N-for-1-Auth: N-wise Decentralized Authentication via One Authentication

Ryan Deng
Weikeng Chen
Raluca Ada Popa

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-240
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-240.html

December 1, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to thank my mentor Weikeng Chen and my advisor Professor Raluca Ada Popa for providing guidance throughout the project. I would also like to thank Professor Alessandro Chiesa and the entire RISE Security Group for their valuable feedback on this project.

N-for-1-Auth: N-wise Decentralized Authentication via One Authentication

Ryan Deng
rdeng2614@berkeley.edu
UC Berkeley

***Abstract*—Decentralizing trust is a fundamental principle in the design of end-to-end encryption and cryptocurrency systems. A common issue in these applications is that users possess critical secrets. If these secrets are lost, users can lose precious data or assets. This issue remains a pain point in the adoption of these systems. Existing approaches such as backing up user secrets through a centralized service or distributing them across *N* mutually distrusting servers to preserve decentralized trust are either introducing a central point of attack or face usability issues by requiring users to authenticate *N* times, once to each of the *N* servers.**

We present N-for-1-Auth, a system that preserves distributed trust by enabling a user to authenticate to *N* servers independently, with the work of only one authentication, thereby offering the same user experience as in a typical centralized system.

I. Introduction

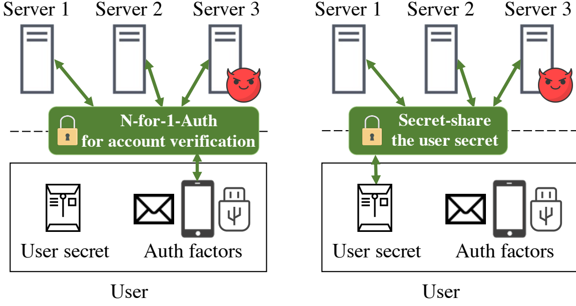
Decentralizing trust is a fundamental principle in designing modern security applications. For example, there is a proliferation of end-to-end encrypted systems and cryptocurrencies, which aim to remove a central point of trust [1–11]. In these applications, users find themselves owning critical secrets, such as the secret keys to decrypt end-to-end encrypted data or the secret keys to spend digital assets. If these secrets are lost, the user permanently loses access to his/her precious data or assets.

To explain the problem that N-for-1-Auth addresses, let us take the example of Alice, a user of an end-to-end encryption application denoted as the "E2EE App" (or similarly, a cryptocurrency system). For such an E2EE App, Alice installs an E2EE App Client on her device, such as her cell phone. The client holds her secret key to decrypt her data. For the sake of usability and adoption, Alice should not have to deal with backing up the key herself. We are concerned with the situation when Alice loses her cell phone. Though she can get a new SIM card by contacting the provider, she loses the secret keys stored on the phone. With WhatsApp [4] and Line [8], end-to-end encrypted chat applications, Alice can use centralized services such as Google Drive and iCloud to backup her chat history. However, such a strategy jeopardizes the end-to-end encryption guarantees of these systems because users' chats become accessible to services that are central points of attack. This is further reaffirmed by Telegram's CEO Pavel Durov who said in a blog post: *"(Centralized backup) invalidates end-to-end encryption for 99% of private conversations"*. To preserve decentralized trust, many companies [7, 12–16] and academic works [17–20] have proposed to secret-share the user's secrets across *N* servers, so that compromising some of the servers does not reveal her secrets.

However, a significant issue with this approach is the burden of authentication. After Alice loses her cell phone with all her secrets for the E2EE App, she can only authenticate with other factors, such as email, short message services (SMS), universal second-factor (U2F), and security questions. How does Alice authenticate to the *N* servers to retrieve her secret? If Alice authenticates to only one server and the other servers trust this server, the first server now becomes a central point of attack. To avoid centralized trust, as the *N* servers cannot trust each other, Alice has to authenticate to each server separately. For email verification, Alice has to perform *N* times the work—reading *N* emails. To avoid a central point of attack, the E2EE App should require multiple factors, which further multiplies Alice's effort.

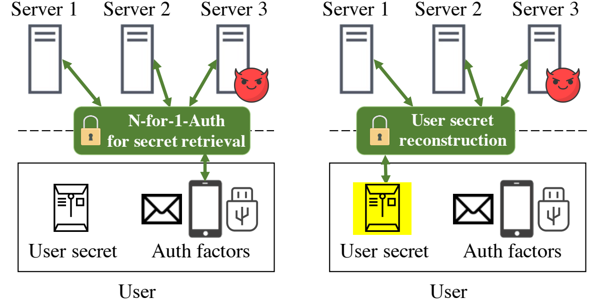
One might think that doing *N* times the work, albeit undesirable for the user, is acceptable in catastrophic situations such as losing one's devices. The issue here is that Alice has to perform this work not only when she is recovering her secrets, but also *when she is joining the system,* because her key's secret shares must be registered with the *N* servers using the multiple factors of authentication, and those servers must check that Alice indeed is the person controlling those factors. Even for $N = 2$ in which there is only one additional email and text message, it is a completely different user experience that adds friction to a space already plagued by usability issues (e.g., "Why Johnny can't encrypt?" [21, 22]). Many academic works [23, 24] reaffirm the importance of the consistency of user experience and minimizing user effort for better adoption.

Therefore, this initial bar of entry is a deterrent against widespread adoption. To validate that this is an important and recurring problem, we presented N-for-1-Auth to prominent companies in the end-to-end encryption and



(a) Servers authenticate the user through authentication factors.
(b) The user secret-shares the user secret among the servers.

Fig. 1: Enrollment workflow.



(a) Servers authenticate the user via authentication factors.
(b) The user reconstructs the secret from shares.

Fig. 2: Authentication workflow.

cryptocurrency custody spaces, who supported our thesis. We summarize their feedback in §I-B.

A few strawman designs seem to address this burden for Alice but are actually unviable. One strawman is to build a client app that automatically performs the N authentications for Alice. In the case of email/SMS authentication, the client app parses the emails or text messages Alice receives from the N servers. However, this either requires the client app to have intrusive permissions (e.g., reading Alice's email) that can affect the security of other applications Alice uses or requires very specific APIs available on the email/SMS server side (e.g., Gmail offering such APIs), which do not exist today for major providers and we find unreasonable. Another strawman [17–20, 25] is to have Alice possess or remember a master secret and then authenticate to each of the N servers by deriving a unique secret to each server, thereby avoiding the issue of having to do the work surrounding email/SMS authentication. However, Alice has to then safeguard this secret, as losing it could lead to an attacker impersonating her to the N servers. In this case, we return to the original problem of Alice needing a reliable way to store this authentication secret.

A. N-for-1-Auth

We present N-for-1-Auth, which alleviates this burden by enabling a user to authenticate to N *servers independently* by doing only the work of authenticating with *one*, as illustrated in Figs. 1 and 2. This matches the usual experience of authenticating to an application with centralized trust.

N-for-1-Auth supports many authentication factors that users are accustomed to, including email, SMS, U2F, and security questions, as discussed in §IV. Specifically, N-for-1-Auth requires no changes to the protocols of these forms of authentication.

N-for-1-Auth offers the same security properties as the underlying authentication protocols even in the presence of a malicious adversary that can compromise up to $N-1$ of the N servers. N-for-1-Auth additionally offers relevant privacy properties for the users. Users of authentication factors may want to hide their email address, phone number, and security questions from the authentication servers. This is difficult to achieve in traditional (centralized) authentication. We discuss the concrete privacy properties of N-for-1-Auth for each factor in §IV.

N-for-1-Auth provides an efficient implementation for several factors and is 8× faster than a naive implementation without our application-specific optimizations. For example, when $N = 5$, our email authentication protocol takes 1.38 seconds to perform the TLS handshake and takes an additional 0.43 seconds to send the email payload, which is efficient enough to avoid a TLS timeout and successfully communicate with an unmodified TLS email server.

B. The case for N-for-1-Auth

We presented N-for-1-Auth to top executives of several prominent companies (which we are not ready to disclose at this moment) in the end-to-end encryption or cryptocurrency custody space. We received valuable feedback from them, which we used to improve N-for-1-Auth.

- Many companies mentioned the need for fault tolerance, which can be addressed in two steps. First, N-for-1-Auth can incorporate threshold secret-sharing, as discussed in §VIII, which enables a subset of servers to recover the secret. Second, each server can set up backup machines within their trust domain/cloud.
- Some companies mentioned the need for more authentication factors, e.g., TOTP (time-based one-time passcodes) and SSO (single sign-on) [26, 27], which can be integrated into N-for-1-Auth in similar ways—TOTP follows a similar format to security questions in which the client stores the TOTP key, and SSO can be inte-

grated using N-for-1-Auth's TLS-in-SMPC protocol.
- Some companies mentioned the need to hide user contact information from the other servers, which we address in §IV.

Overall, we hope that N-for-1-Auth can provide practical value to this space of distributed trust.

C. Summary of techniques

We now describe how N-for-1-Auth maintains the same user experience while decentralizing trust.

One passcode, N servers. Consider email authentication. How do N servers coordinate to send one email with an authentication passcode that they agree on?

First, no server should know the passcode, otherwise this server can impersonate the user. We want to ensure that N-for-1-Auth provides the same security as the traditional solution in which N servers each send a different email passcode to the user.

N-for-1-Auth's solution is to have the N servers jointly generate a random passcode for email authentication in secure multiparty computation (SMPC) [28–31]. In this way, none of them learn the passcode. However, an immediate question arises: how do they send this jointly generated passcode to the user's email address securely?

In the traditional workflow for sending email, one party connects to the user's email service provider (e.g., Gmail) via TLS. The TLS server endpoint is at the email service provider, and the TLS client endpoint is at the sender's email gateway. The mismatch in our setting is that the sender now comprises N servers who must not see the contents of the email.

Sending TLS-encrypted traffic from SMPC. Our insight is that using a new primitive—TLS-in-SMPC—with which the N servers can jointly act as a single TLS client endpoint to communicate with the user's email server over a TLS connection, as Fig. 3 shows. When connecting with the user's email server, the N servers run a maliciously secure SMPC that takes the place of a traditional TLS client. What comes out of the SMPC is TLS-encrypted traffic, which one of the servers simply forwards to the user's email provider. N-for-1-Auth's TLS-in-SMPC protocol stores TLS secrets inside SMPC such that none of the servers can break the security guarantees of TLS. Therefore, the server that forwards the traffic can be arbitrary and does not affect security.

The user's email server, which is unmodified and runs an unmodified version of the TLS server protocol, then decrypts the traffic produced by the TLS-in-SMPC protocol and receives the email. The email is then seen by the user, who can enter the passcode into a client app to authenticate to the N servers, thereby completing N-for-1-Auth's email authentication.

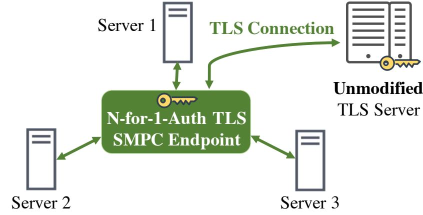


Fig. 3: TLS-in-SMPC's system architecture.

Support for different authentication factors. Beyond email, N-for-1-Auth supports SMS, U2F, and security questions. In addition, due to the generality of N-for-1-Auth's TLS-in-SMPC construct, N-for-1-Auth can also support any web-based authentication such as OAuth [27], which we discuss how to incorporate in §VII. However, we focus on the aforementioned factors in this work. Each factor has its unique challenges for N-for-1-Auth, particularly in ensuring N-for-1-Auth does not reduce the security of these factors. More specifically, replay attacks are a common threat to authentication protocols. In our system, when a malicious server receives a response from the user, this server may attempt to use the response to impersonate the user and authenticate with the other servers. We systematically discuss how to defend against such replay attacks in §IV.

End-to-end implementation for TLS-in-SMPC. TLS is an intricate protocol that involves many cryptographic operations. If we run the TLS endpoint using a maliciously secure SMPC library off the shelf, our experiments in §VI-E show that it would be at least 8× more expensive than our protocol. We designed our TLS-in-SMPC protocol and optimized its efficiency with a number of insights based on the TLS protocol itself, and integrated it with the `wolfSSL` library.

II. System overview

In this section we describe the system at a high level.

System setup. 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. N-for-1-Auth recommends users to use multiple second factors when authenticating to avoid having any single factor act as a central point of attack. The user holds a secret that they wish to distribute among the N servers. Based on our discussion with companies in §I-B, these companies wished that one of the N servers be the actual server of the application, and the other $N-1$ servers be other helper servers. This is natural because the app server is providing the service, and importantly, the app server is

reassured that if it behaves well, the $N-1$ helper servers cannot affect security. Each user can download a *stateless* client application or use a web client to participate in these protocols. This minimalist client app *does not retain secrets* or demand intrusive permissions to data in other applications such as a user's emails or text messages; it simply serves as an interface between the user and the servers. We place such limitations on the client app since we assume the device hosting the app can be lost or stolen, and we want to hide the user's sensitive data from our client app.

Workflow. The system consists of two phases:

- *Enrollment (Fig. 1).* When the user wants to store a secret on the servers, the user provides the servers with a number of authentication factors, which the servers verify using N-for-1-Auth's authentication protocols described in §IV. Then, after authenticating with these factors, the client secret-shares the secret and distributes the shares across the servers.
- *Authentication (Fig. 2).* The user runs the N-for-1-Auth protocols for the authentication factors. Once the user is authenticated, the N servers can perform computation over the secret for the user, which is application-specific, as we describe in §V.

Though in use cases such as key recovery, the authentication phase only occurs in catastrophic situations, users must enroll their factors when joining the system, which typically requires N times the effort and is a different user experience from enrolling to a centralized system.

N-for-1 Authentications. We describe N-for-1-Auth's authentication protocols for several factors in §IV.

- *Email:* The N servers jointly send *one* email to the user's email address with a passcode. During authentication, the servers expect the user to enter this passcode.
- *SMS:* The N servers jointly send *one* message to the user's phone number with a passcode. During authentication, the servers expect the user to enter this passcode.
- *U2F:* The N servers jointly initiate *one* request to a U2F device. During authentication, the servers expect a signature, signed by the U2F device, over this request.
- *Security questions:* The user initially provides a series of questions and answers to the servers. During authentication, the servers ask the user these questions and expect answers consistent with those that are set initially. Passwords are a special case of security questions and can also be verified using this protocol.

Applications. We describe how N-for-1-Auth supports two common applications in §V , but N-for-1-Auth can also be used in other systems with decentralized trust.

- *Key recovery:* The user can backup critical secrets by secret-sharing them among the N servers. Upon successful authentication, the user can then retrieve these secrets from the servers.
- *Digital signatures:* The user can backup a signing key (e.g., secret key in Bitcoin) by secret-sharing it among the N servers. Upon successful authentication, the servers can sign a signature over a message the user provides, such as a Bitcoin transaction.

Example. We illustrate how to use N-for-1-Auth with a simple example. Alice enrolls into N-for-1-Auth through the client app. She provides three authentication factors: her email address, her phone number, and her U2F token. The client app then contacts the N servers and enrolls these factors. The N servers then send *one* email and *one* text message, both containing a random passcode, and *one* request to Alice's U2F device. Alice then enters the passcodes on the client app and confirms on her U2F device. When all the N servers have verified Alice, the client app then secret-shares the key with the servers, and the servers store the shares. Alice performs the same authentication when she wants to recover the secrets.

A. Threat model

N-for-1-Auth's threat model, illustrated in Fig. 4, is as follows. Up to $N-1$ of the N servers can be ***malicious*** and collude with some users, but at least one server is ***honest*** and does not collude with any other parties. The honest users do not know which server is honest. The malicious servers may deviate from the protocol in arbitrary ways, including impersonating the honest user, as Fig. 4 shows. For ease of presentation, we assume that servers do not perform denial-of-service (DoS) attacks, but we discuss how to handle these attacks in §VIII.

Users can also be ***malicious*** and collude with malicious servers. Malicious users may, for example, try to authenticate as an honest user. We assume that an honest user uses an uncompromised client app, but a malicious user may use a modified one. The client app does not carry any secrets, but it must be obtained from a trusted source, as in the case of the software clients in E2EE or cryptocurrency systems. The client app either has hard-coded the TLS certificates of the N servers, or obtains them from a trusted certificate authority or a transparency ledger [32, 33]. This enables clients and servers to connect to one another securely using the TLS protocol.

Security properties. N-for-1-Auth is built on top of existing authentication factors and maintains the same security properties that the existing factors provide under this threat model. This assertion rests on the security

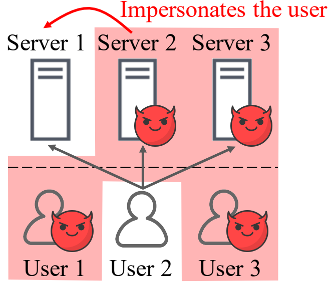


Fig. 4: N-for-1-Auth's threat model. The **red** area indicates a group of malicious parties who collude with one another.

of N-for-1-Auth's TLS-in-SMPC protocol. Formally, we define in App. A an ideal functionality $\mathcal{F}_{\mathsf{TLS}}$ that models the TLS client software that communicates with a trusted, unmodified TLS server. Based on $\mathcal{F}_{\mathsf{TLS}}$, we define the security of our TLS-in-SMPC protocol using a standard definition for (standalone) malicious security [34]:

Definition II.1 (Security of TLS-in-SMPC)**.** *A protocol* Π *is said to* securely compute $\mathcal{F}_{\mathsf{TLS}}$ *in the presence of static malicious adversaries that compromise up to $N-1$ of the N servers, if, for every non-uniform probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ in the real world, there exists a non-uniform PPT adversary $\mathcal{S}$ in the ideal world, such that for every $I \subseteq \{1, 2, ..., N\}$,*

$$\{IDEAL_{\mathcal{F}_{\mathsf{TLS}}, I, \mathcal{S}(z)}(\vec{x})\}_{\vec{x},z} \stackrel{c}{\approx} \{REAL_{\Pi, I, \mathcal{A}(z)}(\vec{x})\}_{\vec{x},z}$$

where $\vec{x}$ denotes all parties' input, z denotes an auxiliary input for the adversary $\mathcal{A}$, $IDEAL_{\mathcal{F}_{\mathsf{TLS}}, I, \mathcal{S}(z)}(\vec{x})$ denotes the joint output of $\mathcal{S}$ and the honest parties, and $\mathrm{REAL}_{\Pi, I, \mathcal{A}}(\vec{x})$ *denotes the joint output of $\mathcal{A}$ and the honest parties.*

We present our TLS-in-SMPC protocol in §III, and we prove that it securely realizes $\mathcal{F}_{\mathsf{TLS}}$ in App. A.

III. TLS IN SMPC

In N-for-1-Auth's email/SMS authentication protocols, the N servers need to establish a secure TLS connection with an *unmodified* TLS server. In this section, we describe TLS-in-SMPC, a protocol that achieves this goal.

Background: secure multiparty computation. The goal of secure multiparty computation (SMPC) [28–31] is to enable N parties to collaboratively compute a function $f(x_1, x_2, ..., x_N)$, in which the i-th party has private input x_i, without revealing x_i to the other parties.

SMPC protocols are implemented using either arithmetic circuits such as in SPDZ [35] or boolean circuits such as in AG-MPC [36, 37]. These protocols consist of an offline phase and an online phase. The offline phase is independent of the function's input and can be run beforehand to reduce the online phase latency.

A. Overview

In TLS-in-SMPC, N servers jointly participate in a TLS connection with an unmodified TLS server. Since these N servers do not trust each other, any one of them must not be able to decrypt the traffic sent over the TLS connection. Therefore, the insight is for these N servers to jointly create a TLS client endpoint within SMPC that can communicate with the TLS server over a TLS connection.

As Fig. 3 shows, the N servers run a TLS client within SMPC, which establishes a TLS connection with the unmodified TLS server. The TLS session keys are only known by the TLS server and the TLS client within SMPC. Hence, the N servers must work together to participate in this TLS connection.

All packets are forwarded between the SMPC and the unmodified TLS server via one of the servers. The specific server that forwards the packets does not affect security since none of the servers know the TLS session keys. That is, none of the servers can decrypt the packets being forwarded or inject valid packets into the TLS connection. The TLS-in-SMPC protocol consists of two phases:

- **TLS connection establishment:** The N servers jointly generate the client-side secret for Diffie-Hellman key exchange. After receiving the server-side secret, they derive the TLS session keys inside SMPC.
- **Data exchange:** The N servers, within SMPC, use the session keys to encrypt or decrypt a message.

Challenge. A straightforward implementation of the TLS-in-SMPC protocol is to use any malicious SMPC protocol off the shelf. If this protocol does not support offline precomputation or is ill-suited for the type of computation being performed, the online latency may cause a timeout that terminates the connection. For example, we found that Gmail's SMTP servers have a TLS handshake timeout of 10 s. Our implementation is efficient enough to consistently meet this timeout, as discussed in §VI.

B. TLS connection establishment

We discuss how N-for-1-Auth's TLS-in-SMPC protocol handles key exchange and how it differs from traditional Diffie-Hellman key exchange. We do not discuss RSA key exchange as it is not supported in TLS 1.3.

Background: Diffie-Hellman key exchange [38]. Let G be the generator of a suitable elliptic curve of prime order p. The key exchange consists of three steps:

1) In the `ClientHello` message, the TLS client samples $\alpha \leftarrow\$ \mathbb{Z}_p^+$ and sends $\alpha \cdot G$ to the TLS server.
2) In the `ServerHello` message, the TLS server samples $\beta \leftarrow\$ \mathbb{Z}_p^+$ and sends $\beta \cdot G$ to the TLS client.

3) The TLS client and server compute $\alpha\beta \cdot G$ and—with other information—derive the TLS session keys, as specified in the TLS standards [39, 40].

Step 1: Distributed generation of client randomness $\boldsymbol{\alpha \cdot G}$. To generate the client randomness $\alpha \cdot G$ used in the `ClientHello` message without revealing α, each server samples a share of α and provides a corresponding share of $\alpha \cdot G$, as follows:

1) For all servers, the i-th server $\mathcal{P}_i$ samples $\alpha_i \leftarrow\$ \mathbb{Z}_p^+$ and broadcasts $\alpha_i \cdot G$, by first committing to $\alpha_i \cdot G$ and then revealing it.
2) $\mathcal{P}_1$ computes and sends $\sum_{i=1}^{N} \alpha_i \cdot G$ to the TLS server.

This step can be done before the connection starts.

Step 2: Distributed computation of key exchange result $\boldsymbol{\alpha\beta \cdot G}$. The N servers need to jointly compute $\alpha\beta \cdot G$, which works as follows: each server computes $\alpha_i(\beta G)$ first, and then the SMPC protocol takes $\alpha_i(\beta G)$ as input from server $\mathcal{P}_i$ and computes $\alpha\beta \cdot G = \sum_{i=1}^{n} \alpha_i(\beta G)$. The result is used to derive the TLS session keys, which we discuss next.

Step 3: Distributed key derivation. The next step is to compute the TLS session keys inside SMPC using a key derivation function [41]. The N servers, within SMPC, derive the handshake secrets from $\alpha\beta \cdot G$ and the hash of the `ClientHello` and `ServerHello` messages, and then derive the handshake keys and IVs within SMPC.

We identify that the hashes of the communication transcript messages, which is needed for key derivation, can be computed outside SMPC, which reduces the overhead. That is, the forwarding server broadcasts these TLS messages to the other servers. Each server computes the hashes, and all servers input these hashes to the SMPC instance. This approach is secure because TLS already prevents against man-in-the-middle attacks, which means that these messages are not sensitive.

Step 4: Verifying the TLS connection. The TLS server sends a response containing its certificate, a signature over $\beta \cdot G$, and verification data, which the TLS client verifies and replies. Performing this verification in SMPC is slow because (1) the certificate format is difficult to parse without revealing access patterns and (2) verifying signatures involves hashing and prime field computation, both of which are slow in SMPC.

In N-for-1-Auth, we are able to remove this task from SMPC. The insight is that the handshake keys, which encrypts the response, are only designed to hide the TLS endpoints' identity, which is unnecessary because in our setting, the servers must confirm the TLS server's identity. Several works show that revealing the keys does not affect other guarantees of TLS [42–45]. We formalize this insight in our definition of the ideal functionality $\mathcal{F}_{\mathsf{TLS}}$, as described in App. B.

Therefore, verifying the TLS server's response is as follows: after all the N servers receive and acknowledge all the messages from `ServerHello` to `ServerFinished` sent by the TLS server and forwarded by the first server, the SMPC protocol reveals the TLS handshake keys to all the N servers. Each server decrypts the response and verifies the certificate, signature, and verification data within it. Then, the N servers can use the handshake key, and then within SMPC assemble the client handshake verification data, which is then sent to the TLS server. Lastly, the N servers derive the application keys, which are used for data exchange, from the handshake secrets and the hash of the transcript inside SMPC.

Step 5: Precomputation for authenticated encryption. The authenticated encryption scheme used in data exchange may allow some one-time precomputation that can be done as part of the TLS connection establishment. For example, for AES-GCM N-for-1-Auth can precompute the AES key schedule and secret-share the GCM power series. We provide more details in §III-C.

Efficient implementation. The key exchange protocol consists of point additions and key derivations. We observe that point additions can be efficiently expressed as an arithmetic circuit whose native field is exactly the point's coordinate field, and key derivations can be efficiently expressed as a boolean circuit. Our insight to achieve efficiency here is to mix SMPC protocols by first implementing point additions with SPDZ using MASCOT [46] for the offline phase, and then transferring the result to AG-MPC [36, 37] for key derivation via a maliciously secure mixing protocol [47–49]. Both SPDZ and AG-MPC support offline precomputation, which helps reduce the online latency and meet the TLS handshake timeout.

We chose MASCOT instead of other preprocessing protocols [35, 50] based on homomorphic encryption because many curves used in TLS have a coordinate field with low "2-arity", which is incompatible with the packing mechanisms in homomorphic encryption schemes.

C. Data exchange

The rest of the TLS-in-SMPC protocol involves data encryption and decryption. An opportunity to reduce the latency is to choose the TLS ciphersuites carefully, as shown by both our investigation and prior work [43, 51].

During key exchange, typically the TLS server offers several TLS ciphersuites that it supports, and the TLS client selects one of them to use. In order to minimize latency, when given the choice, our protocol always selects the most SMPC-friendly ciphersuite that is also secure.

Cost of different ciphersuites in SMPC. The cost of TLS ciphersuites in SMPC has rarely been studied. Here, we implement the boolean circuits of two commonly used ciphersuites, AES-GCM-128 and Chacha20-Poly1305—which are part of the TLS 1.3 standard and supported in many TLS 1.2 implementations—and measure their cost.

After common optimizations, the main overhead rests on the amortized cost of (1) AES without key schedule and (2) Chacha20 in terms of the number of AND gates in boolean circuits. The amortized cost per 128 bits for AES is 5120 AND gates while Chacha20 takes 96256 AND gates due to integer additions. Thus, it is preferred to choose AES-GCM-128 when available.

Efficient GCM tag computation. We adapt from DECO [43] a protocol to efficiently compute the GCM tag to N parties. After deriving the TLS application key within SMPC, the servers compute the GCM generator $H = E_K(0)$ and the power series of H: $H, H^2, H^3, ..., H^L$ within SMPC. The power series is secret-shared among the N servers. To compute the GCM tag for some data $S_1, S_2, ..., S_L$ (authenticated data or ciphertexts), each server computes a share of the polynomial $\sum_{i=1}^{L} S_i \cdot H^i$ and combines these shares with the encryption of initialization vector (IV) within SMPC.

We optimize the choice of L, which has not been done in DECO. For efficiency, L needs to be chosen carefully. A small L will increase the number of encryption operations, and a large L will increase the cost of computing the GCM power series. Formally, to encrypt message of N bytes with AES (the block size is 16 bytes), we find L that minimizes the overall encryption cost:

$$L_{\text{opt}} = \text{argmin}_L \left[\begin{array}{l} (L-1) \cdot 16384 + 1280 + 5120 \\ +M \cdot 5120 + \lceil \frac{N+M}{16} \rceil \cdot 5120 \end{array} \right].$$

where $M = \lceil \frac{N}{16 \cdot (L-2) - 1} \rceil$ is the number of data packets in the TLS layer.[1] For example, for $N = 512$, choosing $L = N/16 = 32$ is 2.3× the cost compared with $L_{\text{opt}} = 5$.

Circuit implementation. We synthesize the circuit files in TLS-in-SMPC using Synopsys's Design Compiler and tools in TinyGarble [52], SCALE-MAMBA [53], and ZKCSP [54]. The circuits have been open-sourced here.

https://github.com/n-for-1-auth/circuits

IV. N-for-1-Auth Authentication

In this section we describe how a user, using the client app, authenticates to N servers via various authentication factors. We also describe the enrollment phase needed to set up each protocol. After passing the authentication, the user can invoke the applications described in §V.

General workflow. In general, N-for-1-Auth's authentication protocols consist of two stages:

– The servers jointly send ***one*** challenge to the client.
– The client replies with a response to each server, which will be different for each server.

Depending on the application, users may want to change their authentication factors, in which they would need to authenticate with the servers beforehand.

Preventing replay attacks. The client needs to provide each server a *different* response to defend against replay attacks. If the user sends *the same response* to different servers, a malicious server who receives the response can attempt to beat the user to the honest servers. The honest servers will expect the same message that the malicious server sends, and if the malicious server's request reaches the honest servers first, the honest servers will consider the malicious server authenticated instead of the honest user. Since up to $N-1$ of the servers can collude with one another, in this scenario, the malicious server can reconstruct the shares and obtain the secret.

To prevent replay attacks, we designed the authentication protocols in a way such that no efficient attacker, knowing $N-1$ out of the N responses from an honest user, can output the remaining response correctly with a non-negligible probability.

A. *N-for-1-Auth Email*

N-for-1-Auth's email authentication protocol sends the user only one email which contains a passcode. If the user proves knowledge of this passcode in some way, the N servers will consider the user authenticated. N-for-1-Auth's email authentication protocol is as follows:

1) The i-th server $\mathcal{P}_i$ generates a random number s_i and provides it as input to SMPC.
2) Inside SMPC, the servers computes $s = \bigoplus_i^N s_i$, where $\oplus$ is bitwise XOR, and outputs $\text{PRF}(s, i)$ to $\mathcal{P}_i$, where PRF is a pseudorandom function.
3) The N servers run the TLS-in-SMPC protocol to create a TLS endpoint acting as an email gateway for some domain. The TLS endpoint opens a TLS connection with the user's SMTP server such as `gmail-smtp-in.l.google.com` for `abc@gmail.com`, and sends an email to the user with the passcode s over this TLS connection. Note that the protocol sends the email using the intergateway SMTP protocol, rather than the one used by a user to send an email.
4) The user receives the email and enters s into the client app, which computes $\text{PRF}(s, i)$ and sends the result to $\mathcal{P}_i$. If the user response matches the output that the

[1] Besides the actual payload data, the GCM hash also adds on two additional blocks (record header and the data size) and one bit (TLS record content type), which explains the term $16 \cdot (L-2) - 1$.

server received in step 2, then $\mathcal{P}_i$ considers the user authenticated.

Enrollment. The enrollment protocol is as follows:

1) The client opens a TLS connection with each of the N servers and secret-shares the user's email address and sends the i-th share to server $\mathcal{P}_i$.
2) The N servers reconstruct the user's email address within SMPC and then jointly send a confirmation email to the user, with a passcode.
3) The client proves knowledge of the passcode using N-for-1-Auth's email authentication protocol, converts the secret into N secret shares, and sends the i-th share to server $\mathcal{P}_i$.
4) If the user is authenticated, each server stores a share of the user's email address and a share of the secret.

Avoiding misclassification as spam. A common issue is that this email might be misclassified as spam, which can be handled using standard practices as follows.

– *Sender Policy Framework (SPF).* N-for-1-Auth can follow the SPF standard [55], in which the sender domain, registered during the setup of N-for-1-Auth, has a `TXT` record indicating the IP addresses of email gateways eligible to send emails from this sender domain.
– *Domain Keys Identified Mail (DKIM).* The DKIM standard [56] requires each email to have a signature from the sender domain under a public key listed in a `TXT` record. N-for-1-Auth can have the server generate the keys and sign the email, both in a distributed manner.

Our experiments show that supporting SPF is sufficient to avoid Gmail labeling N-for-1-Auth's email as spam.

Privacy. We continue with the example of Alice from the introduction, in which she authenticates to N servers, one of which is her E2EE App Server and the other $N-1$ are helper servers. In this setting, the E2EE App Server delivers packets to the TLS server. Based on our interaction with companies, the E2EE App Server does not want to reveal Alice's contact information to the other helper servers. To achieve this, for N-for-1-Auth's email authentication protocol, Alice's email address is secret-shared among these N servers, and the E2EE App Server maintains a mapping from Alice's email address to the index at which the shares are stored. During authentication, the E2EE App Server provides this index to the other helper servers. In addition, the helper servers only need to know the email provider's mail gateway address instead of the full email address. This is because the N servers verify the TLS server certificate outside of SMPC for efficiency reasons and therefore the email provider's mail gateway address is revealed.[2] For example, many companies use Google or Microsoft for email service on their domains. In this case, for a user with email address `A@B.com`, the helper servers know neither `A` nor `B.com`, but only which email service provider is used by `B.com`. This property is useful in the case of data breaches as compromising the helper servers does not reveal Alice's email. We note that for the E2EE App Server to maintain the mapping, they only need to have the full information of only *one* of the user's second-factors, such as their email or phone number, whereas the information for their other registered factors can be secret-shared and hidden from the E2EE App Server as well.

B. N-for-1-Auth SMS

N-for-1-Auth's SMS protocol sends the user *one* text message, which contains a passcode. The enrollment and authentication protocols resemble the email authentication protocol except that the passcode is sent via SMS.

We leverage the fact that many mobile carriers, including AT&T [57], Sprint [58], and Verizon [59], provide commercial REST APIs to send text messages. The N servers, who secret-share the API key, can use N-for-1-Auth's TLS-in-SMPC protocol to send a text message to the user through the relevant API.

Privacy. Similar to email, N-for-1-Auth secret-shares the user's phone number among the N servers, allowing the user's phone number to be hidden from the helper servers. Here, only the sender's carrier and the user's carrier sees the user's phone number, but the helper servers cannot. If the N servers have the SMS API to the user's carrier, they can use that API, so that only one mobile carrier sees the message.

C. N-for-1-Auth U2F

Universal second factor (U2F) [60] is an emerging authentication standard in which the user uses U2F devices to produce signatures to prove the user's identity. Devices that support U2F include YubiKey [61] and Google Titan [62]. The goal of N-for-1-Auth's U2F protocol is to have the user operate on the U2F device *once*.

Background: U2F. A U2F device attests to a user's identity by generating a signature on a challenge requested by a server under a public key that the server knows. The U2F protocol consists of an enrollment phase and an authentication phase, described as follows.

In the enrollment phase, the U2F device generates an application-specific keypair and sends a key handle and

[2] If the certification verification is done in SMPC, the gateway address can be hidden, but with a high overhead, as discussed in §III-B.

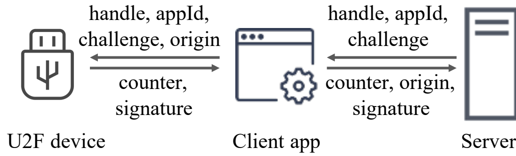


Fig. 5: Protocol of universal second factor (U2F).

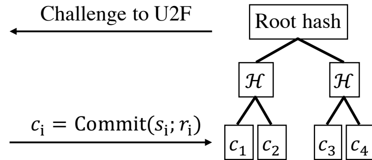


Fig. 6: The Merkle tree for U2F challenge generation.

the public key to the server. The server stores the key handle and the public key.

In the authentication phase, as Fig. 5 shows, the server generates a random challenge and sends over the key handle, the application identifier (appId), and a challenge to a U2F interface such as a client app, which is then, along with the origin name of the server, forwarded to the U2F device. Then, upon the user's confirmation, such as tapping a button on the device [61, 62], the U2F device generates a signature over the request. The signature also includes a monotonic counter to discover cloning attacks. The server receives the signature and verifies it using the public key stored in the enrollment phase.

To avoid the user having to perform a U2F authentication for each server, an intuitive approach is to have the servers generate a joint challenge which is then signed by the U2F device. The client can secret-share the signature, and the servers can then reconstruct and verify the signature within SMPC. However, signature verification in SMPC can be prohibitively expensive.

An insecure strawman. We now describe an insecure strawman, which will be our starting point in designing the secure protocol. Let the N servers jointly generate a random challenge. The strawman lets the client obtain a signature over this challenge from U2F and sends the signature to each server. Then, each server verifies the signature, and the servers consider the user authenticated if the verification passes for each server.

This approach suffers from the replay attack described in §IV. When a malicious server receives the signature from the client, this server can impersonate the honest user by sending this signature to the other servers.

N-for-1-Auth U2F's protocol. Assuming server $\mathcal{P}_i$ chooses a random challenge value s_i, our protocol must satisfy two requirements: (1) the challenge signed by the U2F device is generated using all the servers' randomness $s_1, s_2, ..., s_N$; and (2) the client can prove to server $\mathcal{P}_i$ that the signed challenge uses s_i without revealing information about other parties' randomness.

We identify that aggregating the servers' randomness via a Merkle tree combined with cryptographic commitments, as Fig. 6 shows, satisfies these requirements. We now briefly describe these two building blocks.

In a Merkle tree, if the client places the servers' randomness $s_1, s_2, ..., s_N$ into the leaf nodes, as Fig. 6 shows, then the root hash value is a collision-resistant representation of all the servers' randomness, which we will use as the challenge for the U2F device to sign over.

However, Merkle trees are not guaranteed to hide the leaf nodes' values. To satisfy the second requirement, as Fig. 6 shows, we use cryptographic commitments $c_i = \texttt{Commit}(s_i;\ r_i)$ instead of s_i as the leaf nodes' values, in which r_i is a random string chosen by the client. The commitments provide two guarantees: (1) the server, from the commitment c_i, does not learn s_i and (2) the client cannot open c_i to a different $s_i' \neq s_i$.

Next, the client obtains the signature of the root hash from U2F and sends each server the following response: (1) the signature, (2) a Merkle tree lookup proof that the i-th leaf node has value c_i, and (3) commitment opening secrets r_i and s_i. Here, only the client and the i-th server know the server randomness s_i.

The detailed authentication protocol is as follows:

1) Each server opens a TLS connection with the client and sends over a random value s_i.
2) The client builds a Merkle tree as described above and in Fig. 6 and obtains the root hash. The client requests the U2F device to sign the root hash as the challenge, as Fig. 5 shows, following the U2F protocol.
3) The user then operates on the U2F device *once*, which produces a signature over the root hash. The client app then sends the signature, the Merkle tree lookup proof, and the commitment opening information to each server.
4) Each server verifies the signature, opens the commitment, verifies that the commitment is indeed over the initial value s_i provided by server $\mathcal{P}_i$, and checks the Merkle tree lookup proof. If everything is verified, then $\mathcal{P}_i$ considers the user authenticated.

This protocol prevents replay attacks as described above since the client's response to $\mathcal{P}_i$ contains the opening secret s_i; other servers cannot determine this value with a non-negligible probability.

Enrollment. The enrollment protocol is as follows:

1) The client and the servers engage in the standard U2F enrollment protocol [60], in which the servers obtain

the key handle and the public key.

2) The client and the servers run N-for-1-Auth's U2F authentication protocol as described above.

Privacy. The U2F protocol already provides measures to hide a device's identity [63], which N-for-1-Auth leverages to provide privacy for the user.

D. N-for-1-Auth security questions

The last N-for-1-Auth authentication factor we present is security questions. Although many of the properties provided by N-for-1-Auth's security questions protocol have been achieved by prior works [17–20, 25], we briefly describe this protocol for completeness.

Typically, security questions involve the user answering a series of questions that (ideally) only the user knows all of the answers to [64–67]. During enrollment, the user provides several questions and their answers to the client app. The client then hashes the answers, and then sends secret-shares of the hashes to the N servers. Then, during authentication, the user is asked to answer these questions. The client, similar to before, hashes the provided answers and provides secret-shares to the N servers who then, within SMPC, reconstruct these hashes and compare with the hashes originally stored from enrollment. If all the hashes match, the user is considered authenticated.

Privacy and other benefits over traditional security questions. Traditionally, security questions avoid asking users for critical personal secrets, such as their SSN, because the user may feel uncomfortable sharing such personal information. Hashing and other cryptographic techniques do not help much since the answer is often in a small domain and can be found via in an offline brute-force attack. However in N-for-1-Auth, since the hashes of the answers are secret-shared among the N servers, no server knows the full hash and therefore offline brute force attacks are impossible. The privacy of the user's answers (or their hash) from the servers can encourage users to choose more sensitive questions and enter more sensitive answers that the user would otherwise be uncomfortable sharing. To hide these more sensitive questions, we can leverage an existing industry practice, *factor sequencing* [68], by showing these more sensitive questions only after the user correctly answers less sensitive questions. Furthermore, to mitigate online brute force attacks, in addition to standard rate-limiting mechanisms which N-for-1-Auth supports, they can set other authentication factors as *prerequisites*. That is, only when the user authenticates against prerequisite factors can the user even see the security questions.

V. APPLICATIONS

Once the N servers have authenticated the user, they can perform some operations for the user using their secret that is secret-shared during enrollment, such as key recovery as in our motivating example. To show the usefulness of N-for-1-Auth, we now describe two applications that can benefit from N-for-1-Auth.

Key recovery. The user can backup a key by secret-sharing it as the user secret during the enrollment phase. When the user needs to recover the key, the servers can send the shares back to the user, who can then reconstruct the key from the shares. Key recovery is widely used in end-to-end encrypted messaging apps, end-to-end encrypted file sharing apps, and cryptocurrencies.

Digital signatures. Sometimes, it is preferred to obtain a signature under a secret key, rather than retrieving the key and performing a signing operation with it. This has wide applications in cryptocurrencies, in which the user may not want to reconstruct the key and have it in the clear. Instead, the user delegates the key to several servers, who sign a transaction only when the user is authenticated. The user can also place certain restrictions on transactions, such as the maximum amount of payment per day, which can be enforced by the N servers within SMPC. In N-for-1-Auth, the user secret-shares the signing key in the enrollment phase. Before performing a transaction, the user authenticates with the servers. Once authenticated, the user presents a transaction to the N servers, who then sign it using a multi-party ECDSA protocol [69–73]. An alternative solution is to use multisignatures [74], which N-for-1-Auth can also support, but this option in unavailable in certain cryptocurrencies [3] and may produce very long transactions when N is large.

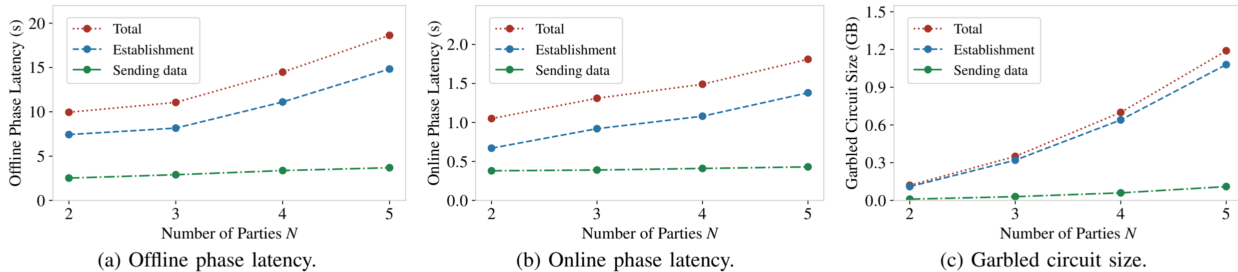
VI. EVALUATION

In this section we discuss N-for-1-Auth's performance by answering the following questions:

1) Is N-for-1-Auth's TLS-in-SMPC protocol practical? Can it meet the TLS handshake timeout? (§VI-C)
2) How efficient are N-for-1-Auth's authentication protocols? (§VI-D)
3) How does N-for-1-Auth compare with baseline implementations and prior work? (§VI-E and §VI-F)

A. Implementation

We use MP-SPDZ [75], emp-toolkit [37, 76] and wolfSSL [77] to implement N-for-1-Auth's TLS-in-SMPC protocol. We implemented the online phase of elliptic-curve point additions within SMPC from scratch in C++.



(a) Offline phase latency. (b) Online phase latency. (c) Garbled circuit size.

Fig. 7: The overall online/offline phase latencies and the garbled circuit size of the TLS-in-SMPC protocol for $N = 2, 3, 4, 5$ servers when sending an email with passcode (the mail body is 34 bytes).

Component	Offline Phase Latency (s)				Online Phase Latency (s)			
	$N = 2$	$N = 3$	$N = 4$	$N = 5$	$N = 2$	$N = 3$	$N = 4$	$N = 5$
TLS connection establishment	7.43	8.16	11.11	14.83	0.67	0.92	1.08	1.38
⋄ Client randomness generation	0.30	0.30	0.30	0.30	—	—	—	—
⋄ Key exchange result computation	0.02	0.06	0.09	0.15	0.25	0.35	0.37	0.47
⋄ Key derivation	6.55	7.05	9.73	13.1	0.37	0.51	0.64	0.83
⋄ GCM power series ($L = 5$)	0.49	0.65	0.87	1.15	0.03	0.04	0.05	0.06
⋄ AES key schedule	0.07	0.10	0.12	0.13	0.02	0.02	0.02	0.02
Sending an email of 34 bytes in TLS	2.52	2.90	3.37	3.69	0.38	0.39	0.41	0.43
Sending a SMTP heartbeat in TLS	0.43	0.49	0.57	0.63	0.06	0.07	0.07	0.07

Tab. I: Breakdown of the TLS-in-SMPC latencies for sending an email with passcode (the mail body is 34 bytes).

B. Setup

We ran our experiments on `c5n.2xlarge` instances on EC2, each equipped with a 3.0 GHz 8-core CPU and 21 GB memory. To model a cross-state setup, we set a 20 ms round-trip time and a bandwidth of 2 Gbit/s between servers (including the TLS server) and 100 Mbit/s between clients and servers.

C. TLS-in-SMPC's performance

For the TLS-in-SMPC protocol, we measured the offline and online phases' latencies and the size of the garbled circuits sent in the offline phase and show the results in Fig. 7. From the figure, we see that the offline and online phase latencies and the total circuit size grow roughly linearly to the number of servers.

We consider N from 2 to 5 in this experiment. In practice, companies with decentralized key management such as Curv [12] and Unbound Tech [13] currently use two mutually distrusting parties, and Keyless [16] uses three in their protocol. For all values of N that we tested, the protocol always meets the TLS handshake timeout.

A large portion of the offline cost is in transmitting the garbled circuits used in AG-MPC, as Fig. 7 shows. N-for-1-Auth's servers run the offline phase before the TLS connection is established to avoid this extra overhead. To load these circuits to the memory efficiently, one can use a memory planner optimized for secure computation [78].

Malicious users can perform DoS attacks by wasting computation done in the offline phase. N-for-1-Auth can defend against such attacks using well-studied techniques, such as proof-of-work or payment [79, 80].

Latency breakdown. In Tab. I we show a breakdown of the offline and online phase latencies for the TLS-in-SMPC protocol. From the table, we see that most of the computation is in the offline phase, and the online phase has a small latency. Therefore, if we run an SMPC protocol off the shelf that does not precompute the offline phase, from Tab. I we see that for $N = 5$, the key exchange has a latency of 14.83 s and cannot meet a TLS handshake timeout of 10 s.

We see from Tab. I that the latency for establishing the TLS connection dominates. However, N-for-1-Auth can create a persistent connection with the email receiving gateway server, allowing this to be a one-time cost, which is particularly useful for popular email service providers like Gmail. With an established connection, sending an email with $N = 5$ only takes 3.69 s in the offline phase and 0.43 s in the online phase, which is drastically smaller. To maintain this connection, N-for-1-Auth servers can send SMTP heartbeats (a `NOOP` command). Our experiment with Gmail show that one heartbeat per 120 s is sufficient to maintain a long-term connection for at least 30 minutes.

	Offline Phase Latency (s)	**Online Phase Latency (s)**
Email	10.96 (2.90)	1.29 (0.39)
SMS	12.26 (4.10)	1.48 (0.56)
U2F	—	0.03
Security Questions	0.03	0.04

Tab. II: Latencies of N-for-1-Auth (N = 3). Numbers in parentheses are the cost given an established TLS connection.

D. N-for-1-Auth's authentication performance

We measured the offline and online phase latencies of the N-for-1-Auth protocols and present the results in Tab. II. We now discuss the results in more detail.

Email/SMS. Using a message of 34 characters, the N-for-1-Auth email protocol (without DKIM signatures) sends 165 bytes via TLS-in-SMPC, and N-for-1-Auth's SMS protocol sends 298 bytes via TLS-in-SMPC, estimated using AT&T's SMS API documentation [57].

U2F. We implement the collision-resistant hash and commitments with SHA256. The computation time for the client and the server is less than 1 ms. The protocol incurs additional communication cost, as the client sends each server a Merkle proof of 412 bytes. We note that all of the overhead comes from the online phase.

Security questions. Checking the hashed answer of one security question can be implemented in AG-MPC, which takes 255 AND gates.

E. Comparison with off-the-shelf SMPC

We compare N-for-1-Auth's implementation with an off-the-shelf implementation in emp-toolkit [37, 76]. We estimate this cost by implementing the computation of $\alpha\beta \cdot G$ in key exchange, which offers a lower bound on its performance of TLS and is already much slower than N-for-1-Auth. With $N = 5$ servers, the overall latency is at least 8× slower compared with N-for-1-Auth's TLS-in-SMPC implementation. This is because computing $\alpha\beta \cdot G$ involves expensive prime field operations, which use 10^7 AND gates. With $N = 5$ servers, this step already takes 150 s in the offline phase and 8.6 s in the online phase.

F. Comparison with DECO

DECO [43] is a work that runs TLS in secure two-party computation. As discussed in §VII, their implementation is not suitable for N-for-1-Auth because it is restricted to two parties and has extra leakage due to a different target setting. During the TLS handshake, DECO uses a customized protocol based on multiplicative-to-additive (MtA) [71] to add elliptic curve points, while N-for-1-Auth uses SPDZ. We are unaware of how to extend DECO's protocol to $N \geq 3$. In addition, when comparing with DECO, N-for-1-Auth's AES implementation reuses the AES key schedule across AES invocations, which reduces the number of AND gates per AES invocation from 6400 to 5120, an improvement of 20%.

VII. Related work

Decentralized authentication. Decentralized authentication has been studied for many years and is still a hot research topic today. The main goal is to avoid having centralized trust in the authentication system. One idea is to replace centralized trust with trust relationships among different entities [81, 82], which has been used in the PGP protocol in which individuals prove the identities of each other by signing each other's public key [83, 84]. Another idea is to make the authentication system transparent to the users. For example, blockchain-based authentication systems, such as IBM Verify Credentials [85], BlockStack [86], and Civic Wallet [87], and certificate/key transparency systems [32, 33, 88–91] have been deployed in the real world.

A recent concurrent work [92] also addresses decentralized authentication for cryptocurrency by integrating U2F and security questions with smart contracts. Their construction does not support SMS/email authentication due to limitations of smart contracts, and does not work with cryptocurrency that does not support smart contracts like Bitcoin. In sum, their approach targets a different setting than N-for-1-Auth, as we focus on the usability issues of having the user perform N-times the work.

Password-based secret generation. There has been early work on generating strong secrets from passwords using several mutually distrusting servers, such as [25]. However, [25] focuses on passwords while N-for-1-Auth focuses on second factors, which brings its own set of unique challenges as N-for-1-Auth needs to be compatible with the protocols of these second-factors. In addition, [25] requires the user to remember a secret, which has its own issues as the secret can be lost. We note that these works are complementary to N-for-1-Auth, which can provide secure key backup for these passwords.

Decentralized storage of secrets. In industry, there are many companies that use decentralized trust to store user secrets, such as Curv [12], Partisia [93], Sepior [15], Unbound Tech [13], and Keyless [16]. These companies use SMPC to store, reconstruct, and apply user secrets in a secure decentralized manner. However, in principle a user still needs to authenticate with each of these company's servers since theses servers do not trust each other. Therefore, in these settings a user still needs to do N times the

work in order to access their secret. N-for-1-Auth's protocols can assist these commercial decentralized solutions to minimize the work of their users in authentication.

TLS and SMPC. There are works using TLS with secure two-party computation (S2PC), but in a prover-verifier setting in which the prover proves statements about information on the web. BlindCA [51] uses S2PC to inject packets in a TLS connection to allow the prover to prove to the certificate authority that they own a certain email address. However, it issue is that the prover possesses all of the secrets of the TLS connection, and all of their traffic sent to the server must go through a proxy owned by the verifier. DECO [43] uses TLS within S2PC, but its approach also gives the prover the TLS encryption key, which our setting does not allow. Overall, both of these works are restricted to two parties based on their intended settings, while N-for-1-Auth supports an arbitrary number of parties.

In addition, a concurrent work [44] also enables running TLS in secure multiparty computation, and their technical design in this module is similar to ours[3], but [44] does not propose or contribute authentication protocols for distributed trust settings and their applications. N-for-1-Auth offers contributions beyond the TLS-in-SMPC module, proposing the idea of performing N authentications with the work of one, showing how this can be achieved by running inside SMPC the SMTP protocol or the HTTP protocol in addition to TLS, to support authentication factors, and demonstrating applications in the end-to-end encryption and cryptocurrency space. In addition, within the TLS-in-SMPC protocol, we provide an end-to-end implementation compatible with an existing TLS library, wolfSSL, and show that it works for N-for-1-Auth's authentication protocols. Specifically, [44] emulated certain parts of the TLS protocol, and they only evaluated the online phase and did not measure the offline cost, which is important for real-world deployment. In contrast, we also benchmark the offline phase of our protocol.

OAuth. OAuth [27] is a standard protocol used for access delegation, which allows users to grant access to applications without giving out their passwords. While OAuth has several desirable properties, it does not work for all of N-for-1-Auth's second factors, notably SMS text messages and email service providers that do not support OAuth, and is therefore less general and flexible than N-for-1-Auth. In addition, if a user authenticates through OAuth and wants distributed trust, they have to perform the authorization N times, once for each server. N-for-1-Auth can incorporate OAuth as a separate authentication factor—the N servers can secret-share the OAuth client secret and then, using TLS-in-SMPC, obtain the identity information through the OAuth API.

[3]We implemented additional optimizations in AES and GCM.

VIII. Discussion

Handling denial-of-service attacks. In this paper, we consider denial-of-service attacks by the servers to be out of scope, as discussed in §II-A. There are some defenses against these types of attacks, as follows:

- *Threshold secret sharing.* A malicious server can refuse to provide its share of the secret to prevent the user from recovering it. To handle this, the user can share the secret in a *threshold* manner with a threshold parameter t which will allow the user's secret to be recoverable as long as t servers provide their shares. This approach has a small cost, as a boolean circuit for Shamir secret sharing only takes 10240 AND gates by using characteristic-2 fields for efficient computation.
- *Identifiable abort.* Some new SMPC protocols allow for identifiable abort, in which parties who perform DoS attacks by aborting the SMPC protocol can be identified [94, 95]. N-for-1-Auth can support identifiable abort by incorporating these SMPC protocols and standard identification techniques in its authentication protocols.

IX. Conclusion

N-for-1-Auth is an authentication system that decentralizes trust across N servers and allows users to authenticate to the servers while only performing the work of a single authentication. N-for-1-Auth offers authentication protocols that achieve this property for various commonly used authentication factors. At the core of N-for-1-Auth is a TLS-in-SMPC protocol, which we designed to be efficient enough to meet the TLS timeout and successfully communicate with an unmodified TLS server. We hope that N-for-1-Auth will facilitate the adoption of new systems with decentralized trust.

REFERENCES

[1] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, https://bitcoin.org/bitcoin.pdf.
[2] *Ethereum*, https://ethereum.org/.
[3] *Zcash: Privacy-protecting digital currency*, https://z.cash/.
[4] *WhatsApp*, https://www.whatsapp.com/.
[5] *Signal*, https://signal.org/.
[6] *Telegram messenger*, https://telegram.org.
[7] *Preveil: Encrypted email and file sharing for the enterprise*, https://www.preveil.com/.
[8] *Line*, https://www.line.me/.
[9] *Keybase*, https://keybase.io/.
[10] *Algorand*, https://www.algorand.com.
[11] *Proton mail*, https://protonmail.com/.
[12] *Curv: The institutional standard for digital asset security*, https://www.curv.co.
[13] *Unbound tech: Secure cryptographic keys across any environment*, https://www.unboundtech.com/.
[14] *Bitgo*, https://www.bitgo.com/.
[15] *Sepior: Threshold cryptographic key management solutions with MPC*, https://sepior.com.
[16] *Keyless: Zero-trust passwordless authentication*, https://keyless.io/.
[17] A. Bagherzandi, S. Jarecki, N. Saxena, and Y. Lu, "Password-protected secret sharing," in *CCS '11*.
[18] M. Abdalla, M. Cornejo, A. Nitulescu, and D. Pointcheval, "Robust password-protected secret sharing," in *ESORICS '16*.
[19] P. D. MacKenzie, T. Shrimpton, and M. Jakobsson, "Threshold password-authenticated key exchange," in *CRYPTO '02*.
[20] M. D. Raimondo and R. Gennaro, "Provably secure threshold password-authenticated key exchange," in *EUROCRYPT '03*.
[21] A. Whitten and J. D. Tygar, "Why johnny can't encrypt: A usability evaluation of PGP 5.0," in *USENIX Security '99*.
[22] S. Ruoti, J. Andersen, D. Zappala, and K. E. Seamons, "Why johnny still, still can't encrypt: Evaluating the usability of a modern PGP client," in *arXiv:1510.08555 '15*.
[23] C. S. Weir, G. Douglas, T. Richardson, and M. A. Jack, "Usable security: User preferences for authentication methods in eBanking and the effects of experience," in *Interacting with Computers '10*.
[24] C. Braz and J. Robert, "Security and usability: The case of the user authentication methods," in *International Conference of the Association Francophone d'Interaction Homme-Machine '06*.
[25] W. Ford and B. S. Kaliski, "Server-assisted generation of a strong secret from a password," in *IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises '00*.
[26] *TOTP: Time-based one-time password algorithm*, https://tools.ietf.org/html/rfc6238.
[27] *OAuth*, https://www.oauth.net/.
[28] A. C.-C. Yao, "How to generate and exchange secrets," in *FOCS '86*.
[29] O. Goldreich, S. M. Micali, and A. Wigderson, "How to play ANY mental game: A completeness theorem for protocols with honest majority," in *STOC '87*.
[30] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *STOC '88*.
[31] D. Chaum, C. Claude, and I. Damgård, "Multiparty unconditionally secure protocols," in *STOC '88*.
[32] *Certificate transparency*, https://www.certificate-transparency.org/.
[33] *Key transparency*, https://github.com/google/keytransparency.
[34] Y. Lindel, "How to simulate it: A tutorial on the simulation proof technique," in *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pp. 277–346.
[35] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits," in *ESORICS '13*.
[36] X. Wang, S. Ranellucci, and J. Katz, "Global-scale secure multiparty computation," in *CCS '17*.
[37] K. Yang, X. Wang, and J. Zhang, "More efficient MPC from improved triple generation and authenticated garbling," in *CCS '20*.
[38] W. Diffie and M. E. Hellman, "New directions in cryptography," in *TIT '76*.
[39] *The transport layer security (TLS) protocol version 1.3*, https://tools.ietf.org/html/rfc8446.
[40] *The illustrated TLS 1.3 connection*, https://tls13.ulfheim.net/.
[41] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *CRYPTO '10*.
[42] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *S&P '17*.
[43] F. Zhang, S. K. D. Maram, H. Malvai, S. Goldfeder, and A. Juels, "DECO: Liberating web data using decentralized oracles for TLS," in *CCS '20*.
[44] D. Abram, I. Damgård, P. Scholl, and S. Trieflinger, "Oblivious TLS via multi-party computation," in *CT-RSA '21*.
[45] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol candidates," in *CCS '15*.
[46] M. Keller and E. Orsini, "MASCOT: Faster malicious arithmetic secure computation with oblivious transfer," in *CCS '16*.
[47] D. Rotaru and T. Wood, "MArBled circuits: Mixing arithmetic and boolean circuits with active security," in *INDOCRYPT '19*.
[48] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood, "Zaphod: Efficiently combining LSSS and garbled circuits in SCALE," in *WAHC '19*.
[49] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for MPC over mixed arithmetic-binary circuits," in *CRYPTO '20*.
[50] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making SPDZ great again," in *EUROCRYPT '18*.

[51] L. Wang, G. Asharov, R. Pass, T. Ristenpart, and a. shelat, "Blind certificate authorities," in *S&P' 19*.

[52] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "Tinygarble: Highly compressed and scalable sequential garbled circuits," in *S&P '15*.

[53] *Scale-mamba*, https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[54] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo, "Zero-knowledge contingent payments revisited: Attacks and payments for services," in *CCS '17*.

[55] *Sender policy framework (SPF) for authorizing use of domains in email*, https://tools.ietf.org/html/rfc7208.

[56] *DomainKeys identified mail (DKIM) signatures*, https://tools.ietf.org/html/rfc6376.

[57] *AT&T SMS API*, https://developer.att.com/sms.

[58] *Sprint enterprise messaging developer APIs*, https://sem.sprint.com/developer-apis/.

[59] *Verizon's enterprise messaging access gateway*, https://ess.emag.vzw.com/emag/login.

[60] *What is U2F?* https://developers.yubico.com/U2F/.

[61] *YubiKey strong two factor authentication*, https://www.yubico.com/.

[62] *Titan security key*, https://cloud.google.com/titan-security-key.

[63] *Why FIDO U2F was designed to protect your privacy*, https://fidoalliance.org/fido-technotes-the-truth-about-attestation/.

[64] M. Just and D. Aspinall, "Personal choice and challenge questions: A security and usability assessment," in *SOUPS '09*.

[65] S. E. Schechter, A. J. B. Brush, and S. Egelman, "It's no secret: Measuring the security and reliability of authentication via 'secret' questions," in *S&P '09*.

[66] A. Rabkin, "Personal knowledge questions for fallback authentication: Security questions in the era of facebook," in *SOUPS '08*.

[67] M. Toomim, X. Zhang, J. Fogarty, and J. A. Landay, "Access control by testing for shared knowledge," in *CHI '08*.

[68] *Mfa factor sequencing*, https://help.okta.com/en/prod/Content/Topics/Security/mfa-factor-sequencing.htm.

[69] R. Gennaro, S. Goldfeder, and A. Narayanan, "Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security," in *ACNS '16*.

[70] D. Boneh, R. Gennaro, and S. Goldfeder, "Using level-1 homomorphic encryption to improve threshold DSA signatures for Bitcoin wallet security," in *LATINCRYPT '17*.

[71] R. Gennaro and S. Goldfeder, "Fast multiparty threshold ECDSA with fast trustless setup," in *CCS '18*.

[72] J. Doerner, Y. Kondi, E. Lee, and A. Shelat, "Threshold ECDSA from ECDSA assumptions: The multiparty case," in *S&P '19*.

[73] R. Gennaro and S. Goldfeder, "One round threshold ECDSA with identifiable abort," in *IACR ePrint 2020/540*.

[74] *Bitcoin's multisignature*, https://en.bitcoin.it/wiki/Multisignature.

[75] *Multi-Protocol SPDZ (MP-SPDZ)*, https://github.com/data61/MP-SPDZ.

[76] *Efficient multi-party (EMP) computation toolkit*, https://github.com/emp-toolkit/.

[77] *Wolfssl embedded ssl/tls library — now supporting tls 1.3*, https://https://www.wolfssl.com/.

[78] S. Kumar, D. Culler, and R. A. Popa, "Nearly zero-cost virtual memory for secure computation," in *OSDI '21*.

[79] D. Lazar and N. Zeldovich, "Alpenhorn: Bootstrapping secure communication without leaking metadata," in *OSDI '16*.

[80] Y. Hu, S. Kumar, and R. A. Popa, "Ghostor: Toward a secure data-sharing system from decentralized trust," in *NSDI '20*.

[81] R. Yahalom, B. Klein, and T. Beth, "Trust relationships in secure systems: A distributed authentication perspective," in *S&P '93*.

[82] T. Beth, M. Borcherding, and B. Klein, "Valuation of trust in open networks," in *ESORICS '94*.

[83] *OpenPGP message format*, https://tools.ietf.org/html/rfc4880.

[84] *Biglumber: Key signing coordination*, http://www.biglumber.com/.

[85] *IBM Verify Credentials: Transforming digital identity into decentralized identity*, https://www.ibm.com/blockchain/solutions/identity.

[86] *Blockstack*, https://www.blockstack.org/.

[87] *Civic wallet - digital wallet for money and cryptocurrency*, https://www.civic.com/.

[88] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: bringing key transparency to end users," in *SEC '15*.

[89] J. Bonneau, "EthIKS: Using Ethereum to audit a CONIKS key transparency log," in *FC '16*.

[90] A. Tomescu, V. Bhupatiraju, D. Papadopoulos, C. Papamanthou, N. Triandopoulos, and S. Devadas, "Transparency logs via append-only authenticated dictionaries," in *CCS '19*.

[91] S. Eskandarian, E. Messeri, J. Bonneau, and D. Boneh, "Certificate transparency with privacy," in *PETS '17*.

[92] F. Breuer, V. Goyal, and G. Malavolta, "Cryptocurrencies with security policies and two-factor authentication," in *EuroS&P '21*.

[93] *Partisia: Digital infrastructure with no single point of trust*, https://partisia.com/key-management/.

[94] C. Baum, E. Orsini, P. Scholl, and E. Soria-Vazquez, "Efficient constant-round MPC with identifiable abort and public verifiability," in *CRYPTO '20*.

[95] Y. Ishai, R. Ostrovsky, and V. Zikas, "Secure multi-party computation with identifiable abort," in *CRYPTO '14*.

[96] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven, "The wonderful world of global random oracles," in *EUROCRYPT '18*.

[97] R. Canetti, A. Jain, and A. Scafuro, "Practical UC security with a global random oracle," in *CCS '14*.

[98] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS '01*.

[99] H. K. Maji, M. Prabhakaran, and M. Rosulek, "Complexity of multi-party computation functionalities," in *IACR ePrint 2013/042*.

APPENDIX

In this section we provide a security proof for TLS-in-SMPC, following the definition in §II-A.

A. Overview

We model the security in the real-ideal paradigm [34], which considers the following two worlds:

- **In the real world,** the N servers run protocol Π, N-for-1-Auth's TLS-in-SMPC protocol, which establishes, inside SMPC, a TLS client endpoint that connects to an unmodified, trusted TLS server. The adversary $\mathcal{A}$ can statically compromise up to $N-1$ out of the N servers and can eavesdrop and modify the messages being transmitted in the network, although some of these messages are encrypted.
- **In the ideal world,** the honest servers, including the TLS server, hand over their information to the ideal functionality $\mathcal{F}_{\mathsf{TLS}}$. The simulator $\mathcal{S}$ obtains the input of the compromised parties in $\vec{x}$ and can communicate with $\mathcal{F}_{\mathsf{TLS}}$. $\mathcal{F}_{\mathsf{TLS}}$ executes the TLS 1.3 protocol, which is assumed to provide a secure communication channel.

We then prove the security in the $\{\mathcal{F}_{\mathsf{SMPC}}, \mathcal{F}_{\mathsf{rpRO}}\}$-hybrid model, in which we abstract the SPDZ protocol and the AG-MPC protocol as one ideal functionality $\mathcal{F}_{\mathsf{SMPC}}$ and abstract the random oracle used in commitments with an ideal functionality for a *restricted programmable random oracle* $\mathcal{F}_{\mathsf{rpRO}}$, which is formalized in [96, 97].

Remark: revealing the server handshake key is safe. In the key exchange protocol described in §III-B, the protocol reveals the server handshake key and IV to all the N-for-1-Auth servers after they have received and acknowledged the handshake messages. This has benefits for both simplicity and efficiency as TLS-in-SMPC does not need to validate a certificate inside SMPC, which would be expensive.

Informally, revealing the server handshake key is secure because these keys are designed only to hide the server's identity [39], which is a new property of TLS 1.3 that does not exist in TLS 1.2. This property is unnecessary in our setting in which the identity of the unmodified TLS server is known.

Several works have formally studied this problem and show that revealing the keys does not affect other guarantees of TLS [42–45]. Interested readers can refer to these works for more information.

B. Ideal functionalities

Ideal functionality. In the ideal world, we model the TLS interaction with the unmodified, trusted TLS server as an ideal functionality $\mathcal{F}_{\mathsf{TLS}}$. We adopt the workflow of the standard secure message transmission (SMT) functionality $\mathcal{F}_{\mathsf{SMT}}$ defined in [98].

Given the input $\vec{x}$, $\mathcal{F}_{\mathsf{TLS}}$ runs the TLS client endpoint, which connects to the TLS server, and allows the adversary to be a man-in-the-middle attacker by revealing the messages in the connection to the attacker and allowing the attacker to modify such messages. In more detail,

1) To start, all the N servers must first provide their parts of the TLS client input $\vec{x}$ to $\mathcal{F}_{\mathsf{TLS}}$.
2) For each session id sid, $\mathcal{F}_{\mathsf{TLS}}$ launches the TLS client with input $\vec{x}$ and establishes the connection between the TLS client and the TLS server.
3) The adversary can ask $\mathcal{F}_{\mathsf{TLS}}$ to proceed to the next TLS message by sending a $(\texttt{Proceed}, sid)$ message. Then, $\mathcal{F}_{\mathsf{TLS}}$ generates the next message by continuing the TLS protocol and sends this message to the adversary for examination. The message is in the format of a backdoor message $(\texttt{Sent}, sid, S, R, m)$ where S and R denote the sender and receiver. When the adversary replies with $(\texttt{ok}, sid, m', R')$, $\mathcal{F}_{\mathsf{TLS}}$ sends out this message m' to the receiver R'.
4) The adversary can send $(\texttt{GetHandshakeKeys}, sid)$ to $\mathcal{F}_{\mathsf{TLS}}$ for the server handshake key and IV after the server's handshake response has been delivered. This is secure as discussed in App. D. $\mathcal{F}_{\mathsf{TLS}}$ responds with $(\texttt{reveal}, sid, skey, siv, ckey, civ)$ where $skey$ and siv are the server handshake key and IV, and $ckey$ and civ are the client handshake key and IV.
5) If any one of the TLS client and server exits, either because there is an error due to invalid messages or because the TLS session ends normally, $\mathcal{F}_{\mathsf{TLS}}$ considers the session with session ID sid ended and no longer handles requests for this sid.
6) $\mathcal{F}_{\mathsf{TLS}}$ ignores other inputs and messages.

Multiparty computation functionality. In the hybrid model, we abstract SPDZ and AG-MPC as an ideal functionality $\mathcal{F}_{\mathsf{SMPC}}$, which provides the functionality of multiparty computation with abort. We require $\mathcal{F}_{\mathsf{SMPC}}$ to be reactive, meaning that it can take some input and reveal some output midway through execution, as specified in the function f being computed. A reactive SMPC can be constructed from a non-reactive SMPC scheme by secret-sharing the internal state among the N parties in a non-malleable manner, as discussed in [99]. $\mathcal{F}_{\mathsf{SMPC}}$ works as follows:

1) For each session sid, $\mathcal{F}_{\mathsf{SMPC}}$ waits for party $\mathcal{P}_i$ to send $(\texttt{input}, sid, i, x_i, f)$, in which sid is the session ID, i is the party ID, x_i is the party's input, and f is the function to be executed.
2) Once $\mathcal{F}_{\mathsf{SMPC}}$ receives all the N inputs, it checks if

all parties agree on the same f, if so, it computes the function $f(x_1, x_2, ..., x_N) \to (y_1, y_2, ..., y_N)$ and sends $(\texttt{output}, sid, i, y_i)$ to party $\mathcal{P}_i$. Otherwise, it terminates this session and sends $(\texttt{abort}, sid)$ to all the N parties.

3) If $\mathcal{F}_{\mathsf{SMPC}}$ receives $(\texttt{Abort}, sid)$ from any of the N parties, it sends $(\texttt{abort}, sid)$ to all the N parties.
4) $\mathcal{F}_{\mathsf{SMPC}}$ ignores other inputs and messages.

Restricted programmable random oracle. We use commitments in §III-B to ensure that in Diffie-Hellman key exchange, the challenge $\alpha \cdot G$ is a random element. This is difficult to do without commitments because the adversary can control up to $N-1$ parties to intentionally affect the result of $\alpha \cdot G = \sum_{i=1}^{N} \alpha_i \cdot G$. In our security proof, we leverage a restricted programmable random oracle [96, 97], which is described as follows:

1) $\mathcal{F}_{\mathsf{rpRO}}$ maintains an initially empty list of (m, h) for each session, identified by session ID sid, where m is the message, and h is the digest.
2) Any party can send a query message $(\texttt{Query}, sid, m)$ to $\mathcal{F}_{\mathsf{rpRO}}$ to ask for the digest of message m. If there exists h such that (m, h) is already in the list for session sid, $\mathcal{F}_{\mathsf{rpRO}}$ returns $(\texttt{result}, sid, m, h)$ to this party. Otherwise, it samples h from random, stores (m, h) in the list for sid, and returns $(\texttt{result}, sid, m, h)$.
3) Both the simulator $\mathcal{S}$ and the real-world adversary $\mathcal{A}$ can send a message $(\texttt{Program}, m, h)$ to $\mathcal{F}_{\mathsf{rpRO}}$ to program the random oracle at an unspecified point h, meaning that there does not exist m such that (m, h) is on the list.
4) In the real world, all the parties can check if a hash is programmed, which means that if $\mathcal{A}$ programs a point, other parties would discover. However, in the ideal world, only $\mathcal{S}$ can perform such a check, and thus $\mathcal{S}$ can forge the adversary's state as if no point had been programmed.

C. Simulator

We now describe the simulator $\mathcal{S}$. Without loss of generality, we assume the attacker compromises exactly $N-1$ servers and does not abort the protocol, and we also assume that $\mathcal{A}$ does not program the random oracle, since in the real world, any parties can detect that and can then abort. We now follow the TLS workflow to do simulation. As follows, we use I to denote the set of identifiers of the compromised servers.

1) Simulator $\mathcal{S}$ provides the inputs of the compromised servers to $\mathcal{F}_{\mathsf{TLS}}$, which would start the TLS protocol.
2) $\mathcal{S}$ lets $\mathcal{F}_{\mathsf{TLS}}$ proceed in the TLS protocol and obtains the `ClientHello` message, which contains a random $\alpha \cdot G$. Now, $\mathcal{S}$ simulates the distributed generation of $\alpha \cdot G$ as follows:
   a) $\mathcal{S}$ samples a random h in the digest domain, pretends that it is the honest party's commitment, and generates the commitments of $\alpha_i \cdot G$ for $i \in I$.
   b) $\mathcal{S}$ sends $(\texttt{Program}, r||(\alpha \cdot G - \sum_{i \in I} \alpha_i \cdot G), h)$ to $\mathcal{F}_{\mathsf{rpRO}}$, where r is the randomness used for making a commitment, and $||$ is concatenation. As a result, $\mathcal{S}$ can open the commitment h to be $\alpha \cdot G - \sum_{i \in I} \alpha_i \cdot G$.
   c) $\mathcal{S}$ continues with the TLS-in-SMPC protocol, in which the N parties open the commitments and construct $\alpha \cdot G$ as the client challenge.
3) $\mathcal{S}$ lets $\mathcal{F}_{\mathsf{TLS}}$ proceed in the TLS protocol and obtains the messages from `ServerHello` to `ClientFinished`, which contain $\beta \cdot G$ and ciphertexts of the server's certificate, the server's signature of $\beta \cdot G$, and the server verification data. Now $\mathcal{S}$ needs to simulate the rest of the key exchange.
   a) $\mathcal{S}$ sends $(\texttt{GetHandshakeKeys}, sid)$ to $\mathcal{F}_{\mathsf{TLS}}$ to obtain the server/client handshake key and IV.
   b) $\mathcal{S}$ simulates the computation of the handshake keys in SMPC by pretending that the SMPC output is the handshake keys. Note: we already assume that without loss of generality, the compromised servers provide the correct $\alpha\beta \cdot G$. If they provide incorrect values, $\mathcal{S}$ would have detected this and can replace the output with an incorrect key.
   c) $\mathcal{S}$ then simulates the remaining operations of key exchange in SMPC, which checks the server verification data and produces the client verification data.
4) $\mathcal{S}$ simulates the message encryption and decryption of the application messages by simply pretending the SMPC output is exactly the ciphertexts taken from actual TLS messages, also provided by $\mathcal{F}_{\mathsf{TLS}}$.
5) In the end, $\mathcal{S}$ outputs whatever the adversary $\mathcal{A}$ would output in the real world.

D. Proof of indistinguishability

We now argue that the two worlds' outputs are computationally indistinguishable. The outputs are almost identical, so we only need to discuss the differences.

1) In distributed generation of $\alpha \cdot G$, the only difference in the simulated output compared with Π's is that the honest party chooses its share as $\alpha \cdot G - \sum_{i \in I} \alpha_i G$ and uses a programmed hash value h for commitment. Since $\alpha \cdot G$ is sampled from random by the TLS client inside $\mathcal{F}_{\mathsf{TLS}}$, it has the same distribution as the

$\alpha_i \cdot G$ sampled by an honest party. The properties of restricted programmable random oracle $\mathcal{F}_{\mathsf{rpRO}}$ show that no parties can detect that h has been programmed.

2) For the remaining operations, the main difference is that the SMPC is simulated without the honest party's secret (in the real-world protocol Π, such secret is a share of the internal SMPC state that contains the TLS session keys). The properties of SMPC show that such simulation is computationally indistinguishable.

As a result, we have the following theorem.

Theorem A.1. Assuming secure multiparty computation, random oracle, and other standard cryptographic assumptions, the TLS-in-SMPC protocol Π with N parties securely realizes the TLS client ideal functionality $\mathcal{F}_{\mathsf{TLS}}$ in the presence of a malicious attacker that statically compromises up to $N-1$ out of the N parties.