \mathcal{C} alone learning all handshake keys for the purpose of certificate checking. Whilst it is unclear that this approach will work¹⁷, we note that the description given in DECO does not specify whether \mathcal{V} *ever* learns the SF or SCV message in the TLS 1.3 version of DECO, or if \mathcal{C} alone has access to this information. We argue that this approach leads to a very straightforward attack. Namely, suppose that \mathcal{C} is malicious and wishes to falsely attest of data using a server \mathcal{S} . In this attack, \mathcal{C} begins a three-party handshake with \mathcal{V} , and receives \mathcal{V} ’s keyshare for the three-party handshake. Then, rather than contacting \mathcal{S} and carrying out the usual three-party handshake, \mathcal{C} simply replies to \mathcal{V} with its own keyshare. Intuitively, this step is the same as establishing a TLS 1.3 session between \mathcal{C} and \mathcal{V} without the involvement of \mathcal{S} . This step produces the following outcomes:

- 1) \mathcal{V} , upon receipt of \mathcal{C} ’s keyshare, is unable to determine whether it belongs to \mathcal{S} or \mathcal{C} , as it is unauthenticated at this stage. Thus, \mathcal{V} must assume that the keyshare is a legitimate value from \mathcal{S} .
- 2) On the other hand, \mathcal{C} can now locally derive *all* handshake and traffic secrets for the session, as \mathcal{C} knows the entirety of the shared handshake secret. Thus, \mathcal{C} can systematically deceive \mathcal{V} in an undetectable manner. Moreover, as \mathcal{V} never learns the SF message (or even the SCV message), \mathcal{V} is unable to check if the handshake transcript was valid. As such, if \mathcal{C} indicates that the checks passed, then \mathcal{V} continues as if the transcript was authentic.

At this stage, \mathcal{C} can forge any TLS traffic between itself and \mathcal{S} , without \mathcal{V} being able to detect the forgeries, leading to false attestations. Moreover, this attack persists even if \mathcal{V} explicitly checks the SF message of the transcript: as the transcript between \mathcal{V} and \mathcal{C} is a valid TLS 1.3 transcript, the secret derivation process will produce valid secrets, and checks on SF will pass.

Formally speaking, this attack arises from the fact that, although \mathcal{V} and \mathcal{C} are treated as a singular entity from the perspective of \mathcal{S} , they are in fact very different entities. Practically speaking, the main issue is that \mathcal{C} obtains a valid copy of the transcript, whereas \mathcal{V} does not. Moreover, ensuring that \mathcal{V} receives any meaningful authenticity guarantees in the face of a malicious adversary whilst respecting privacy is somewhat difficult.

17. Given that deriving the traffic keys requires SF to be incorporated into the transcript hash, it seems impossible that the application and traffic secrets could be derived in one step.

Indeed, a trivial solution to the aforementioned attack would be to simply require that \mathcal{V} learns the identity of \mathcal{S} and the SCV value, allowing \mathcal{V} to authenticate \mathcal{S} .

Attack vectors on the Janus protocol. We argue that the recently proposed Janus [43] TLS attestation protocol is potentially vulnerable to an attack with a similar root cause, albeit with a different mechanism.

Briefly, the Janus protocol is a hybrid between DiStefano and DECO’s proxy mode. In its presented configuration, Janus treats the verifier as a TLS 1.3 proxy that forwards (and records) all traffic between \mathcal{C} and \mathcal{S} . Notably, the Janus protocol differs from DiStefano by using a set of semi-honest garbled circuits for all secret derivation, other than AES-GCM tagging and verification. In this model, the adversary follows an agreed-upon protocol without deviation. Instead, the primary aim of the adversary is to learn the secret inputs of the honest party. Whilst this change is primarily made for efficiency, the authors of Janus also claim that the authenticity guarantees given by the SCV and SF messages allows for this change to be made without any loss of security.

In order to highlight a potential attack, we now briefly describe the differences in secret derivation between Janus and DiStefano. We note that the protocols are essentially identical up until the secret derivation procedure, and thus we omit these details.

- 1) \mathcal{C} and \mathcal{V} invoke a semi-honest garbled circuit protocol, that outputs the SHTS value to \mathcal{C} . \mathcal{C} uses an authentic copy of the transcript to check the SF and SCV messages. If the checks fail, then \mathcal{C} aborts. Otherwise, \mathcal{C} reveals SHTS to \mathcal{V} , and \mathcal{V} repeats the same checks.
- 2) \mathcal{C} and \mathcal{V} repeat similarly for the CHTS circuit.
- 3) Finally, \mathcal{C} and \mathcal{V} derive the traffic secrets using a series semi-honest garbled circuits. We note that, according to Figure 4 of [43], the dHS secret appears to be derived *after* the SHTS value has been checked. As we shall soon explain, this is problematic when only semi-honest garbled circuits are used.

We now present a potential attack using a malicious adversary that is undetectable using semi-honest garbled circuits. First, assume that \mathcal{V} is a malicious verifier that wishes to compromise the confidentiality of \mathcal{C} . In order to achieve this, \mathcal{V} generates a public key K_v and establishes a TLS session with \mathcal{S} , recording the entire transcript T and deriving all traffic secrets. Note that from the perspective of \mathcal{S} , \mathcal{V} is behaving honestly, and thus \mathcal{S} cannot mitigate this attack. Then, when \mathcal{C} begins the Janus protocol, \mathcal{V} replays the transcript T to \mathcal{C} and maliciously garbles the SHTS derivation circuit such that the SHTS from T is revealed. We stress that this step requires \mathcal{V} to be a malicious actor, as the circuit would otherwise output a different SHTS due to the keyshare input by \mathcal{C} . At this stage, \mathcal{C} will check the SF and SCV messages and conclude that they are valid. \mathcal{V} , hence, has succeeded in their attack: \mathcal{V} can simply ignore

the request for validation using the CHTS, and, as they also know all other traffic secrets, \mathcal{V} can undetectably decrypt or modify any messages exchanged between \mathcal{S} and \mathcal{C} . Moreover, even if \mathcal{V} is unable to modify the SHTS check, it seems clear that modifying the output of the dHS circuit to a known value is enough to remove any security guarantees that Janus offers; indeed, setting the dHS to some known value affords the exact same powers to \mathcal{V} in a manner that is undetectable to \mathcal{C} .

Why will these attacks work?. Intuitively, all the aforementioned attacks rely on the mismatch between the transcript seen by \mathcal{V} and the transcript seen by \mathcal{C} . On the one hand, omitting certain information from the view of \mathcal{V} allows \mathcal{C} to act in an arbitrarily powerful fashion, providing no guarantees at all to \mathcal{C} . Yet, we consider the attack on Janus to be equally as serious; here, the session guarantees expected by \mathcal{C} simply fade away in an undetectable manner, even though \mathcal{C} may have access to a full view of the transcript. In both cases, the attacks exist simply because there is no concrete binding between the transcript shown to \mathcal{C} by \mathcal{V} and the actual, underlying session that is being carried out.

Mitigating these attacks appears fairly straightforward. DiStefano avoids the attack on DECO's TLS 1.3 variant by simply requiring that \mathcal{C} provides commitments to session traffic as part of the authentication process. This, coupled with a ring signature of the validity of the SCV value, is enough to ensure that \mathcal{C} cannot fool \mathcal{V} without a collaborating \mathcal{S} . Moreover, the use of maliciously-secure garbled circuit protocols is enough to prevent \mathcal{V} from modifying the output of the SHTS derivation in a predictable or useful fashion. Finally, we mention that a potential defence against the attack described on Janus would be to simply require that \mathcal{V} proves that the value it sends in the client handshake somehow involves the share provided by \mathcal{C} . This could be achieved by revealing both $T = K_v + K_c$ and K_v to \mathcal{C} , which would allow \mathcal{C} to ensure that its view of the transcript is authentic relative to the current session. However, we caution against the belief that this safeguard alone is sufficient, as it is clear that any attack against the derivation of dHS is sufficient to allow a malicious adversary to break the security of the Janus protocol. We leave clarifying the security properties of a DiStefano-like protocol that uses semi-honest primitives for future work.

Appendix J. The TLS 1.3 handshake

An overview of the TLS 1.3 handshake is given in Fig. 8, our notation is based on the notation defined in [22], which we provide in Table 7 and Table 6 for completeness.

Figure 8. TLS 1.3 handshake with certificate-based authentication. Shorthands correspond to [22]. Purple represents messages sent or calculated by \mathcal{C} , pink by \mathcal{S} . Messages with an asterisk (*) are optional, and those within braces ({}) are encrypted.

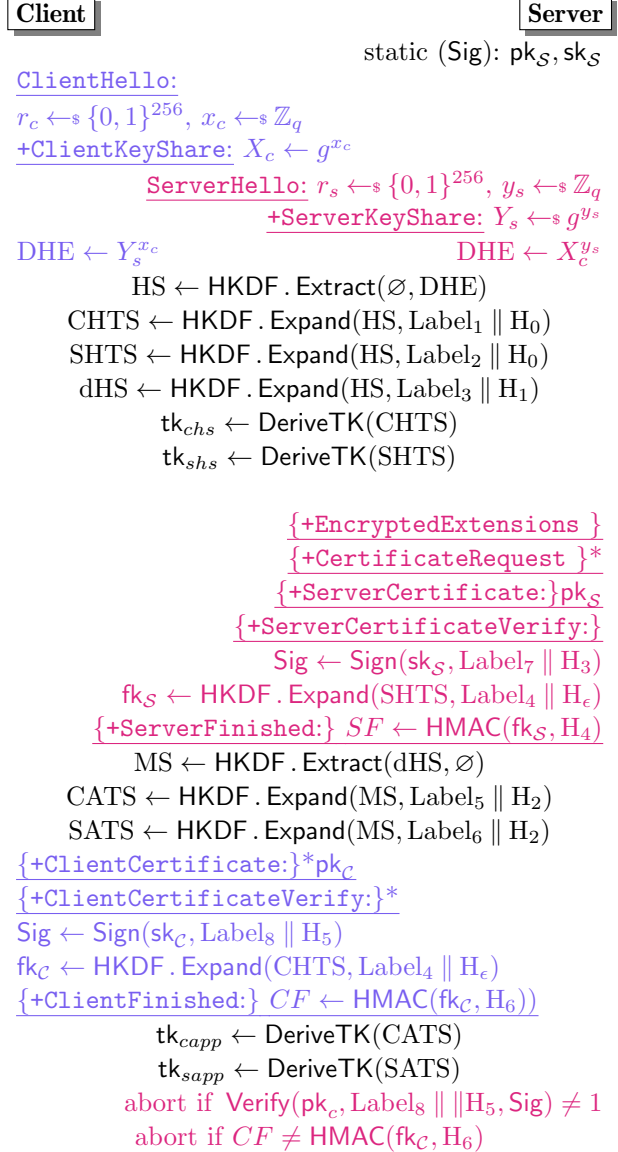


Table 6. TLS 1.3 HANDSHAKE AND TRAFFIC SECRETS.

Secret	Context Input	Label
CHTS	$H_0 = \mathcal{H}(\text{CH}, \dots, \text{SH})$	Label ₁ = "c hs traffic"
SHTS	$H_0 = \mathcal{H}(\text{CH}, \dots, \text{SH})$	Label ₂ = "s hs traffic"
dHS	$H_1 = \mathcal{H}(\dots)$	Label ₃ = "derived"
$\text{fk}_{\mathcal{S}}$	$H_{\epsilon} = \mathcal{H}(\emptyset)$	Label ₄ = "finished"
$\text{fk}_{\mathcal{C}}$	$H_{\epsilon} = \mathcal{H}(\emptyset)$	Label ₄ = "finished"
CATS	$H_2 = \mathcal{H}(\text{CH}, \dots, \text{SF})$	Label ₅ = "c ap traffic"
SATS	$H_2 = \mathcal{H}(\text{CH}, \dots, \text{SF})$	Label ₆ = "s ap traffic"

Table 7. TLS 1.3 AUTH MESSAGES AND ASSOCIATED HASHES,
 WHERE Label₇ is “TLS 1.3, SERVER CERTIFICATEVERIFY” AND
 Label₈ is “TLS 1.3, CLIENT CERTIFICATEVERIFY”.

Auth message	Context Input	Label
SCV	$H_3 = \mathcal{H}(\text{CH}, \dots, \text{SCRT})$	Label ₇
SF	$H_4 = \mathcal{H}(\text{CH}, \dots, \text{SCV})$	
CCV	$H_5 = \mathcal{H}(\text{CH}, \dots, \text{CCRT})$	Label ₈
CF	$H_6 = \mathcal{H}(\text{CH}, \dots, \text{CCV})$	