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

Weikeng Chen
weikengchen@berkeley.edu
UC Berkeley

Ryan Deng
rdeng2614@berkeley.edu
UC Berkeley

Raluca Ada Popa
raluca.popa@berkeley.edu
UC Berkeley

***Abstract*—Decentralizing trust is a prominent principle in the design of end-to-end encryption and cryptocurrency systems. A common issue in these applications is that users possess critical secrets, and users can lose precious data or assets if these secrets are lost. This issue remains a pain-point in the adoption of these systems. Existing approaches to solve this issue 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 which enables a user to authenticate to N servers *independently*, with the work of only one authentication. N-for-1 Auth provides the same user experience in the distributed trust setting to the user experience in a typical centralized system.**

I. Introduction

Decentralizing trust is a core principle in the design of 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 [1–10]. 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 or stolen, the user can lose access to his/her precious data or assets.

To explain concretely 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 typically installs a E2EE App Client on her device such as her cell phone. The client contains her secret key to decrypt her data. For the sake of usability and adoption, Alice should not have to deal with backing up the secret key herself, or interface with the application in ways that a typical user is not used to. We are concerned with the situation when Alice loses her cell phone, thus losing access to her secret key. With WhatsApp [4] and Line [8], which are end-to-end encrypted chat applications, Alice uses 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, 11–15] and academic works [16–19] have proposed to secret-share her secrets across N servers, such 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 loses her ability to decrypt her data and to authenticate with the E2EE App servers. The E2EE App must rely on existing factors such as email, SMS, U2F, security questions and others to authenticate Alice. 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, since the N servers cannot trust each other, Alice has to authenticate to each server separately. Consider email verification. This means that Alice has to perform N times the work—reading N emails. For the sake of distributed trust, the E2EE App should require multiple factors (email, SMS, U2F, security questions, and others), 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 whether or not she will lose her devices.* The reason is that 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. In addition, even for $N = 2$ in which there might be only one additional email and text message, this is a completely different user experience that adds friction. The end-to-end encryption space has already been plagued by usability issues [20, 21]. Academic works [22, 23] argue the importance of the consistency of user experience as well as minimizing user effort for the adoption of such systems. To give a

concrete example, PreVeil, an end-to-end encryption company attempted to deploy such a protocol, but according to their CEO, "We considered applying the principle of distributed trust to recovering individual keys by storing shards of keys across independent systems, but the user experience of using such an approach, in essence authenticating to multiple systems one at a time, is confusing and cumbersome." As a result, users who do not take special actions to backup their key when joining the system lose access to their data [4–9, 24]. This initial bar of entry to use the system is a deterrent that prevents widespread adoption of end-to-end encryption.

There are a number of natural strawman designs that can address this burden for Alice, but they are unviable.

One potential solution is to build a client app that can automatically perform the N authentications for Alice. In the case of email/SMS authentication, the client app would need to parse the emails or text messages Alice receives from the N servers. However, this either requires the client app to have intrusive permissions or requires very specific APIs on the email/SMS server side (which are currently undeveloped), which we believe to be unreasonable for users of these applications. Another class of approaches [16–19, 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 as her to the N servers. In this case, we return back to the original problem of Alice needing a reliable way to store this authentication secret.

A. *N-for-1 Auth Authentication*

We present N-for-1 Auth, which alleviates this burden by enabling a user to authenticate to N servers by doing only the work of authenticating with one. This matches the experience of authenticating to an application with centralized trust.

N-for-1 Auth supports many second factors that users are accustomed to, including email, SMS, security questions, and authentication tokens (e.g., YubiKey). Importantly, N-for-1 Auth requires no changes to the protocols of these forms of authentication. We discuss N-for-1 Auth's authentication protocols for each factor in §IV.

N-for-1 Auth provides 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. We discuss the threat model in more detail in §II-A.

N-for-1 Auth strives to provide meaningful privacy properties for the users. Users of second-factor authentication 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. N-for-1 Auth only reveals minimal metadata to the servers. We discuss N-for-1 Auth's privacy properties for each second factor in §IV.

N-for-1 Auth provides an efficient implementation for several second factors and is 10× faster than a naive implementation that does not leverage these application-specific optimizations. For example, N-for-1 Auth's email authentication protocol is efficient enough to avoid a TLS timeout and can successfully communicate with an unmodified TLS email server.

B. *Techniques*

We now discuss the techniques N-for-1 Auth uses to maintain the same user experience while decentralizing trust.

One passcode, *N* servers. Let us consider email authentication. How do N servers coordinate to send one email with a 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 inside secure multiparty computation (SMPC) [26–29]. In this way, none of the servers learn the passcode. However, an immediate question arises: how do these N servers 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 mail gateway server. The mismatch in our setting is that the sender is now composed of 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. 1 shows. When connecting with the user's email server, the N servers establish 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 generates TLS secrets in a secure distributed manner such that none of the servers learn the full TLS secret or the derived TLS keys. Therefore, the server that forwards the traffic can be arbitrary and does not affect security, since none of the servers know the TLS keys used to encrypt and authenticate the traffic.

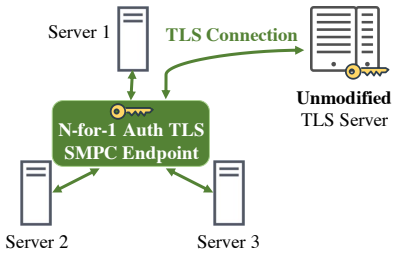


Fig. 1: TLS-in-SMPC's system architecture.

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.

Support for different authentication factors. Beyond email, N-for-1 Auth supports text message, security questions, and authentication tokens that use the universal second factor (U2F) protocol. 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. N-for-1 Auth systematically discusses how to defend against replay attacks for generic authentication protocols in §IV.

End-to-end implementation for TLS-in-SMPC. TLS is an intricate protocol that involves many cryptographic operations. If we naively run the TLS endpoint using a maliciously-secure SMPC library off the shelf, our experiments in §VI-E show that the online phase latency is 45 s. Take Gmail as an example, which has a TLS handshake timeout of 10 s. With the naive implementation, Gmail's server will terminate the connection due to timeout, thus making the system impractical.

We designed our TLS-in-SMPC protocol and optimized it for concrete efficiency with a number of insights based on the TLS protocol itself. We provide an end-to-end implementation that allows N servers to successfully connect with existing TLS endpoints.

Contributions. N-for-1 Auth's contributions are as follows:

- The design and end-to-end implementation of protocols that enable decentralized trust while preserving the same user experience for several common second factors such as email, SMS, security questions, and hardware tokens.
- The design and implementation of TLS-in-SMPC, a protocol for running a TLS client endpoint within SMPC, with a number of optimizations for concrete efficiency.
- An experimental evaluation of N-for-1 Auth's protocols.

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 a central point of trust. The user holds a secret that they wish to store on N-for-1 Auth's servers in a decentralized manner.

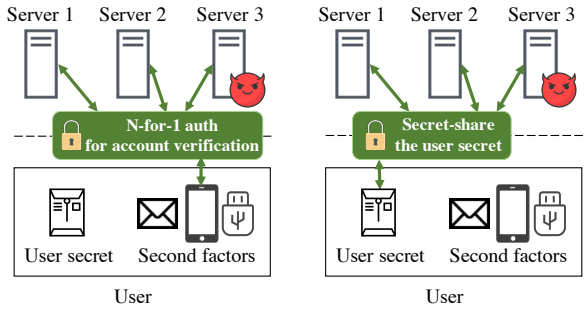
Each user can download a *stateless* client application or use a web client to participate in these protocols. Here, this minimalist client app *does not retain secrets* or demand intrusive permissions to data in other applications such as a user's emails or text messages; it simply serves as an interface 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 to hide the user's sensitive data from our client app, respectively.

Workflow. The system consists of two phases:

– *Registration (Fig. 2).* 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 user secret and distributes the shares across the servers.

– *Authentication (Fig. 3).* The user runs the N-for-1 Auth protocols for the authentication factors. Once the user is authenticated, the N servers can perform some computation over the secret for the user, which is application-specific, as we describe in §V.

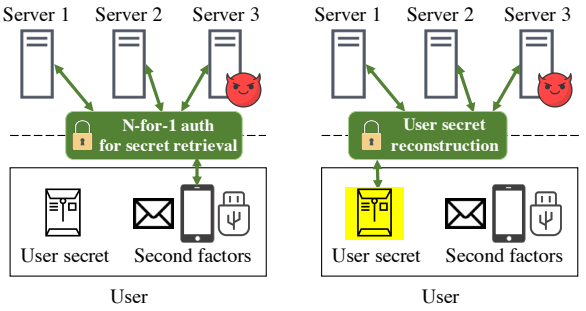
Although in some use cases such as key recovery, the authentication step only occurs in catastrophic situations, all users must perform this authentication step with the N servers when registering with the system. This authentication typically requires N times the effort and is a different user experience when compared to authenticating with a



(a) Servers authenticate the user through authentication factors.

(b) The user secret-shares the user secret among the servers.

Fig. 2: Registration workflow.



(a) Servers authenticate the user via authentication factors.

(b) The user reconstructs the secret from shares.

Fig. 3: Authentication workflow.

centralized system.

N-for-1 Authentications. We describe N-for-1 Auth's authentication mechanisms for several factors.

– *Email (§IV-A).* 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 correctly.

– *SMS (§IV-B).* The N servers jointly send *one* text message via *short message service* (SMS) to the user's phone number with a passcode. During authentication, the servers expect the user to enter this passcode.

– *U2F (§IV-C).* The N servers collaboratively initiate *one* request to a universal second factor (U2F) device. During authentication, the servers expect a digital signature, signed by the U2F device, over this request.

– *Security questions (§IV-D).* The user initially provides a series of questions and corresponding answers to the servers. During authentication, the N servers ask the user to answer these questions and expect answers consistent with those initially provided by the user. We note that passwords are a special case of security questions and can be verified using this protocol.

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

– *Key recovery.* The user can backup critical secrets by secret-sharing the secrets 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 registers with N-for-1 Auth through the client app. She provides them with three second factors: her email address, her phone number, and her U2F public key and key handle. The client app then contacts the N servers and then secret-shares the second factor information to them. The N servers then send *one* email and *one* text message, both containing a random passcode, and also send *one* request to Alice's U2F device. Alice then enters the passcodes on the client app, and responds to the request sent to her U2F device by the client app. 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.

A. Threat model and security guarantees

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 presenting our protocols, 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 or checked against a trusted hash, as in the case of the software clients in end-to-end encrypted or cryptocurrency systems. The client app either has hardcoded the TLS certificates of the N servers, or obtains them from a trusted certificate authority or a transparency ledger [30, 31]. This enables clients and servers to connect to one

another securely using the TLS protocol.

N-for-1 Auth is built on top of existing second-factor authentication and maintains the same security properties that the existing protocols provide under this threat model.

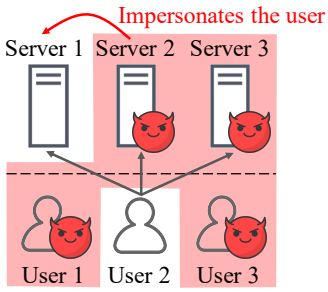


Fig. 4: N-for-1 Auth's threat model. The **red** area indicates a group of malicious parties who collude with one another.

At a high level, the core security guarantee of N-for-1 Auth is as follows. For a given authentication factor that is not compromised, even if an attacker compromised $N-1$ out of the N servers and tries to authenticate as an honest user, the attacker will not succeed to authenticate in N-for-1 Auth.

This guarantee rests on the security of N-for-1 Auth's TLS-in-SMPC protocol, which we now define. 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 [32]:

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* $I \subseteq \{1,2,...,N\}$,

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

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,* $\mathrm{REAL}_{\Pi,I,\mathcal{A}}(\vec{x})$ *denotes the joint output of* $\mathcal{A}$ *and the honest parties, and* $\overset{c}{\approx}$ *refers to computational indistinguishability.*

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

III. TLS in SMPC

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

Background: secure multiparty computation. The goal of secure multiparty computation (SMPC) [26–29] 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 typically implemented using either arithmetic circuits such as in SPDZ [33] or boolean circuits such as in AG-MPC [34]. These protocols consist of an offline phase and an online phase. The offline phase is independent of the function's input and can therefore 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 TLS.

As Fig. 1 shows, the N-for-1 Auth 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 through the first N-for-1 Auth server. The specific server that forwards the packets does not affect security since none of the servers possess the TLS session keys. Therefore, none of the servers can decrypt the packets being forwarded or inject valid packets into the TLS connection. The TLS-in-SMPC protocol has two components:

- **Key exchange:** The N-for-1 Auth servers collaboratively generate the client-side secret for Diffie-Hellman key exchange. After receiving the server-side secret, they derive the TLS session keys inside SMPC.
- **Message encryption and decryption:** The N-for-1 Auth servers, within SMPC, use the session keys to encrypt or decrypt a message.

Challenges. A straightforward implementation of the TLS-in-SMPC protocol is to use an existing malicious SMPC protocol off the shelf. However, the overhead of

running a TLS client within SMPC causes a timeout that terminates the connection. Our experiments show that Gmail's SMTP servers have a TLS handshake timeout of 10 s, and Apache's default timeout is 60 s. In §VI-E, we show that this approach does not meet the TLS handshake timeout while our implementation consistently meets the timeout.

In the rest of the section, we describe our protocol, which consists of two parts: key exchange and message encryption/decryption.

B. Key exchange

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 [35]. 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—run the key derivation function to obtain the TLS session keys, as specified in the TLS standards [36, 37].

Step 1: Distributed generation of client randomness $\alpha \cdot G$. To generate the client randomness $\alpha \cdot G$ used in the `ClientHello` message without revealing α, N-for-1 Auth's TLS-in-SMPC protocol has each server sample a share of α and provide a corresponding share of $\alpha \cdot G$. Formally, the protocol works as follows:

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

Note that this step can be done in the offline phase.

Step 2: Distributed computation of key exchange result $\alpha\beta \cdot G$. As in Diffie-Hellman key exchange, the N-for-1 Auth servers need to jointly compute $\alpha\beta \cdot G$, which works as follows: each N-for-1 Auth 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 [38]. The protocol is as follows:

1) The N servers, within SMPC, derive the handshake secrets from $\alpha\beta \cdot G$ and the hash of the `ClientHello` and `ServerHello` messages.
2) The N servers compute the client handshake verification data inside SMPC, using $\alpha\beta \cdot G$ and the hash of the messages from `ClientHello` all the way to `ServerFinished`.
3) The N servers derive the application keys from the handshake secrets and the hash of the messages from `ClientHello` to `ServerFinished` inside SMPC. The application keys are later used to encrypt and decrypt the TLS payload inside SMPC.

We identify that the hashes of most of the TLS messages can be computed outside SMPC, which reduces the latency of the protocol. That is, the first N-for-1 Auth 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, informally, the TLS protocol is designed to prevent man-in-the-middle attackers, which permits us to share a number of TLS messages among these N servers.

Step 4: Validate the TLS server's response. Typically, in TLS, the TLS server sends a response containing its certificate, a signature over $\beta \cdot G$, and verification data, which the TLS client verifies. Performing this verification in SMPC is slow because (1) the certificate is in a format that is difficult to parse without revealing access patterns and (2) verifying signatures involves hashing and modular exponentiation operations, 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 server handshake key, which encrypts the response, is only designed to hide the TLS server's identity. This property is unnecessary in our system because, in N-for-1 Auth's setting, the servers must confirm the TLS server's identity. Consider the example from before in which the servers authenticate via Alice's email address. In N-for-1 Auth's TLS-in-SMPC protocol, the servers must confirm that they are sending an email to Alice's email service provider for security.

Revealing this key is known to be secure, as Bhargavan, Blanchet, and Kobeissi's symbolic analysis of TLS 1.3 [39] shows, meaning that it does not affect the other security properties of TLS. We offer a more detailed discussion on this in App. A and 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-for-1 Auth servers receive and

acknowledge all the handshake messages sent by the TLS server, namely all the messages from `ServerHello` to `ServerFinished` forwarded by the first server, the SMPC protocol reveals the TLS server handshake key to all the N servers. Each server uses the handshake key to decrypt the response and verifies the certificate, signature, and verification data within it.

Efficient implementation. The key exchange protocol mainly involves 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 expressed as a boolean circuit. Our insight to achieve efficiency here is to reflect the nature of these computations into a blend of SMPC protocols, each targeting one of them. Hence, we implement point additions with SPDZ using MASCOT [40] for the offline phase. The result of point additions are then transferred to AG-MPC [34] for key derivation, via a maliciously secure mixing protocol [41–43]. Both SPDZ and AG-MPC push most of the heavy cryptography operations into the offline phase, which helps reduce the online latency.

We choose MASCOT instead of more commonly used homomorphic-encryption-based preprocessing protocols such as Overdrive [44] and SPDZ-2 [33] because the latter do not work well for our setting. This is 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; MASCOT does not suffer from this limitation.

Another aspect of our efficient implementation is offloading computation into an offline preprocessing phase, which minimizes the online latency of the protocol. We discuss that in §VI-E, not doing so will result in a protocol that is too slow to meet the TLS handshake timeout.

C. Message encryption and decryption

The rest of the TLS-in-SMPC protocol involves message 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 [45, 46].

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

Cost of different ciphersuites in SMPC. The cost of TLS ciphersuites in SMPC has rarely been studied. Here, we implement the boolean circuits of three commonly used ciphersuites and measure their cost, as Tab. I shows.

The three ciphersuites we consider are AES-GCM-128, Chacha20-Poly1305, and AES-CCM-128. They are part of the TLS 1.3 standard and supported in many TLS 1.2 implementations. We compare their cost in terms of the total number of gates and the number of AND gates in boolean circuits. From Tab. I, we can see that CCM is the most efficient, which is due to the following reasons:

	# Total gates	**# AND gates**
AES-GCM-128	62214	21504
Chacha20-Poly1305	300430	91405
AES-CCM-128	58376	9984

Tab. I: Amortized cost per 128 bits in boolean circuits of commonly used TLS authenticated encryption.

- Although AES-CCM invokes AES twice as many times when compared with AES-GCM, AES-GCM involves operations in a large field $GF(2^{128})$, which results in more AND gates.
- Chacha20 and Poly1305 involve integer addition over 2^{32} and $2^{130}-5$, respectively, which are more expensive in boolean circuits than AES-GCM and AES-CCM.

Circuit implementation. We optimize these ciphers for fewer AND gates by synthesizing the circuits using Synopsys's Design Compiler and tools in TinyGarble [47], SCALE-MAMBA [48], and ZKCSP [49].

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 registration 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 could be different for each server.

Depending on the application, users may want to update their authentication methods, in which they would need to authenticate with the servers before updating.

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 this attack, 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 described in §III 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 commonly 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 $\mathcal{P}_i$.
5) For $\mathcal{P}_i$, if the user response matches the output that the server received in step 2, then $\mathcal{P}_i$ considers the user authenticated.

Registration. The registration 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 the 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 the share of the user's email address and the share of the user's secret.

Avoiding misclassification as spam. A common issue for email verification is that the email might be misclassified as spam. We can resolve this issue by following standard practices described below.

– *Sender Policy Framework (SPF).* Our system can follow the SPF standard [50], 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 who are allowed to send emails from this sender domain.
– *Domain Keys Identified Mail (DKIM).* The DKIM standard [51] requires each email to have a valid signature from the sender domain, under the RSA public way indicated in a `TXT` record. N-for-1 Auth can generate a RSA secret key in a distributed way [52–55] and secret-share the key among the servers. To send an email, the N servers reveals the email's hash from SMPC, and they jointly sign the email using the secret-shared secret key. To avoid leaking the passcode from this hash, the protocol adds random data in the email header.

Privacy. N-for-1 Auth's email authentication protocol reveals only minimal "metadata" to the N servers. Specifically, the N servers only need to know the email provider's mail gateway address instead of the full email address.[1] The full email address is secret-shared among the N servers and therefore hidden from them. The gateway address is needed because in the SMTP protocol, the sender (in this case the N servers) needs to contact the user's gateway server for them to receive the email.

B. *N-for-1 Auth SMS*

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

We leverage the fact that many mobile carriers, including AT&T [56], Sprint [57], and Verizon [58], 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. 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 them, which is difficult to achieve in traditional SMS authentication. We note that

[1] For example, many companies and schools use Google or Microsoft for email service on their domains. In this case, for a user with email address `A@B.com`, the N servers know neither `A` nor `B.com`, but only which email provider is used by `B.com`.

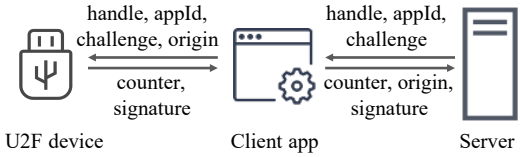


Fig. 5: Protocol of universal second factor (U2F).

the user's phone number is still revealed to the sender's carrier and the user's carrier, which is unavoidable.[2]

C. *N*-for-1 Auth U2F

Universal second factor (U2F) [59] 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 [60] and Google Titan [61]. 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 digital signature on a challenge requested by a server under a public key that the server knows. The U2F protocol consists of a registration phase and an authentication phase, described as follows.

In the registration phase, the U2F device generates an application-specific keypair and sends a key handle and 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 [60, 61], 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 registration phase.

If the user needs to be authenticated by N mutually distrusting servers, naturally the user needs to do N operations on the U2F device, such as tapping the button N times, which is burdensome. We want to reduce the amount of work the user performs to a single operation on the U2F device.

A straightforward approach to achieve is to have the servers generate a joint challenge which is then signed by the client. After the U2F device produces a signature over the jointly generated challenge, 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.

[2]To reduce exposure, the user can provide their mobile carrier to the servers. The servers can then use the same mobile carrier's API when available, so that only one mobile carrier sees the message.

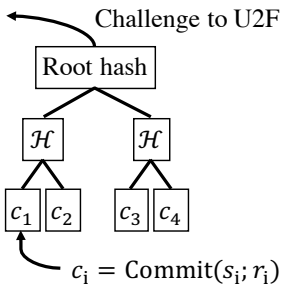


Fig. 6: The Merkle tree for U2F challenge generation.

An insecure strawman. We now describe an insecure strawman that does not use SMPC, 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 the U2F device and sends the signature to each server. Then, each server verifies the signature against the random challenge, 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 now impersonate the honest user by sending this signature to the honest 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.

A Merkle tree is a data structure in which a non-leaf node's value is a collision-resistant hash of the two children's values. If the client places the servers' randomness $s_1, s_2, ..., s_N$ into the leaf nodes, as Fig. 6 shows, then the value at the root of the tree, i.e., *the root hash*, 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 the U2F device 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 of the N-for-1 Auth servers 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.
3) The client requests the U2F device to sign the root hash as the challenge, as Fig. 5 shows; here, in the U2F request, the client app indicates the `origin` field to be an unique name that represents all the servers' identities to prevent man-in-the-middle attacks.
4) 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.
5) 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.

Registration. The registration protocol is as follows:

1) The client and the servers engage in the standard U2F registration protocol [59], 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, such as using different keys for each service and using the same attestation keys for many devices, 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 method we present is security questions. Thought it is simpler than the other ones that we have presented and it is similar to password authentication, we find it useful and present it for completeness.

Typically, security questions involve the user answering a series of personal questions that ideally only the user knows all of the answers to [62–65]. During registration, the user picks a series of questions and provides answers to them. Then, during authentication, the user is asked to answer these questions. If the answers match what the user provided during registration, the user considered authenticated.

In N-for-1 Auth, the questions and the hashes of the answers are secret-shared among the servers. The authentication protocol for N-for-1 Auth's security question is as follows:

1) The client establishes a TLS connection with each server and requests the security question for that user. The servers send over their shares of each question, and the client reconstructs the questions and displays them to the user.
2) Once the user inputs the answers, the client hashes the answers, secret-shares the hash into N shares, and sends the i-th share to server $\mathcal{P}_i$. Upon receiving the shares, the servers run a SMPC protocol that reconstructs the hash and compares it with the stored hash. If every answer the user provides matches what the servers have, the user is considered authenticated.

To defend against brute-force attacks in which a malicious attacker tries to guess the answers to security questions of honest users, N-for-1 Auth can implement standard rate-limiting mechanisms.

Registration. The registration protocol is as follows:

1) The client opens a connection with each of the N-for-1 Auth servers. For each security question, the user provides corresponding answer. The client secret-shares each security question and answer, and it sends the i-th share to server $\mathcal{P}_i$.
2) The client secret-shares the user secret s and sends the i-th share s_i to server $\mathcal{P}_i$.

Privacy and advantages over traditional security questions. The N-for-1 Auth security questions have desirable properties over traditional security questions. Previously, security questions have to avoid asking users for critical personal secrets, such as their SSN, because the user may feel uncomfortable to share such personal information with the website during registration. Hashing and other cryptographic techniques do not help 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, none of the servers can see the answer or the hash of

the answer, which may encourage a user to choose more sensitive questions and enter more sensitive answers that the user would otherwise be uncomfortable sharing. The privacy aspect that N-for-1 Auth's security question protocol offers also defends against offline attacks against hash functions since the hash is secret-shared and unknown to the attacker, even if they compromise some servers. In addition, if the user wants to minimize the exposure of sensitive security questions, or avoid online brute force attacks, the user can set other authentication factors or less-sensitive security questions as *prerequisites*. That is, only when the user authenticates against prerequisite factors can the user see the sensitive security questions. We note that preventing offline attacks of passwords is not new, but we discuss it for completeness.

V. Applications

Once the N servers have authenticated the user, they can perform some operations for the user using the user's secret that is secret-shared during registration, such as key recovery as in our motivating example in §II. To show the usefulness of N-for-1 Auth, we now describe four 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 registration 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 such as WhatsApp and Signal [4, 5], end-to-end encrypted file sharing apps such as Keybase [9], and cryptocurrencies such as Bitcoin and Ethereum [1, 2].

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. In N-for-1 Auth, the user secret-shares the signing key in the registration 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 [66–70]. An alternative solution is to use multisignatures [71], which N-for-1 Auth can also support, but this option in unavailable in certain cryptocurrencies [3] and may produce 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 [72], emp-toolkit [73] and wolfSSL [74] to implement N-for-1 Auth's TLS-in-SMPC protocol. We also implemented the online phase of elliptic-curve point additions within SMPC from scratch in C++.

B. Setup

We ran our experiments on `c5n.2xlarge` instances on EC2 with a 3.0 GHz 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.

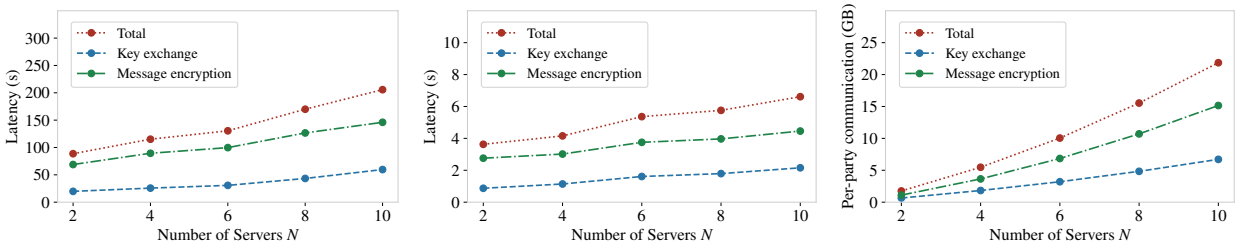
C. TLS-in-SMPC's performance

We measured the offline and online phases' latencies and the offline phase's communication for N servers to connect to an unmodified TLS server and show the results in Fig. 7. From the figure, we see that the total offline and online phase latencies and the offline communication grow roughly linearly to the number of parties.

We consider N from 2 to 10 in this experiment. In practice, companies that perform decentralized key recovery such as Curv [11] and Unbound Tech [12] currently use two mutually distrusting parties, and Keyless [15] uses three in their protocol. For all values of N that we tested, the TLS-in-SMPC protocol can consistently meet the TLS handshake timeout.

As Fig. 7 shows, the offline phase latency dominates, due to the large amount of communication needed. N-for-1 Auth's servers run the offline phase before the TLS connection is established in order to avoid incurring this extra overhead during the TLS connection. Malicious users could attempt to perform DoS attacks by wasting computation done in the offline phase. N-for-1 Auth can defend against such DoS attacks using well-studied techniques, such as proof-of-work or payment [75, 76].

Latency breakdown. In Tab. II we show a detailed breakdown of the offline and online phase latencies for N-for-1 Auth's TLS-in-SMPC protocol. From the table, we see that most of the computation is done in the offline phase, and the online phase has a small latency. We also



(a) Offline phase latency. (b) Online phase latency. (c) Offline phase communication per party.

Fig. 7: The overall online/offline phase latencies and the offline phase communication of the TLS-in-SMPC protocol for $N = 2, 4, 6, 8, 10$ servers when handling 1 KB of data in message encryption and decryption.

Component	Offline Phase Latency (s)			Online Phase Latency (s)		
	$N=3$	$N=5$	$N=10$	$N=3$	$N=5$	$N=10$
Key exchange (§III-B)	23.39	27.58	59.65	1.08	1.43	2.15
◇ Client randomness generation	0.30	0.30	0.30	—	—	—
◇ Key exchange result computation	0.06	0.15	0.51	0.35	0.47	0.63
◇ Key derivation	21.57	24.65	53.03	0.68	0.88	1.39
◇ Certificate validation	1.46	2.48	5.81	0.05	0.08	0.14
Message encryption/decryption (§III-C)	80.60	121.10	146.06	2.96	3.43	4.46
◇ AES encryption (64 blocks)	37.07	42.70	54.37	1.42	1.55	1.89
◇ GCM computation	43.53	78.40	91.69	1.54	1.88	2.57

Tab. II: Breakdown of the TLS-in-SMPC latencies when handling 1 KB of data.

see that key derivation dominates the latency for the TLS handshake phase. Therefore, if we run an SMPC protocol off the shelf and do not precompute the offline phase, from Tab. II we see that even for $N = 3$, the key exchange has a latency of 23.39 s and cannot meet a TLS handshake timeout of 10 s.

Asymptotic efficiency. We discuss the asymptotic efficiency of TLS-in-SMPC to help understand the linear growth patterns shown in Fig. 7 and Tab. II. Let N be the number of parties and λ be the security parameter. The offline phase latency is $O(\lambda(N^2 + C_1 N))$, where the N^2 term captures the cost of point additions in SPDZ, and the $C_1 N$ term captures the cost of key derivation and message encryption in AG-MPC. Since the constant C_1 is large, as Fig. 7 shows, empirically the offline phase latency grows linearly in N.

The online phase latency is $O(\lambda(N^3 + C_2 N^2 + C_3 N))$, where the $N^3 + C_2 N^2$ term captures the cost of running the SPDZ protocol, and the $C_3 N$ term captures the cost of running the AG-MPC protocol; similarly, since C_3 is large, empirically the online phase latency grows linearly in N, as Fig. 7 shows.

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. III. We now discuss the results in more detail.

	Offline Phase Latency (s)	Online Phase Latency (s)
Email (§IV-A)	128.12	4.24
SMS (§IV-B)	134.96	4.45
U2F (§IV-C)	—	0.03
Security Questions (§IV-D)	0.29	0.04

Tab. III: Latencies of N-for-1 Auth ($N = 10$).

Email/SMS. Using a standard authentication message of around 30 words, the N-for-1 Auth email protocol sends 352 bytes via TLS-in-SMPC (without DKIM signatures), and N-for-1 Auth's SMS authentication protocol, using AT&T's SMS API [56], sends 419 bytes via TLS-in-SMPC. The client's computation time is less than 1 ms.

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 a state-of-the-art maliciously secure MPC library, AG-MPC [34]. Since implementing TLS inside AG-MPC is a project in itself, we implemented a subset of TLS in AG-MPC, which offers a lower bound on the performance of TLS in AG-MPC. This lower bound is already considerably slower than N-for-1 Auth. With $N = 10$ servers, the offline latency is 10× slower and the online latency is 20× slower compared with N-for-1 Auth's TLS-in-SMPC implementation.

When using AG-MPC for the TLS handshake, computing $\alpha\beta \cdot G$ in key exchange involves expensive prime field operations. With $N = 10$ servers, these operations use 10^7 AND gates, which takes half an hour in the offline phase and 45 s in the online phase. In comparison, for these operations, N-for-1 Auth's implementation takes only 0.51 s for the offline phase and 0.63 s for the online phase.

F. Comparison with DECO

We also compare with DECO [45], a prior 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 for two parties and has extra leakage due to a different target setting.

During TLS handshake, DECO uses a customized protocol based on multiplicative-to-additive (MtA) [68] to add elliptic curve points, while N-for-1 Auth uses SPDZ. We are unaware of how to extend DECO's protocol to $N \geq 3$. We implemented DECO's point-addition protocol that they described in their paper and find that for $N = 2$, N-for-1 Auth's online phase latency is 0.25 s, whereas our implementation of DECO's protocol takes 0.31 s. The roughly 21% performance improvement comes from N-for-1 Auth's ability to push operations to the offline phase.

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, a 20% improvement.

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 [77, 78], which has been used in the PGP protocol in which individuals prove the identities of each other by signing each other's public key [79, 80]. Another idea is to make the authentication system transparent to the users. For example, blockchain-based authentication systems, such as IBM Verify Credentials [81], BlockStack [82], and Civic Wallet [83], and certificate/key transparency systems [30, 31, 84–87] have been deployed in the real world.

Collaborative generation of challenges. There has been work on generating secure secrets from collaboration among several mutually distrusting servers [18, 25]. However, their work 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 or forgotten. We note that these works are complementary to N-for-1 Auth, as these works can use N-for-1 Auth to allow users to store their passwords.

Decentralized storage of secrets. In industry, there are many companies that use decentralized trust to store user secrets, such as Curv [11], Partisia [88], Sepior [14], Unbound Tech [12], and Keyless [15]. 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.

TLS and SMPC. There are existing works using TLS with secure 2-party computation (S2PC), but they are in a prover-verifier setting in which the prover proves statements about information on the web. BlindCA [46] uses S2PC to inject packets in a TLS connection to allow the prover to prove to the certificate authority the ownership of a certain email address. However, it has the restriction that the prover possesses all of the secrets of the TLS connection, and all of the prover's traffic to the server must go through a proxy owned by the verifier. DECO [45] 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 limited to two parties and based on their intended settings, their protocols do not easily extend to N parties, while N-for-1 Auth supports an arbitrary number of parties.

In addition, a concurrent work [89] also enables running TLS in secure multiparty computation, and their technical design in this module is similar to ours, but [89] does not

propose or contribute authentication protocols 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 encrypted and cryptocurrencies space. In addition, within the TLS-in-SMPC protocol, we provide an end-to-end implementation compatible with an existing TLS library, and show that it works for *N*-for-1 Auth's authentication protocols.

End-to-end encryption systems. In academia, there are many works in end-to-end encrypted systems such as DepSky [90], Ghostor [76], M-Aegis [91], Metal [92], Mylar [93], Plutus [94], ShadowCrypt [95], Sieve [96], and SiRiUS [97]. End-to-end encryption has also been deployed in industry, such as Keybase [9], LINE [8], Signal [5], Telegram [6], and WhatsApp [4]. As mentioned in §V, these systems can use *N*-for-1 Auth to make the authentication process for key recovery more approachable for the end users.

Cryptocurrencies. There have also been works in cryptocurrencies such as Bitcoin [1], Ethereum [2], and Zcash [3]. As mentioned in §V, these systems can use *N*-for-1 Auth to make the process for obtaining signatures to authenticate transactions less burdensome for the end users.

OAuth. OAuth [98] is used for access delegation, which allows users to grant access to applications without giving the applications 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 legacy email services which do not support OAuth, and is therefore less general and flexible than *N*-for-1 Auth's TLS-in-SMPC construct.

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. Nevertheless, 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 [99, 100]. *N*-for-1 Auth can support identifiable abort by incorporating these SMPC protocols and standard identification techniques in its authentication protocols as described in §IV.

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] Satoshi 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] *Curv: The institutional standard for digital asset security*. https://www.curv.co.

[12] *Unbound tech: Secure cryptographic keys across any environment*. https://www.unboundtech.com/.

[13] *BitGo*. https://www.bitgo.com/.

[14] *Sepior: Threshold cryptographic key management solutions with MPC*. https://sepior.com.

[15] *Keyless: Zero-trust passwordless authentication*. https://keyless.io/.

[16] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. "Password-protected secret sharing". In: *CCS '11*.

[17] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. "Robust Password-Protected Secret Sharing". In: *ESORICS '16*.

[18] Philip D. MacKenzie, Thomas Shrimpton, and Markus Jakobsson. "Threshold Password-Authenticated Key Exchange". In: *CRYPTO '02*.

[19] Mario Di Raimondo and Rosario Gennaro. "Provably Secure Threshold Password-Authenticated Key Exchange". In: *EUROCRYPT '03*.

[20] Alma Whitten and J. Doug Tygar. “Why Johnny Can’t Encrypt: A Usability Evaluation of PGP 5.0”. In: *USENIX Security ’99*.

[21] Scott Ruoti, Jeff Andersen, Daniel Zappala, and Kent E. Seamons. “Why Johnny Still, Still Can’t Encrypt: Evaluating the Usability of a Modern PGP Client”. In: *arXiv:1510.08555 ’15*.

[22] Catherine S. Weir, Gary Douglas, Tim Richardson, and Mervyn A. Jack. “Usable security: User preferences for authentication methods in eBanking and the effects of experience”. In: *Interacting with Computers ’10*.

[23] Christina Braz and Jean-Marc Robert. “Security and usability: the case of the user authentication methods”. In: *International Conference of the Association Francophone d’Interaction Homme-Machine ’06*.

[24] *Proton Mail*. https://protonmail.com/.

[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] Andrew Chi-Chih Yao. “How to generate and exchange secrets”. In: *FOCS ’86*.

[27] Oded Goldreich, Silvio M. Micali, and Avi Wigderson. “How to play ANY mental game: A completeness theorem for protocols with honest majority”. In: *STOC ’87*.

[28] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness theorems for non-cryptographic fault-tolerant distributed computation”. In: *STOC ’88*.

[29] David Chaum, Crépeau. Claude, and Ivan Damgård. “Multiparty unconditionally secure protocols”. In: *STOC ’88*.

[30] *Certificate transparency*. https://www.certificate-transparency.org/.

[31] *Key transparency*. https://github.com/google/keytransparency.

[32] Yehuda 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.

[33] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. “Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits”. In: *ESORICS ’13*.

[34] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. “Global-scale secure multiparty computation”. In: *CCS ’17*.

[35] Whitfield Diffie and Martin E. Hellman. “New directions in cryptography”. In: *TIT ’76*.

[36] *The transport layer security (TLS) protocol version 1.3*. https://tools.ietf.org/html/rfc8446.

[37] *The illustrated TLS 1.3 connection*. https://tls13.ulfheim.net/.

[38] Hugo Krawczyk. “Cryptographic extraction and key derivation: The HKDF scheme”. In: *CRYPTO ’10*.

[39] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. “Verified models and reference implementations for the TLS 1.3 standard candidate”. In: *S&P ’17*.

[40] Marcel Keller and Emmanuela Orsini. “MASCOT: Faster malicious arithmetic secure computation with oblivious transfer”. In: *CCS ’16*.

[41] Dragos Rotaru and Tim Wood. “MArBled circuits: Mixing arithmetic and boolean circuits with active security”. In: *INDOCRYPT ’19*.

[42] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. “Zaphod: Efficiently combining LSSS and garbled circuits in SCALE”. In: *WAHC ’19*.

[43] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. “Improved primitives for MPC over mixed arithmetic-binary circuits”. In: *CRYPTO ’20*.

[44] Marcel Keller, Valerio Pastro, and Dragos Rotaru. “Overdrive: Making SPDZ great again”. In: *EUROCRYPT ’18*.

[45] Fan Zhang, Sai Krishna Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. “DECO: Liberating web data using decentralized oracles for TLS”. In: *CCS ’20*.

[46] Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and abhi shelat. “Blind certificate authorities”. In: *S&P’ 19*.

[47] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. “TinyGarble: Highly compressed and scalable sequential garbled circuits”. In: *S&P ’15*.

[48] *SCALE-MAMBA*. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[49] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. “Zero-knowledge contingent payments revisited: Attacks and payments for services”. In: *CCS ’17*.

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

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

[52] Dan Boneh and Matthew Franklin. “Efficient generation of shared RSA keys”. In: *CRYPTO ’97*.

[53] Yair Frankel, Philip D. MacKenzie, and Moti Yung. “Robust efficient distributed RSA-key generation”. In: *STOC ’98*.

[54] Carmit Hazay, Gert L. Mikkelsen, Tal Rabin, and Tomas Toft. “Efficient RSA key generation and threshold Paillier in the two-party setting”. In: *CT-RSA ’12*.

[55] Megan Chen, Ran Cohen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, and Abhi Shelat. “Multiparty generation of an RSA modulus”. In: *CRYPTO ’20*.

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

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

[58] *Verizon’s enterprise messaging access gateway*. https://ess.emag.vzw.com/emag/login.

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

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

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

[62] Mike Just and David Aspinall. “Personal choice and challenge questions: A security and usability assessment”. In: *SOUPS ’09*.

[63] Stuart E. Schechter, A. J. Bernheim Brush, and Serge Egelman. "It's no secret: Measuring the security and reliability of authentication via 'secret' questions". In: *S&P '09*.

[64] Ariel Rabkin. "Personal knowledge questions for fallback authentication: Security questions in the era of Facebook". In: *SOUPS '08*.

[65] Michael Toomim, Xianhang Zhang, James Fogarty, and James A. Landay. "Access control by testing for shared knowledge". In: *CHI '08*.

[66] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. "Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security". In: *ACNS '16*.

[67] Dan Boneh, Rosario Gennaro, and Steven Goldfeder. "Using level-1 homomorphic encryption to improve threshold DSA signatures for Bitcoin wallet security". In: *LATINCRYPT '17*.

[68] Rosario Gennaro and Steven Goldfeder. "Fast multiparty threshold ECDSA with fast trustless setup". In: *CCS '18*.

[69] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. "Threshold ECDSA from ECDSA assumptions: The multiparty case". In: *S&P '19*.

[70] Rosario Gennaro and Steven Goldfeder. "One round threshold ECDSA with identifiable abort". In: *IACR ePrint 2020/540*.

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

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

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

[74] *wolfSSL Embedded SSL/TLS Library — Now Supporting TLS 1.3*. https://https://www.wolfssl.com/.

[75] David Lazar and Nickolai Zeldovich. "Alpenhorn: Bootstrapping secure communication without leaking metadata". In: *OSDI '16*.

[76] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. "Ghostor: Toward a secure data-sharing system from decentralized trust". In: *NSDI '20*.

[77] Raphael Yahalom, Birgit Klein, and Thomas Beth. "Trust relationships in secure systems: A distributed authentication perspective". In: *S&P '93*.

[78] Thomas Beth, Malte Borcherding, and Birgit Klein. "Valuation of trust in open networks". In: *ESORICS '94*.

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

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

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

[82] *BlockStack*. https://www.blockstack.org/.

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

[84] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. "CONIKS: Bringing key transparency to end users". In: *SEC '15*.

[85] Joseph Bonneau. "EthIKS: Using Ethereum to audit a CONIKS key transparency log". In: *FC '16*.

[86] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. "Transparency logs via append-only authenticated dictionaries". In: *CCS '19*.

[87] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. "Certificate transparency with privacy". In: *PETS '17*.

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

[89] Damiano Abram, Ivan Damgård, Peter Scholl, and Sven Trieflinger. *Oblivious TLS via Multi-Party Computation*. Cryptology ePrint Archive, Report 2021/318. https://eprint.iacr.org/2021/318.

[90] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. "DepSky: Dependable and secure storage in a cloud-of-clouds". In: *EuroSys '11*.

[91] Billy Lau, Simon Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. "Mimesis Aegis: A mimicry privacy shield: A system's approach to data privacy on public cloud". In: *SEC '14*.

[92] Weikeng Chen and Raluca Ada Popa. "Metal: A metadata-hiding file-sharing system". In: *NDSS '20*.

[93] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. "Building web applications on top of encrypted data using Mylar". In: *NSDI '14*.

[94] Mahesh Kallahalla, Erik Riedel, Ram P. Swaminathan, Qian Wang, and Kevin Fu. "Plutus: Scalable secure file sharing on untrusted storage". In: *FAST '03*.

[95] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. "ShadowCrypt: Encrypted web applications for everyone". In: *CCS '14*.

[96] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. "Sieve: Cryptographically enforced access control for user data in untrusted clouds". In: *NSDI '16*.

[97] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. "SiRiUS: Securing remote untrusted storage". In: *NDSS '03*.

[98] *OAuth*. https://www.oauth.net/.

[99] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. "Efficient constant-round MPC with identifiable abort and public verifiability". In: *CRYPTO '20*.

[100] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. "Secure multi-party computation with identifiable abort". In: *CRYPTO '14*.

[101] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. "The wonderful world of global random oracles". In: *EUROCRYPT '18*.

[102] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. "Practical UC security with a global random oracle". In: *CCS '14*.

[103] *ProVerif: Cryptographic protocol verifier in the formal model*. https://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[104] *RefTLS*. https://github.com/Inria-Prosecco/reftls/blob/master/pv/tls-lib.pvl.
[105] Ran Canetti. "Universally composable security: A new paradigm for cryptographic protocols". In: *FOCS '01*.
[106] Hemanta K. Maji, Manoj Prabhakaran, and Mike 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 [32], 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 [101, 102].

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 [36], 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.

Indeed, symbolic analysis of TLS shows that if this server does not require anonymity, it is safe to reveal the server handshake key. Bhargavan, Blanchet, and Kobeissi [39] conduct a symbolic analysis of TLS 1.3 using ProVerif, an automatic cryptographic protocol verifier [103]. They prove secrecy, forward secrecy, authentication, and replay protection in the presence of a more powerful attacker that sees all the handshake keys before the TLS server sends any message encrypted by these keys.

More specifically, in their proofs which have been open-sourced [104] and can be verified by ProVerif, the handshake keys are sent out to the network in plaintext, and the handshake responses of both the client and the server are unencrypted.

This result confirms that when server anonymity is not needed, revealing the server handshake key does not affect the rest of the properties provided by TLS that is needed for a secure communication channel.

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 $\mathcal{F}_{\mathrm{SMT}}$ defined in [105].

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 $(\texttt{GetSrvHandshakeKey}, sid)$ 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 $(\texttt{reveal}, sid, key, iv)$ where key and iv are the server 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 [106]. $\mathcal{F}_{\mathsf{SMPC}}$ 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 $f(x_1, x_2, ..., x_N) \rightarrow (y_1, y_2, ..., y_N)$ 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 model the random oracle as a restricted programmable random oracle, 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 $(\texttt{GetSrvHandshakeKey}, sid)$ to $\mathcal{F}_{\mathsf{TLS}}$ to obtain the server handshake key and IV.
   b) $\mathcal{S}$ simulates the computation of the server handshake key in SMPC by pretending that the SMPC output is the server handshake key. 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, by pretending that the SMPC output is the correct ciphertext of the client verification data, which is taken directly from the TLS messages provided by $\mathcal{F}_{\mathsf{TLS}}$.

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.