2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179481

MPCAuth: Multi-factor Authentication for Distributed-trust Systems

Sijun Tan, Weikeng Chen, Ryan Deng, and Raluca Ada Popa
University of California, Berkeley
{sijuntan, weikengchen, rdeng2614, raluca.popa}@berkeley.edu

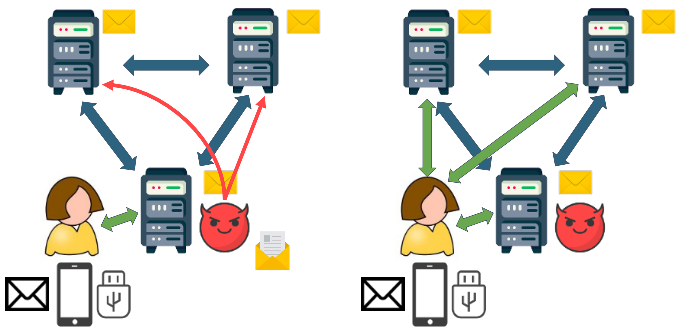
***Abstract*—Systems with distributed trust have attracted growing research attention and seen increasing industry adoptions. In these systems, critical secrets are distributed across N servers, and computations are performed privately using secure multi-party computation (SMPC). Authentication for these distributed-trust systems faces two challenges. The first challenge is ease-of-use. Namely, how can an authentication protocol maintain its user experience without sacrificing security? To avoid a central point of attack, a client needs to authenticate to each server separately. However, this would require the client to authenticate N times for each authentication factor, which greatly hampers usability. The second challenge is privacy, as the client's sensitive profiles are now exposed to all N servers under different trust domains, which creates N times the attack surface for the profile data.**

We present MPCAuth, a multi-factor authentication system for distributed-trust applications that address both challenges. Our system enables a client to authenticate to N servers independently with the work of only one authentication. In addition, our system is profile hiding, meaning that the client's authentication profiles such as her email username, phone number, passwords, and biometric features are not revealed unless all servers are compromised. We propose secure and practical protocols for an array of widely adopted authentication factors, including email passcodes, SMS messages, U2F, security questions/passwords, and biometrics. Our system finds practical applications in the space of cryptocurrency custody and collaborative machine learning, and benefits future adoptions of distributed-trust applications.

I. Introduction

Decentralizing trust has become a fundamental principle in designing modern computing systems and security applications. In the past decade, we have seen tremendous interest from both academia and industry in systems that rely on distributed trust. Some examples include: cryptocurrency custody (e.g. Fireblocks [16], Curv [11], MPCVault [22]), collaborative learning (e.g. Meta [23], Ant Group [3], Cerebro [90], Senate [75]), encrypted search systems (e.g. Dory [56]), and aggregate statistics (e.g. Prio [55]).

In many of these systems, clients (users) possess critical secrets that they want to store securely and later perform computations on them privately. To do so, they distribute their secrets to N servers hosted in different trust domains so that none of the servers can access the secrets. Some of these servers, but not all, can be compromised by a malicious attacker. These servers later jointly perform private computations using cryptographic tools such as secure multi-party computation (SMPC) [88, 62, 42, 53]. This construction avoids a central point of attack as no servers can see the secrets.



(a) If the client authenticates to a master server that other servers trust, then an attacker controlling it could impersonate the client to recover the secrets.

(b) If the client authenticates to each server individually, she needs to perform the work $N \times M$ times for N servers and M factors.

Fig. 1: The dilemma between ease-of-use and security for authentication under distributed trust. The closed envelopes are the client's secrets secret shared on the servers.

One concrete application of such systems is cryptocurrency custody. The critical secrets in this space are the signing keys, which represent ownership of digital assets (e.g. cryptocurrencies, Defi tokens, NFTs) that could be worth millions of dollars today. If an attacker compromises the keys, he can steal the client's valuable assets. Cryptocurrency custody companies such as Curv [11] and Fireblocks [16] offer services to store clients' signing keys as secret shares on multiple servers. When a client wants to authorize a transaction, the servers jointly produce a digital signature with the secret-shared signing key using SMPC.

Collaborative training/analytics on sensitive organizational data provides another application. Organizations such as banks or hospitals want to collaborate on each other's data to jointly train a machine learning (ML) model or perform SQL analytics. However, these data often contain sensitive information that organizations do not want to disclose. Major tech companies (e.g. Meta [23], Ant group [3]) as well as confidential computing startups (e.g. Huakong Tsingjiao [18], Basebit [5]) provide services for clients to distribute their data as secret-shares to multiple servers, and then use SMPC to jointly train ML models or perform SQL analytics on the

DOI 10.1109/SP46215.2023.00190

secret-shared data.

One critical component of these services is authentication. A client must first authenticate to the N servers before she can authorize cryptocurrency transactions or train machine learning models. To avoid a central point of attack with authentication, the client must use multiple factors of authentication, say M. Two challenges naturally arise in this setting.

The first challenge is ease-of-use. Namely, how can we retain the ease of use of a traditional system while keeping the security guarantees of a distributed-trust application? One strawman could be for the client to authenticate to one master server that the other servers trust. However, this approach breaks distributed trust because, if the master server is compromised, an attacker can authenticate as the client with all the N servers and issue unauthorized transactions with the client's secrets. To maintain security, the client can authenticate to N servers independently. However, for common factors such as email passcodes, SMS messages, and U2F, the client now has to perform N times the work – reading N emails, looking at N messages, and tapping the U2F device N times. This results in a client work of $N \times M$ authentication factors. Even for $N = 2$ in which there is only one additional email and text message, the user experience is completely different and adds friction to a space already plagued by usability issues (e.g., "Why Johnny can't encrypt?" [85, 78]). Many academic works [84, 47] reaffirm the importance of the consistency of user experience and minimizing user effort for better adoption. Another strawman approach one might think of is to build a client app that automatically performs the N authentication for the clients. These solutions work for some authentication factors such as passwords and biometrics, but fall short of other factors. We explain in Section VI why they are not sufficient enough to address the problem.

The second challenge is privacy. Authentication factors such as email, SMS messages, and server-based biometrics can leak sensitive information about the client – her email username, phone number, and biometric features – to the servers. Whereas in a traditional centralized trust system, one (logical) server learns the client's profile information, in a distributed trust setting, there are additional $N - 1$ servers in different trust domains also learning this private information. Essentially, this is N times the attack surface for the client's identity data.

A. MPCAuth

In this work, we propose MPCAuth, an authentication system for distributed-trust applications that simultaneously addresses both challenges. Specifically, MPCAuth enables a client to authenticate to N servers *independently* by doing only the work of authenticating to *one*. MPCAuth also hides the client's authentication profile, such as her email username, phone number, and biometric information from the N servers.

MPCAuth supports many authentication factors that clients are accustomed to, including email, SMS, U2F, security questions/passwords, and biometrics, as discussed in Section IV. For all supported factors, MPCAuth offers the same security properties as the underlying authentication protocols even in the presence of a malicious adversary that can compromise up to $N - 1$ of the N servers.

B. Summary of techniques

We now summarize the insights and techniques behind MPCAuth.

One logical server, N physical servers. In a typical authentication workflow, the server sends a challenge to the client, to which the client responds. For example, in the email/SMS authentication case, the challenge is a one-time passcode sent to the client's email address or phone number, and a client with the credential can respond with the correct passcode. In email/SMS authentication, the N servers need to talk to an email/SMS server that is responsible for distributing the email/SMS message to the client. MPCAuth enables these N physical servers to jointly act as a single logical server, and generate one collective challenge to the client. Since servers do not trust each other, they need to verify the client response individually with their own secret challenge. The question is, how can the servers generate a collective secret challenge, and send *one* email or SMS to the user with this secret challenge inside without any server learning this secret? While this formulation seems reminiscent of secure multi-party computation, it does not answer the questions of what server can send the actual email or SMS over the network given the distributed trust, and how to run the SMTP protocol or the SMS protocol in a secure distributed trust way so that the user's email/SMS provider accepts this email/SMS.

The insight is that the N servers in MPCAuth run *in SMPC a logical server* (instead of a function) within which they: (1) generate a joint challenge, (2) run the SMTP email or SMS protocol for sending email or SMS, and (3) run the TLS protocol for encrypting the channel of communication for the email or SMS – all these operations in SMPC. MPCAuth's SMPC is maliciously-secure tolerating the compromise of $N-1$ servers. What comes out of the SMPC is TLS-encrypted and authenticated traffic. Since this content is protected by TLS, any one of the servers can simply forward it to the email provider.

Efficient TLS-encrypted channel within SMPC. TLS is an intricate protocol that involves many cryptographic operations. If we run the TLS endpoint using an off-to-shelf maliciously secure SMPC library, our experiments in Section V-E show that it would be at least $8\times$ more expensive than our protocol. We designed our TLS-in-SMPC protocol and optimized its efficiency with a number of insights based on the TLS protocol itself, and integrated it with the `wolfSSL` library.

Hiding client profiles inside SMPC. In a traditional centralized system, hiding a client's profile from the server is difficult – the server needs the client's email address or phone number to send emails or SMS messages to him. In a distributed-trust setting, however, we leverage the preexisting distributed-trust infrastructure to achieve profile hiding. MPCAuth shards the user's information, such as email username and phone number, across the N servers. When the N servers perform SMPC to send email or SMS to the user, they also assemble the email or phone number of the user inside SMPC and encrypt that information with TLS. The server sending the network packets only needs to know the email provider or mobile carrier. None of the servers sees the profiles in plaintext during the whole process. We give our concrete definition of MPCAuth's profile-hiding property in Definition II.3, and discuss how each factor achieves the desired property in Section IV.

Secure and practical constructions of common authentication factors. Beyond email/SMS, MPCAuth additionally supports U2F, security questions, and server-side biometrics. These are all widely-adopted authentication factors on the web today, and all of them encounter either usability or privacy issues under the distributed-trust setting. For U2F, the client needs to tap the U2F button N times to respond to each server's challenges. And for security questions and server-side biometrics, the answers to the questions and biometric features are exposed to all servers. While each factor has its unique set of challenges, one core challenge for all these factors is to ensure their security properties are not weakened under MPCAuth's threat model. More specifically, replay attacks are a common threat to authentication protocols. Under our threat model, when a malicious server receives a response from the client, this server may attempt to use the response to impersonate the client and authenticate with the other servers. We formally define MPCAuth's security goal in Section II-D and prove our constructions' security in Section C.

C. Summary of contributions

Here are the main contributions of this paper:

- We present a novel multi-factor authentication system for distributed-trust applications. Our system enables a client to authenticate independently to N servers by doing the work of one authentication.
- We design *secure*, *practical*, and *profile-hiding* constructions for a comprehensive set of authentication factors. As a subpart of this, we propose TLS-in-SMPC, an efficient protocol to establish TLS connections in SMPC, as an important building block to some of our supported factors.
- We provide efficient implementations and experimental evaluations of all our authentication protocols, and demonstrate their practicality. In particular, our TLS-in-SMPC protocol is $8\times$ faster than a naive implementation without our application-specific optimizations. As a result, when $N = 5$, our email authentication protocol only takes 1.81 seconds to perform the TLS handshake and send the email payload, which is efficient enough to avoid a TLS timeout and successfully communicate with an unmodified TLS email server.

II. System Overview

A. System setup and trust model

A deployment of MPCAuth consists of a set of N servers and many clients. These servers jointly host an application (e.g. cryptocurrency custody, collaborative learning) that requires distributed trust. A client needs to authenticate to these servers to use the service. These N servers are hosted under different trust domains.

Our system operates under the multi-factor authentication (MFA) model. Each client selects multiple authentication factors for their account, with a minimum of two distinct factors required for registration. Typically the first factor is password; MPCAuth supports this factor but gives the client flexibility to choose other passwordless authentication factors as their first factor. Additionally, clients must provide a unique identifier (e.g., username) to MPCAuth. We note that each factor's protocol within MPCAuth operates independently from the others, enabling the individual protocols to be used separately in authentication models beyond MFA.

Since MPCAuth operates in an MFA model, trust is distributed across multiple authentication providers. MPCAuth can therefore tolerate untrusted authentication providers that try to impersonate the clients, so long as they do not all collude. To enable account recovery, the client needs to register additional factors so that when she loses some factor, she could resort to other factors to recover her account. We discuss account recovery in greater detail in Section VI.

Each client can download a stateless client application or use a web client to participate in these protocols. This minimalist client app does not retain secrets or demand intrusive permissions to data in other applications such as a client's emails or text messages; it simply serves as an interface between the client and the servers.

Supported authentication factors. MPCAuth supports the following authentication factors:

- *Email/SMS:* The N servers jointly send *one* email/SMS to the client's address with a passcode. During authentication, the servers expect the client to enter this passcode.
- *U2F:* The N servers jointly initiate *one* request to a U2F device. During authentication, the servers expect a signature, signed by the U2F device, over this request.
- *Security questions/passwords:* The client initially provides a series of questions and answers to the servers. During authentication, the servers expect answers consistent with those that are set initially. Passwords are a special case of security questions.
- *Biometrics:* The client scans her biometrics and sends its feature vector to the servers. During authentication, the client scans her biometrics again to obtain a new feature vector. The servers expect this new vector to be close to the one stored initially.

Authentication workflow. The authentication workflow is as follows:

- *Enrollment.* A client contacts the N servers through the client app and provides a number of authentication factors. If the client is new, the N servers ask the client to provide a unique identifier and set up an account with that identifier. For each factor, the N servers run MPCAuth's authentication protocol of that factor. If the client passes all factors' protocol checks, the servers store the relevant information of that factor under the client's account for future authentication.

- *Authentication*. The client contacts the N servers through the client app, provides his unique identifier, and chooses at least two authentication factors. For each factor, the N servers invoke MPCAuth's authentication protocol of that factor. If the client passes the protocol's check, the N servers consider the client authenticated.

B. Practical use cases for MPCAuth

We analyze a few real-world deployments of distributed-trust applications and show how MPCAuth could fit into their trust model to provide authentication.

- Curv [11], Fireblocks [16], and MPCVault [22] are companies that offer digital asset custody services. These services protect the client's asset by secret-sharing its signing key among N servers. When the client authorizes a transaction, the servers use SMPC to jointly produce a digital signature with the signing key. These companies operate under different trust models. Some service providers take full custody of the client's digital assets. They maintain all servers, and distribute client secrets to these servers to avoid a central point of attack. In this case, the clients do have to trust the service provider not to recover their signing keys. Other providers let some external institutions maintain a subset of servers so that there are multiple trust domains. In either case, a client needs to authenticate to these servers before she could store her assets on them and later authorize transactions. These services could leverage MPCAuth to provide multi-factor authentication.
- Meta [23], Ant group [3], Huakong Tsingjiao [18], and Basebit [5] are companies that provide collaborative learning services. Typically, institutions that want to collaborate on data would each maintain their own server that has the SMPC software installed. The service provider may provide a server that acts as a trusted third party or directly participates in the SMPC protocol. Clients, in this case, would be the employees of these institutions that are responsible for uploading data and authorizing collaborative training/analytics. These clients need to authenticate to all trust domains to prevent a central point of attack, which MPCAuth could help.

Beyond these existing use cases, we expect other emerging distributed-trust applications to benefit from MPCAuth in the upcoming years.

C. Threat model

In MPCAuth's threat model, there are N servers hosting a distributed-trust application, multiple honest clients that use the application, and a malicious attacker. The malicious attacker tries to impersonate these honest clients. The malicious attacker can compromise up to $N-1$ of the N servers, but at least one server is honest and uncompromised. The attacker can observe and modify all inputs, states, and network traffic of the compromised servers. The attacker does not control the honest client, but it can observe all network traffic between the honest client and compromised servers. In email/SMS authentication, there is an additional TLS server hosted by the authenticator that the N servers talk to. We assume the TLS server is uncompromised, and the attacker does not control or collude with the authenticator.

We assume that an honest client uses an uncompromised client app. The client app does not carry any secrets, but it must be obtained from a trusted source. The client app either has hard coded the TLS certificates of the N servers, or obtains them from a trusted certificate authority or a transparency ledger [9, 21]. This enables clients and servers to connect to one another securely using the TLS protocol. As guaranteed by the security of the TLS protocol, the attacker can not observe the network traffic between an honest client and an honest server.

D. Security goals

Since MPCAuth is built on top of existing authentication factors, it maintains the same security properties that the existing factors provide under this threat model. For each authentication factor, MPCAuth's security goal is to enable an honest client that holds credential w of that factor to successfully authenticate to the servers, and reject a malicious attacker who does not hold the credential, even if the attacker compromises $N-1$ out of N servers.

Definition II.1 (Security of an authentication protocol)**.** Informally, an N-wise authentication protocol Π for a factor is said to be secure if it satisfies the following properties:

- *Correctness*: Assuming all N servers are honest, Π accepts an honest client who holds credential w of that factor with overwhelming probability.
- *Soundness*: Given a malicious attacker who compromises $N-1$ servers, but does not hold credential w. Π rejects with overwhelming probability.

In particular, for soundness to hold, the authentication protocol needs to protect against replay attacks. A replay attack happens if the client authenticates to the servers with the same response. Under MPCAuth's threat model, if a malicious server that receives the client's response reaches the honest server faster than the client, then the honest server will consider the malicious server authenticated instead of the client. In this case, the malicious server can reconstruct the shares and obtain the client's critical secrets. To prevent replay attacks, the client needs to send each server a different response.

We give our formal definition of an authentication protocol, the security of the protocol, as well as the security proofs in Section C. Our system does not handle denial-of-service (DoS) attacks, but we discuss how to extend our system to handle these attacks in Section VI.

The security of MPCAuth, in addition, relies on the security of the TLS-in-SMPC protocol. Formally, we define in Section B 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 [70]:

Definition II.2 (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 $I \subseteq \{1,2,...,N\}$,

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

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

We go over our TLS-in-SMPC protocol, in Section III, present a formal description in Fig. 6, and prove that it securely realizes $\mathcal{F}_{\mathsf{TLS}}$ in Section B.

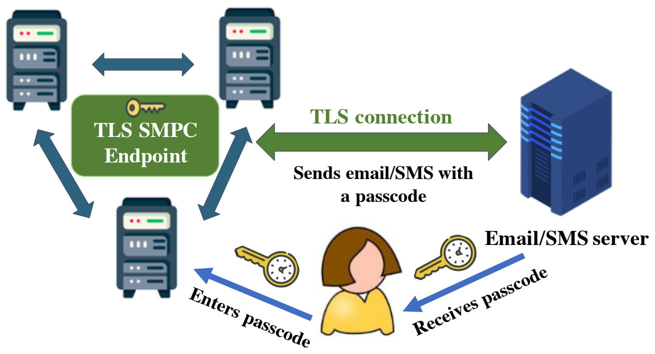


Fig. 2: TLS-in-SMPC's application to email/SMS authentication. The N servers jointly create a TLS endpoint connecting to the client's email/SMS server. Then, the N servers generate a passcode and send it over the TLS channel; the client who receives the passcode from her email/SMS server enters it on her client app, which forwards it back to the N servers.

E. Hiding authentication profiles

Additionally, a client may want to hide her authentication profiles from the N servers. Authentication factors of different types have distinct privacy goals to meet. For knowledge (e.g. security questions/passwords) or inherence factors (e.g. biometrics), profile-hiding means that the client hides the knowledge/inherence from the servers. For authentication factors involving possession (e.g., email, SMS, U2F), the factors are considered profile-hiding if the servers cannot associate the client's application account with their owned device or object. To be more concrete, we define the profile-hiding property we hope to achieve for each authentication factor below.

Definition II.3 (Profile hiding)**.** MPCAuth's authentication protocol is profile-hiding if, given an attacker that compromises $N-1$ out of N servers:

- *Email*: The attacker does not learn the email username. It may learn the email domain name of the user.
- *SMS*: The attacker does not learn the client's phone number. It may learn the mobile carrier of the client.
- *U2F*: The attacker cannot associate the U2F's public key to the client.
- *Security questions & passwords*: The attacker does not learn the contents of the security questions as well as their answers (passwords).
- *Biometric*: The attacker does not learn the biometric features of the client.

We require the client to provide a unique identifier to the N servers that all servers store as plaintext. To preserve privacy, this unique identifier should be made different from the user's email address or phone number. Typically, this unique identifier is a username the client enters when setting up an account. From now on, we will use 'username' to refer to this identifier. The servers use this username as the index for all the client's account information. One question is how a client can recover her account if she forgets his username. The client cannot resort to her authentication factors since they are indexed by this username and stored privately on the servers. In Section VI, we discuss this problem in greater detail and propose some recovery mechanisms.

While MPCAuth's profile-hiding protocols provide enhanced privacy, there are still several limitations to consider. First, for email and SMS authentication, our our system does not hide all profile information – it leaks the client's email service provider and mobile carrier to the N servers. We discuss these leakages in greater detail in Section IV. Second, our system does not hide the IP addresses of the client that authenticates to the servers. Finally, our protocols do not hide access patterns of the clients, which could be used to infer sensitive information about client behavior. Academic work, such as [82, 58, 54], explores using ORAM on top of SMPC to hide access patterns for distributed-trust applications. We leave it as future work to extend MPCAuth to address access pattern leakage.

III. Building Block: TLS in SMPC

One important building block for our authentication protocols is TLS-in-SMPC, which enables N servers to jointly establish a secure TLS connection with an *unmodified* TLS server. This tool is later used to construct our email and SMS authentication protocols. In our work, we choose to build upon the latest TLS 1.3 [76] protocol.

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. The insight is for these N servers to jointly create a TLS client endpoint within SMPC that can communicate with the TLS server over a TLS connection.

As Figure 2 shows, the N servers run a TLS client within SMPC, which establishes a TLS connection with the unmodified TLS server. The TLS session keys are only known by the TLS server. The N servers that act as the TLS client jointly hold the session keys within SMPC, but each of them does not have access to the keys. Hence, the N servers must work together to participate in this TLS connection.

In the following section, we give a step-by-step breakdown of the TLS-in-SMPC protocol. We also provide a formal description of the protocol in Figure 6, and prove its security in Section B.

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

Comparisons with related work. There are a few prior and concurrent works that also propose constructions involving TLS and SMPC. We highlight our differences with them in Section VII.

B. Notations and definitions

We use $\mathcal{P}$ to denote the logical TLS client that the N servers act as, and $\mathcal{P}_i$ to denote the i-th server. We use $\mathcal{S}$ to denote the TLS server. We use $[N]$ to denote the set $\{1, ..., N\}$, $\mathbb{Z}_p^+$ to denote the set $\{1, 2, ..., p-1\}$ where p is a prime. We use $[\![x]\!]_i$ to denotes $\mathcal{P}_i$'s secret-share of x, and λ denotes the security parameter.

MPCAuth's construction relies on several cryptographic primitives. We use (1) $\mathsf{PRF}(\mathsf{sk}, x)$ to denote a pseudorandom function with sk as its seed and x as inputs, (2) CRH to denote a collision-resistant hash function and $\mathsf{Hash}(x)$ to denote its hash on an input x, (3) $\mathsf{Sig} := (\mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify})$ to denote a digital signature algorithm, and (4) $\mathsf{Commit}(m, r)$ to denote a commitment scheme that produces a commitment of message m with randomness r. Their detailed definitions are described in Section A.

C. TLS handshake in SMPC

TLS 1.3 uses Diffie-Hellman key exchange to obtain a shared DH key, which is then used by both the client and the server to derive session keys using the key derivation function. In this section, we discuss how MPCAuth's TLS-in-SMPC protocol handles 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 [57]. Let G be the generator of a suitable elliptic curve of prime order p. The key exchange consists of three steps:

1) In the `ClientHello` message, the TLS client samples $\alpha \leftarrow \mathbb{Z}_p^+$ and sends $\alpha \cdot G$ to the TLS server.
2) In the `ServerHello` message, the TLS server samples $\beta \leftarrow \mathbb{Z}_p^+$ and sends $\beta \cdot G$ to the TLS client.
3) The TLS client and server compute $\alpha\beta \cdot G$ and—with other information—derive the TLS session keys, as specified in the TLS standards [31, 20].

In our protocol, the N servers act as the TLS client $\mathcal{P}$ which establishes connections with a TLS server $\mathcal{S}$. The TLS handshake protocol proceeds as follows:

Step 1: Distributed generation of key share $\alpha \cdot G$. The N servers need to jointly generate the `ClientHello` message and send it to $\mathcal{S}$. To generate the key share $\alpha \cdot G$ used in the `ClientHello` message without revealing α, each server $\mathcal{P}_i$ randomly samples α_i, and they jointly construct a corresponding key share $\alpha \cdot G$ as follows:

1) $\forall i \in [N]$, the i-th server $\mathcal{P}_i$ samples $\alpha_i \leftarrow \mathbb{Z}_p^+$, computes $\alpha_i \cdot G$ locally, and revealing $\alpha_i \cdot G$ to the other parties.
2) $\mathcal{P}_1$ receives $a_i G$ from all other parties, computes $\alpha \cdot G = \sum_{i=1}^{N} \alpha_i \cdot G$ and sends it to the TLS server.

Step 2: Distributed computation of key exchange result $\alpha\beta \cdot G$. After receiving the `ServerHello` message $\beta \cdot G$ from the TLS server, the N servers need to jointly compute the DH key $\alpha\beta \cdot G$, which works as follows:

- Each party $\mathcal{P}_i$ computes $\alpha_i(\beta G)$ locally, and then the SMPC engine takes $\alpha_i(\beta G)$ as input from $\mathcal{P}_i$ to compute $\alpha\beta \cdot G = \sum_{i=1}^{n} \alpha_i(\beta G)$.

After the protocol, each party obtains a secret share of the DH key $\alpha\beta \cdot G$, denoted as $[\![\alpha\beta \cdot G]\!]_i$. The result is used to derive the TLS handshake keys as discussed next.

Step 3: Distributed key derivation. In TLS 1.3, the next step is to compute the TLS handshake keys, which are later used in step 4. The N servers, which act as the TLS client, need to compute this key derivation function KDF in SMPC. The key derivation function takes the share of the DH key $[\![\alpha\beta \cdot G]\!]_i$, the hash of `ClientHello` $\mathsf{Hash}(\alpha \cdot G)$ and `ServerHello` message $\mathsf{Hash}(\beta \cdot G)$ as input, and outputs the handshake keys hk as secret-shares within SMPC.

We identify that the hashes of the `ClientHello` and `ServerHello` messages can be computed outside SMPC, which reduces the overhead. The forwarding server $\mathcal{P}_1$ who computes $\alpha \cdot G$ and receives $\beta \cdot G$ from the TLS server can broadcast these messages to other parties. Each party $\mathcal{P}_i$ then computes the hash of these messages in plaintext, and uses these hashes, as well as their share of the DH key, as inputs to the SMPC engine.

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

In MPCAuth, we are able to remove this task from SMPC. The insight is that the handshake keys, which encrypt the response, are only designed to hide the TLS endpoints' identity, which is unnecessary because the N servers must confirm the TLS server's identity in our setting. Several works show that revealing the keys does not affect other guarantees of TLS [44, 89, 38, 59]. We formalize this insight in our definition of the ideal functionality $\mathcal{F}_{\mathsf{TLS}}$, as described in Section B-B.

Therefore, verifying the TLS server's response is as follows: after all the N servers receive and acknowledge all the messages from `ServerHello` to `ServerFinished` sent by the TLS server and forwarded by the first server, the SMPC protocol reveals the TLS handshake keys hk to all the N servers. Each server decrypts the response, and verifies the certificate, signature, and verification data within it. Using the handshake keys, the N servers can then assemble the client's verification message `ClientFinished` within SMPC, and send it to the TLS server.

Step 5: Session keys derivation and precomputation for authenticated encryption. Lastly, the N servers use the same key derivation function from step 3 to derive the session keys (application keys) and IVs (initialization vectors) in SMPC. This key derivation function maintains an internal state and takes the hash of the new messages from the transcript as additional inputs to generate new keys. These session keys and IVs are later used by the authenticated encryption protocol during the data exchange phase.

We now finish the TLS handshake phase and are ready to proceed to the next stage: the data exchange phase. Before we get to the next stage, we observe that the authenticated encryption scheme used in the data exchange phase may allow some one-time precomputation that can be done as part of the TLS handshake phase. For example, for AES-GCM, MPCAuth can precompute the AES key schedule and secret-share the GCM power series. We provide more details of these protocols in Section III-D.

D. Data exchange in SMPC.

The rest of the TLS-in-SMPC protocol involves data encryption and decryption. The session keys obtained from the handshake protocol are now used as the symmetric keys to encrypt and apply MAC on data transmitted over the network. In TLS, the MAC and the ciphertexts are generated together using an authenticated encryption algorithm chosen from the TLS cipher suites. An opportunity to reduce the latency is to choose the TLS cipher suites carefully, as shown by both our investigation and prior work [89, 81].

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

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

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

Efficient AES-GCM-128 in SMPC. The AES-GCM-128 cipher applies GCM mode-of-operation on the AES block cipher to simultaneously ensure the confidentiality and authenticity of data. AES-GCM-128 takes the session keys, IVs, and plaintext data as inputs, and outputs the ciphertext and MAC. Many prior works have already proposed efficient protocols to evaluate AES in SMPC [63, 72, 71, 83], and we adapt the circuit proposed in [83].

For the GCM protocol, we use a protocol from DECO [89] to efficiently compute the GCM tag, and adapt it to the N-party setting. After deriving the TLS application key within SMPC, the servers compute the GCM generator $H = E_K(0)$ and the power series of H: $H, H^2, H^3, ..., H^L$ within SMPC. The power series is secret-shared among the N servers. To compute the GCM tag for some data $S_1, S_2, ..., S_L$ (authenticated data or ciphertexts), each server computes a share of the polynomial $\sum_{i=1}^{L} S_i \cdot H^i$ and combines these shares with the encryption of initialization vector (IV) within SMPC.

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

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

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

IV. MPCAuth Authentication

In this section, we present MPCAuth's authentication protocol for email, SMS, U2F, security questions, and biometrics. For each factor, we also describe how it achieves profile hiding, and prove its security in Section C. In addition, we describe how a new client sets up the authentication factor when she enrolls in the service.

General workflow. In general, MPCAuth's authentication protocols consist of two stages:

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

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

A. MPCAuth *email*

MPCAuth's email authentication protocol sends the client only one email which contains a passcode. If the client proves knowledge of this passcode in some way, the N servers will consider the client authenticated. Note that the protocol sends the email using the intergateway SMTP protocol, rather than the one normally used by a client.

Email authentication protocol $\Pi_{\textbf{email}}$. Given the share of the client's email address $[\![\mathsf{addr}]\!]_i$, MPCAuth's email authentication protocol proceeds 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 $\pi_i' = \mathsf{PRF}(s, i)$ to $\mathcal{P}_i$.
3) The N servers run the TLS-in-SMPC protocol to create a TLS endpoint acting as an email gateway for some domains. The TLS endpoint opens a TLS connection with the client's SMTP server. Over this connection, the N servers send a message which includes the email address addr, and the email content that contains the passcode s. The SMTP server then forwards the email to addr.
4) The client receives the email and enters s into the client app, which computes $\pi_i = \mathsf{PRF}(s, i)$ and sends π_i to $\mathcal{P}_i$. $\mathcal{P}_i$ considers the client authenticated if $\pi_i = \pi_i'$.

Enrollment. The enrollment workflow is as follows:

1) The client opens a TLS connection with each of the N servers, secret-shares her email address, and sends the i-th share $[\![\mathsf{addr}]\!]_i$ to $\mathcal{P}_i$.
2) The N servers run Π_{email}, in which the servers jointly send a confirmation email to the client, with a passcode.
3) If the client is authenticated, each server keeps a mapping from the client's username to $[\![\mathsf{addr}]\!]_i$.

Profile hiding. Profile hiding of email authentication requires that the servers to not learn the email username of the client. This is achieved because the email address is stored as secret shares on the servers, and during authentication, this address is encrypted and transmitted within SMPC during TLS's data exchange phase. None of the servers see the email address in plaintext, and an attacker cannot recover the address unless it compromises all N servers.

Note that in TLS-in-SMPC, since the N servers verify the TLS server certificate outside SMPC for efficiency reasons, the email provider's mail gateway address is revealed to the N servers. We believe that hiding only the email username instead of the full address suffices for our privacy goal because (1) major email service providers (e.g. Google, Microsoft) have millions and even billions of accounts, and (2) most organizations with custom domains also uses email services from these major providers (e.g. many universities and companies use Gmail or Outlook). In the latter case, the N servers contact the email service provider's SMTP server, and only learn the provider's gateway address instead of the organization's custom domain name.

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

– *Sender Policy Framework (SPF).* MPCAuth can follow the SPF standard [28], in which the sender domain, registered during the setup of MPCAuth, has a `TXT` record indicating the IP addresses of email gateways eligible to send emails from this sender domain.
– *Domain Keys Identified Mail (DKIM).* The DKIM standard [12] requires each email to have a signature from the sender domain under a public key listed in a `TXT` record. MPCAuth can have the server generate the keys and sign the email, both in a distributed manner.

Our experiments show that supporting SPF is sufficient to avoid Gmail labeling MPCAuth's email as spam.

B. MPCAuth *SMS*

MPCAuth's SMS protocol sends the client *one* text message, which contains a passcode. The enrollment and authentication protocols resemble the email authentication protocol except that the passcode is sent via SMS.

SMS authentication protocol $\Pi_{\textbf{SMS}}$. We leverage the fact that many mobile carriers, including AT&T [4], Sprint [29], and Verizon [32], provide commercial REST APIs to send text messages. The N servers, who secret-share the API key, can use MPCAuth's TLS-in-SMPC protocol to send a text message to the client through the relevant API. The authentication protocol for SMS is the same as Π_{email} except that during step 3), the N servers create a TLS endpoint with the mobile carrier's gateway server. During enrollment, the client's phone number is stored as secret shares.

Profile hiding. Similar to profile hiding for email authentication, the client's phone number is stored as secret shares during enrollment and encrypted within TLS-in-SMPC so none of the servers sees this number in plaintext. The N servers need to know the client's mobile carrier to send the message. Since major mobile carriers such as AT&T and T-Mobile have hundreds of millions of users, we believe this leakage is acceptable in practice.

C. MPCAuth *U2F*

The universal second factor (U2F) [33] 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 [36] and Google Titan [30]. The goal of MPCAuth's U2F protocol is to have the user operates on the U2F device *once*.

Background: U2F. A U2F device attests to a client's identity by generating a signature on a challenge requested by a server under a public key that the server knows. The U2F protocol

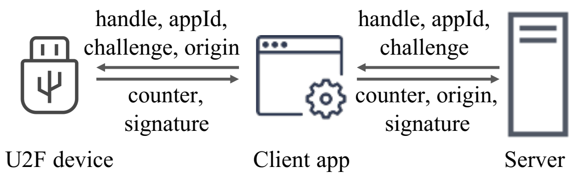


Fig. 3: Protocol of universal second factor (U2F).

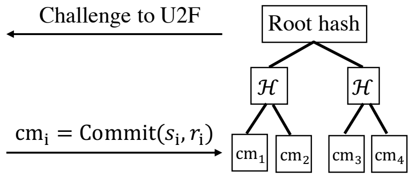


Fig. 4: The Merkle tree for U2F challenge generation.

consists of an enrollment phase and an authentication phase, described as follows.

In the enrollment phase, the U2F device generates an application-specific keypair and sends a key handle and the public key to the server. The server stores the key handle and the public key. In the authentication phase, the server generates a random challenge and sends it over together with the key handle and the application identifier (appId) 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 [36, 30], the U2F device generates a signature over the request. The signature also includes a monotonic counter to discover cloning attacks. The server receives the signature and verifies it using the public key stored in the enrollment phase.

An insecure strawman. An intuitive approach to avoid the client tapping the U2F device multiple times is to have the servers generate a joint challenge which is then signed by the U2F device. The client can secret-share the signature, and the servers can then reconstruct and verify the signature within SMPC. This approach works but introduces unnecessary overhead.

Note that, unlike other factors, U2F does not store any private authentication profiles on the servers. The public key and the signature are meant to be public, so there is no need to secret-share them and run SMPC on them. The client can obtain a signature over the servers' joint challenge from U2F, and sends the signature to each server. Each server can simply verify this signature individually.

This strawman approach, however, is insecure because it suffers from the replay attack described in Section II-D. When a malicious server receives the signature from the client, this server can impersonate the honest user by sending this signature to the other servers.

MPCAuth U2F's protocol $\Pi_{\mathbf{U2F}}$. To address the usability challenge while preventing the replay attack, our construction needs to satisfy the following three requirements: (1) the challenge signed by the U2F device is generated using all the servers' randomness $s_1, s_2, ..., s_N$; (2) the client can prove to server $\mathcal{P}_i$ that the signed challenge uses s_i without revealing s_i to other servers; and (3) the client's response to each server is unique, and the attacker cannot forge $\mathcal{P}_i$'s response without knowing s_i.

We identify that aggregating the servers' randomness via a Merkle tree combined with a commitment scheme, as Figure 4 shows, satisfies these requirements. We can use a cryptographic commitment scheme to commit to the client's randomness s_i, and aggregate all these commitments using a Merkle tree. The U2F device can then sign a signature over the Merkle root hash. The detailed protocol is as follows:

1) Each server $\mathcal{P}_i$ opens a TLS connection with the client and sends over a random value s_i.
2) The client randomly samples r_i for $\mathcal{P}_i$ and builds a Merkle tree over the committed values $\mathsf{cm}_i = \mathsf{Commit}(s_i, r_i)$, as illustrated in Figure 4. The client then requests the U2F device to sign the root hash root.
3) The client then operates on the U2F device *once*, which produces a signature σ_{root} over root. The client app computes the Merkle tree inclusion proof π_i^{Merkle}, and sends the tuple $(\mathsf{root}, \sigma_{\mathsf{root}}, \pi_i^{\mathsf{Merkle}}, \mathsf{cm}_i, r_i)$ to each server.
4) Each server $\mathcal{P}_i$ (1) verifies the signature σ_{root} with $\mathsf{Sig.Verify}$, (2) checks that $\mathsf{cm}_i = \mathsf{Commit}(s_i, r_i)$ and (3) checks the Merkle lookup proof to verify cm_i is indeed a leaf in the tree. $\mathcal{P}_i$ consider the client authenticated if everything is verified.

Enrollment. The enrollment workflow is as follows:

1) The client and the servers engage in the standard U2F enrollment protocol [33], in which the servers obtain the key handle and the public key.
2) The client and the servers run MPCAuth's U2F authentication protocol Π_{U2F} as described above.

Profile hiding. Based on the definition, profile hiding of Π_{U2F} means that the N servers cannot associate their public key with the client's device. The original U2F protocol already provides measures to hide the device's identity. [34, 15]. Namely, the U2F device does not have any publicly available serial number, is not tied to the client's device, and generates a new key pair for every service it registers. These properties are preserved in the N servers setting; an attacker compromising $N-1$ servers can neither identify the U2F's secret key given the public key, nor associate the key pair with the client's device.

D. MPCAuth *Biometrics*

Biometric authentication [45, 66] relies on a client's unique biological characteristics to verify her identity. Fingerprints, facial recognition, and retina scans are some of the common biometrics used today. Biometrics protocols fall into two categories: device-side and server-side. Device-side biometrics [1, 2] performs the authentication locally, where the client uses her biometric information to unlock a key stored on her device. This key is then used to sign the servers' challenges. Since the authentication workflow of device-side biometrics is similar to that of U2F, we can use U2F's protocol described in Section IV-C to authenticate to N servers. In this section, we focus our presentation on the server-side protocol.

For server-side biometrics, the client scans her biometrics on a device. The biometric information is processed by some processing algorithms (e.g. a machine learning model) to obtain the feature vector, which is sent to the server. The server compares this vector with the client's registered feature vector by computing a distance function (normally an l_2 distance), and accepts if the distance is below a certain threshold.

MPCAUTH's Biometrics protocol Π_{bio}. Given a processing algorithm $M : \mathbb{R}^* \to \mathbb{R}^\lambda$ that produces a feature vector of length λ, the protocol works as follows:

- During enrollment, the client scans her biometrics information $\mathbf{x}$, the client app processes it with M and get the feature vector $\mathbf{y} = M(\mathbf{x})$. The client app then secret-shares $\mathbf{y}$ and sends $[\![\mathbf{y}]\!]_i$ to server $\mathcal{P}_i$.
- During authentication, the client scans her biometrics x'. The client app computes its feature vector $\mathbf{y}' = M(\mathbf{x}')$, and sends $[\![\mathbf{y}']\!]$ to the servers. The servers compute the l_2 distance between $[\![\mathbf{y}]\!]$ and $[\![\mathbf{y}']\!]$, and compare it with a given threshold, both in SMPC. The servers reconstruct the comparison bit $b \in \{0, 1\}$ and accept if the b is 1.

Bounding vector components. The l_2 distance between two n dimensional vector is defined as $l_2(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + ... + (x_n - y_n)^2}$. As SMPC protocol works over finite rings, individual component of the vector wraps around the ring. It is possible to create an attack vector with large component-wise distances but a small l_2 distance by overflowing the l_2 value. Therefore, we additionally need to specify a maximum value w for each individual component x_i so that $-w < x_i < w$. In addition, instead of comparing l_2 with the threshold, we compare $(l_2)^2$ with the square of the threshold to avoid the costly square root operation in SMPC.

Profile hiding. Privacy concerns are the major obstacle that prevents the adoption of server-side biometrics [17, 7]. Previously, preserving the privacy of biometrics is difficult: to compare between features, the server needs to store the client's biometric features directly without applying any hashes. If an attacker compromises the server, they can not only use them to authenticate to other services, but even recover the original biometrics as well.

In the distributed-trust setting, we are able to hide this information from the servers since they are stored as secret shares. The authentication protocol is also computed within SMPC. The servers learn no information other than the fact that the client is authenticated.

E. MPCAUTH *security questions*

The last MPCAuth authentication factor we present is security questions. Note that passwords are a special type of security question – the account name is the question and the password is the answer. As the most common authentication factor, protocols for security questions have been extensively studied by prior work [40, 37]. We present MPCAuth's protocol here for completeness. Without loss of generality, let's assume that the client provides one security question and answer. The protocol is as follows:

- During enrollment, the client provides security question $q \in \{0, 1\}^*$ and its answer $a \in \{0, 1\}^*$ to the client app, hashes each of them with a CRH, and sends the secret-shares of the hashes to the N servers. Server $\mathcal{P}_i$ holds the tuple $([\![\mathsf{Hash}(q)]\!]_i, [\![\mathsf{Hash}(a)]\!]_i)$.
- During authentication, the client again sends secret shares of the hashes $([\![\mathsf{Hash}(q)]\!]'_i, [\![\mathsf{Hash}(a)]\!]'_i)$ to $\mathcal{P}_i$. The N servers runs a private comparison protocol between the pair $([\![\mathsf{Hash}(q)]\!]_i, [\![\mathsf{Hash}(q)]\!]_i)$ and $([\![\mathsf{Hash}(a)]\!]_i, [\![\mathsf{Hash}(a)]\!]'_i)$ to check that their reconstructions are the same. The server considers the client authenticated if the check passes.

Profile hiding. The construction is profile hiding because both questions and answers are stored as secret shares, and are never materialized during authentication. Note that in the special case where the answer is a password, we do not hide the account name from the servers. This is because the account name is a unique identifier for the client and must be stored publicly. (see Section II-D)

Previously, security questions have to avoid asking users for critical personal secrets, such as their SSN, because the secrets are in a small domain and an offline brute-force attack is enough to recover them. The profile-hiding aspect of MPCAuth's protocol may encourage the user to choose sensitive questions and answers that they are previously uncomfortable sharing.

V. IMPLEMENTATION AND EVALUATION

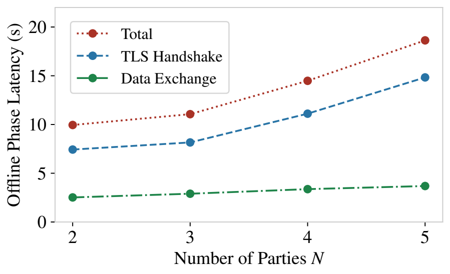
In this section we discuss MPCAuth's performance by answering the following questions:

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

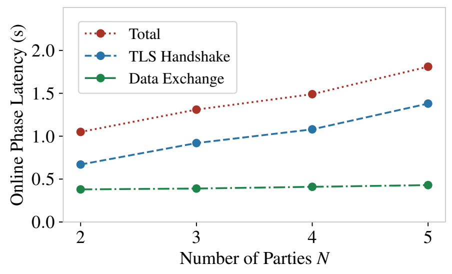
Our benchmark results show that MPCAuth's TLS-in-SMPC protocol can consistently meet the TLS handshake timeout for $N \leq 5$, and is 8x faster compared to a baseline implementation.

A. *Implementation details.*

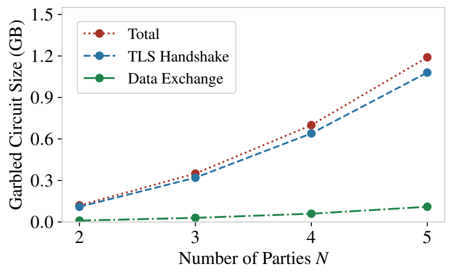
We use MP-SPDZ [24], emp-toolkit [13, 87] and wolfSSL [35] to implement MPCAuth's TLS-in-SMPC protocol. The TLS handshake protocol consists of point additions (for elliptic curve Diffie-Hellman) and key derivations. We observe that point additions can be efficiently expressed as an arithmetic circuit whose native field is exactly the point's coordinate



(a) Offline phase latency.



(b) Online phase latency.



(c) Garbled circuit size.

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

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

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

field, and key derivations can be efficiently expressed as a boolean circuit. Our insight to achieve efficiency here is to mix SMPC protocols by first implementing point additions with SPDZ using MASCOT [67] for the offline phase, and then transferring the result to AG-MPC [83, 87] for key derivation via a maliciously secure mixing protocol [77, 39, 60]. Both SPDZ and AG-MPC support offline precomputation, which helps reduce the online latency and meet the TLS handshake timeout.

The AES-GCM-256 algorithm during the data exchange phase can be expressed as a garbled circuit. Besides, AES-GCM-256, many operations in TLS-in-SMPC, including the hash function SHA-256 and the key derivation function are expressed as garbled circuits. We synthesize the circuit files in TLS-in-SMPC using Synopsys's Design Compiler and tools in TinyGarble [79], SCALE-MAMBA [27], and ZKCSP [50]. These circuits are already open-sourced.

All our authentication protocols are implemented in C++ and Python. For our biometrics protocol, we choose facial recognition as the biometrics and use an open-sourced 'face_recognition' [14] library to extract facial features. The library provides APIs to encode facial images into feature vectors of 128 dimensions.

B. Experiment setup

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

Remark V.1 (On the size of N)**.** N is the number of parties that host the distributed-trust applications. Most industry adoption of MPC systems (e.g. Fireblocks MPC wallet [16], Meta private ads [23] consider less than 5 parties, and the majority of them only have 2-3 parties in their settings. We, therefore, consider N from 2 to 5 in our experiments.

C. TLS-in-SMPC's performance

For the TLS-in-SMPC protocol, we measured the offline and online phases' latencies and the size of the garbled circuits sent in the offline phase and show the results in Figure 5. From the figure, we see that the offline and online phase latencies and the total circuit size grow roughly linearly to the number of servers. For all values of N that we tested, the protocol always meets the TLS handshake timeout.

A large portion of the offline cost is in transmitting the garbled circuits used in AG-MPC, as Figure 5 shows. MPCAuth's servers run the offline phase before the TLS connection is established to avoid this extra overhead. To load these circuits to the memory efficiently, one can use a memory planner optimized for secure computation [68]. Malicious users can perform DoS attacks by wasting computation done in the offline phase. MPCAuth can defend against such attacks using well-studied techniques, such as proof-of-work or payment [69, 64].

Latency breakdown. In Table I we show a breakdown of the offline and online phase latencies for the TLS-in-SMPC protocol. From the table, we see that most of the computation is in the offline phase, and the online phase has a small latency. Therefore, if we run an SMPC protocol off the shelf that does not precompute the offline phase, from Table I we see that for

	Offline Phase Latency (s)	**Online Phase Latency (s)**
Email	10.96 (2.90)	1.29 (0.39)
SMS	12.26 (4.10)	1.48 (0.56)
U2F	—	0.03
Security Questions	0.03	0.04
Biometrics	8.89	0.38

TABLE II: Latencies of MPCAuth ($N = 3$). Numbers in parentheses are the cost given an established TLS connection.

$N = 5$, the key exchange has a latency of 14.83 s and cannot meet a TLS handshake timeout of 10 s.

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

D. MPCAuth's performance

We measured the offline and online phase latencies of the MPCAuth protocols and presented the results in Table II. We now discuss the results in more detail.

Email/SMS. Using a message of 34 characters that includes a short sentence and a passcode, the MPCAuth email protocol (without DKIM signatures) sends 165 bytes via TLS-in-SMPC, and MPCAuth's SMS protocol sends 298 bytes via TLS-in-SMPC.

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 costs, 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.

Biometrics. We implemented the entire protocol in MP-SPDZ using MASCOT. Given a 128-dimension feature vector, the protocol performs 257 comparisons and 128 multiplications, which requires 14293 authenticated triples generated during the preprocessing phase.

E. Comparisons

Comparison with off-the-shelf SMPC. We compare MPCAuth's implementation with an off-the-shelf one in emp-toolkit [13, 87]. We estimate this cost by implementing the computation of $\alpha\beta \cdot G$ in key exchange, which offers a lower bound on its performance of TLS and is already much slower than MPCAuth. With $N = 5$ servers, the overall latency is at least $8\times$ slower compared with MPCAuth's TLS-in-SMPC implementation. This is because computing $\alpha\beta \cdot G$ involves expensive prime field operations using 10^7 AND gates. With $N = 5$ servers, this step already takes 150 s in the offline phase and 8.6 s in the online phase.

VI. 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 Section II-C. 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 [41, 65]. MPCAuth can support identifiable abort by incorporating these SMPC protocols and standard identification techniques in its authentication protocols.

A client app is not sufficient. One potential approach to address the usability challenge is to build a client app that automatically performs the N authentications for the client. This approach works for passwords and biometrics, but it does not work for email and SMS. Consider a strawman where a client app would read the emails and SMSes of a user to discover messages from the N servers. This process requires the client app to receive intrusive permissions to scan over the client's email and SMS messages, which significantly affects the privacy of the client. Such apps become a dangerous central point of attack because the client's email and SMS are used to authenticate to other systems and applications as well (e.g., banking, payments, shopping). In fact, operating systems such as iOS do not give the app permission to access this data because of this privacy infringement. Alternatively, the client app can rely on specialized APIs from the operating systems to autofill these one-time passcodes N times without scanning the client data itself. For example, both iOS and Android provide APIs to autofill one-time passcodes for SMS messages. This approach is still problematic because the client still has to tap the autofill button N times, which remains an undesirable user experience, and this autofill approach does not work for email.

Account Recovery. In Section II-A, we require the client to register at least two distinct factors. This is not enough if she wants to enable account recovery. If she loses access to one of the factors, then she would not be able to successfully log in. Therefore, she needs to register for additional factors in case some factors are lost. For account recovery, it would be important for the client to provide at least two authentication factors. Otherwise, a malicious authenticator could use the account recovery mechanism to steal her account. In general, to tolerate k lost factors, the client needs to register for at least $k+2$ factors.

In our constructions, we also require that our client remember her username, which is used by the servers to index her authentication profiles. Without this information, the servers are unable to authenticate the client. There are several approaches to recover this username in case it is forgotten/lost. First, the client app can store this username for the client. Second, if the client registers for email/SMS authentication, the servers can include the client's username in every email/SMS message. The client can also ask the servers to send an additional email/SMS that contains the username.

VII. Related work

TLS and SMPC. There are works using TLS with secure two-party computation (S2PC), but in a prover-verifier setting in which the prover proves statements about the information on the web. BlindCA [81] uses S2PC to inject packets in a TLS connection to allow the prover to prove to the certificate authority that they own a certain email address. DECO [89] uses TLS within S2PC to prove to an outside party statements about data encrypted in the TLS channel. Both work heavily rely on the fact that one party knows *all the plaintext* because this party needs to prove knowledge of the plaintext of TLS-encrypted traffic. For this reason, many of their techniques do not apply to our setting. The technique we could reuse from DECO (Section III-D) was only a small component of our protocol. In addition, both of these works are restricted to two parties based on their intended settings, while MPCAuth supports an arbitrary number of parties.

A concurrent work [38] [2]also proposes running TLS in secure multiparty computation similarly to our building block TLS-in-SMPC. When comparing to our building block TLS-in-SMPC, they did not implement their proposal (we provide an end-to-end implementation compatible with an existing TLS library, wolfSSL), did not evaluate a concrete implementation, and did not contribute techniques for speeding up AES and GCM in MPC (Section III-D). More importantly, when comparing overall contributions, [38] does not propose or contribute authentication protocols for distributed-trust applications, whereas the core ideas and contributions of MPCAuth are (1) the idea that a user can authenticate to N distrusting servers simultaneously while performing only the usual work of authenticating to one ("imaginary") server, (2) showing how this can be achieved by running inside SMPC the SMTP protocol or the HTTP protocol in addition to TLS, (3) implementation and evaluation of the resulting authentication protocols, and (4) demonstrating how this capability addresses crucial roadblocks for applications like cryptocurrency custody/wallets, and collaborative learning/analytics, currently seeing adoption from companies in these areas.

[2]This work is concurrent to the eprint version of our paper, which is originally published in March 2021.

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 [86, 43], which has been used in the PGP protocol in which individuals prove the identities of each other by signing each other's public key [26, 6]. Another idea is to make the authentication system transparent to the users. For example, blockchain-based authentication systems, such as IBM Verify Credentials [19], BlockStack [8], and Civic Wallet [10], and certificate/key transparency systems [9, 74, 21, 46, 80, 61] have been deployed in the real world.

A recent concurrent work [48] also addresses decentralized authentication for cryptocurrency by integrating U2F and security questions with smart contracts. Their construction does not support SMS/email authentication due to limitations of smart contracts, and does not work with cryptocurrencies that do not support smart contracts like Bitcoin. In sum, their approach targets a different setting than MPCAuth, as we focus on the usability issues of having the user perform N-times the work.

OAuth. OAuth [25] is a standard protocol used for access delegation, which allows users to grant access to applications without giving out their passwords. While OAuth has several desirable properties, it does not work for all of MPCAuth's factors. It is, therefore, less general and flexible than MPCAuth. In addition, if a user authenticates through OAuth and wants distributed trust, she has to perform the authorization N times, once for each server. MPCAuth can incorporate OAuth as a separate authentication factor—the N servers can secret-share the OAuth client secret and then, using TLS-in-SMPC, obtain the identity information through the OAuth API.

VIII. Conclusion

MPCAuth is an authentication system for distributed-trust applications that enables users to authenticate to N servers by only performing the work of a single authentication, and in the meantime hides the client's authentication profile. MPCAuth offers authentication protocols that achieve both properties for various commonly used authentication factors. An important building block of MPCAuth 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 MPCAuth will facilitate the adoption of new systems with distributed trust.

References

[1] Accessing keychain items with face id or touch id. https://developer.apple.com/documentation/localauthentication/accessing_keychain_items_with_face_id_or_touch_id.

[2] Android's biometrics. https://source.android.com/docs/security/features/biometric.
[3] Ant morse. https://antchain.antgroup.com/products/morse.
[4] AT&T SMS API. https://developer.att.com/sms.
[5] Basebit. https://www.basebit.ai/en/core.aspx.
[6] Biglumber: Key signing coordination. http://www.biglumber.com/.
[7] Biometrics and privacy. https://ovic.vic.gov.au/privacy/resources-for-organisations/biometrics-and-privacy-issues-and-challenges.
[8] Blockstack. https://www.blockstack.org/.
[9] Certificate transparency. https://www.certificate-transparency.org/.
[10] Civic wallet - digital wallet for money and cryptocurrency. https://www.civic.com/.
[11] Curv: The institutional standard for digital asset security. https://www.curv.co.
[12] DomainKeys identified mail (DKIM) signatures. https://tools.ietf.org/html/rfc6376.
[13] Efficient multi-party (EMP) computation toolkit. https://github.com/emp-toolkit/.
[14] face recognition. https://github.com/ageitgey/face_recognition.
[15] Fido technotes: The truth about attestation. https://fidoalliance.org/fido-technotes-the-truth-about-attestation/.
[16] Fireblocks. https://fireblocks.com/.
[17] How the biometrics industry can overcome the privacy obstacle. https://findbiometrics.com/yir-how-biometrics-industry-overcome-privacy-obstacle-702099.
[18] Huakong tsingjiao. https://www.tsingj.com/.
[19] IBM Verify Credentials: Transforming digital identity into decentralized identity. https://www.ibm.com/blockchain/solutions/identity.
[20] The illustrated TLS 1.3 connection. https://tls13.ulfheim.net/.
[21] Key transparency. https://github.com/google/keytransparency.
[22] Mpcvault. https://mpcvault.com/.
[23] Multi-party computation - facebook. https://privacytech.fb.com/multi-party-computation/.
[24] Multi-Protocol SPDZ (MP-SPDZ). https://github.com/data61/MP-SPDZ.
[25] OAuth. https://www.oauth.net/.
[26] OpenPGP message format. https://tools.ietf.org/html/rfc4880.
[27] Scale-mamba. https://github.com/KULeuven-COSIC/SCALE-MAMBA.
[28] Sender policy framework (SPF) for authorizing use of domains in email. https://tools.ietf.org/html/rfc7208.
[29] Sprint enterprise messaging developer APIs. https://sem.sprint.com/developer-apis/.
[30] Titan security key. https://cloud.google.com/titan-security-key.
[31] The transport layer security (TLS) protocol version 1.3. https://tools.ietf.org/html/rfc8446.
[32] Verizon's enterprise messaging access gateway. https://ess.emag.vzw.com/emag/login.
[33] What is U2F? https://developers.yubico.com/U2F/.
[34] Why FIDO U2F was designed to protect your privacy. https://www.yubico.com/blog/fido-u2f-designed-protect-privacy.
[35] wolfssl embedded ssl/tls library — now supporting tls 1.3. https://https://www.wolfssl.com/.
[36] YubiKey strong two factor authentication. https://www.yubico.com/.
[37] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. Robust password-protected secret sharing. In *ESORICS '16*, 2016.
[38] Damiano Abram, Ivan Damgård, Peter Scholl, and Sven Trieflinger. Oblivious TLS via multi-party computation. In *CT-RSA '21*, 2021.
[39] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In *WAHC '19*.
[40] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *CCS '11*, 2011.
[41] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Efficient constant-round MPC with identifiable abort and public verifiability. In *CRYPTO '20*, 2020.
[42] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88*.
[43] Thomas Beth, Malte Borcherding, and Birgit Klein. Valuation of trust in open networks. In *ESORICS '94*.
[44] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *S&P '17*.
[45] Debnath Bhattacharyya, Rahul Ranjan, Farkhod Alisherov, Minkyu Choi, et al. Biometric authentication: A review. *International Journal of u-and e-Service, Science and Technology*, 2(3):13–28, 2009.
[46] Joseph Bonneau. EthIKS: Using Ethereum to audit a CONIKS key transparency log. In *FC '16*.
[47] 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*, 2006.
[48] Florian Breuer, Vipul Goyal, and Giulio Malavolta. Cryptocurrencies with security policies and two-factor authentication. In *EuroS&P '21*, 2021.
[49] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In *EUROCRYPT '18*.
[50] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *CCS '17*.
[51] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS '01*.
[52] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In *CCS '14*.
[53] David Chaum, Crépeau. Claude, and Ivan Damgård. Multiparty unconditionally secure protocols. In *STOC '88*.
[54] Weikeng Chen and Raluca Ada Popa. Metal: A metadata-hiding file-sharing system. In *NDSS Symposium 2020*, 2020.
[55] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 259–282, Boston, MA, March 2017. USENIX Association.
[56] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. {DORY}: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1101–1119, 2020.
[57] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. In *TIT '76*.
[58] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.
[59] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *CCS '15*, 2015.
[60] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO '20*.
[61] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. Certificate transparency with privacy. In *PETS '17*.
[62] 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*.
[63] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round mpc combining bmr and oblivious transfer. *Journal of Cryptology*, 33(4):1732–1786, 2020.
[64] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. Ghostor: Toward a secure data-sharing system from decentralized trust. In *NSDI '20*, 2020.
[65] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *CRYPTO '14*, 2014.
[66] Anil K Jain and Karthik Nandakumar. Biometric authentication: System security and user privacy. *Computer*, 45(11):87–92, 2012.
[67] Marcel Keller and Emmanuela Orsini. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS '16*, 2016.
[68] Sam Kumar, David Culler, and Raluca Ada Popa. Nearly zero-cost virtual memory for secure computation. In *OSDI '21*, 2021.
[69] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *OSDI '16*.
[70] Yehuda Lindel. *How to simulate it: A tutorial on the simulation proof technique*, pages 277–346. 2017.
[71] Yehuda Lindell, Benny Pinkas, Nigel P Smart, and Avishay Yanai. Efficient constant-round multi-party computation combining bmr and spdz. *Journal of Cryptology*, 32(3):1026–1069, 2019.
[72] Yehuda Lindell, Nigel P Smart, and Eduardo Soria-Vazquez. More efficient constant-round multi-party computation from bmr and she. In *Theory of Cryptography Conference*, pages 554–581. Springer, 2016.
[73] Hemanta K. Maji, Manoj Prabhakaran, and Mike Rosulek. Complexity of multi-party computation functionalities. In *IACR ePrint 2013/042*.

[74] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: bringing key transparency to end users. In *SEC '15*.
[75] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. Senate: A Maliciously-Secure MPC platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2129–2146. USENIX Association, August 2021.
[76] Eric Rescorla. The transport layer security (tls) protocol version 1.3. Technical report, 2018.
[77] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and boolean circuits with active security. In *INDOCRYPT '19*, 2019.
[78] 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*, 2015.
[79] 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*.
[80] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *CCS '19*.
[81] Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and abhi shelat. Blind certificate authorities. In *S&P' 19*, 2019.
[82] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861, 2015.
[83] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS '17*, 2017.
[84] 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*, 2010.
[85] Alma Whitten and J. Doug Tygar. Why johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security '99*, 1999.
[86] Raphael Yahalom, Birgit Klein, and Thomas Beth. Trust relationships in secure systems: A distributed authentication perspective. In *S&P '93*, 1993.
[87] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In *CCS '20*, 2021.
[88] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS '86*.
[89] 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*, 2020.
[90] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. Cerebro: A platform for Multi-Party cryptographic collaborative learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2723–2740. USENIX Association, August 2021.

A Cryptographic building blocks

Pseudorandom functions. $\mathsf{PRF} = \{\mathsf{PRF}_k : \{0,1\}^* \to \{0,1\}^{O(|k|)}\}_k$ where k denotes the seed. Functions in PRF are computationally indistinguishable from random functions to anyone who does not know the seed k.

Collision-resistant hash functions. A collision resistant hash function is a tuple of algorithms $\mathsf{CRH} = (\mathsf{Setup}, \mathsf{Hash})$ that works as follows:

- $\mathsf{Setup} \to \mathsf{pp}_{\mathsf{CRH}}$: samples a public parameter $\mathsf{pp}_{\mathsf{CRH}}$.
- $\mathsf{Hash}(\mathsf{pp}_{\mathsf{CRH}}, m) \to h$: given the public parameter, outputs a hash h on the message m.

The hash function is collision resistant if it is computationally infeasible to find $x \neq y$ such that $\mathsf{Hash}(\mathsf{pp}_{\mathsf{CRH}}, x) = \mathsf{Hash}(\mathsf{pp}_{\mathsf{CRH}}, y)$.

Digital signatures. A digital signature scheme is a tuple of algorithms $\mathsf{Sig} = (\mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify})$ that enables a user to produce a signature over a message that can be later verified by everyone.

- $\mathsf{Setup} \to (\mathsf{sk}, \mathsf{vk})$: generate a signing key sk and a verification key vk.
- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$: produce a signature with signing key sk on message m.
- $\mathsf{Verify}(\mathsf{vk}, \sigma, m) \to \{0,1\}$: given the verification key vk, outputs 1 if σ is a valid signature on message m, and 0 otherwise.

A secure signature scheme needs to be unforgeable, meaning that someone without sk is unable to forge a valid signature that makes Verify accept.

Commitments. A commitment schemes is a tuple of algorithms $\mathsf{Comm} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ that produces a commitment to a message m.

- $\mathsf{Setup}(\lambda) \to \mathsf{pp}_{\mathsf{Comm}}$: samples a public parameter $\mathsf{pp}_{\mathsf{CRH}}$.
- $\mathsf{Commit}(\mathsf{pp}_{\mathsf{Comm}}, m, r) \to \mathsf{cm}$: produces a commitment on the message m, given the randomness r.
- $\mathsf{Open}(\mathsf{pp}_{\mathsf{Comm}}, m, r, \mathsf{cm}) \to \pi$: outputs a proof π showing that cm is indeed the commitment of message m.

A commitment scheme needs to satisfy hiding and binding properties. Hiding means that cm reveals no information about m, and binding means that the commitment cm bound to message m can only be opened by m itself.

B Security proof of TLS-in-SMPC

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

A. Overview

We model the security in the real-ideal paradigm [70], which considers the following two worlds:

- **In the real world,** the N servers run protocol Π, MPCAuth'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 [49, 52].

Remark: revealing the server handshake key is safe. In the key exchange protocol described in Section III-C, the

The TLS-in-SMPC protocol

Notation: We use $\mathcal{F}_{\mathsf{SMPC}}^{\mathsf{Func}}$ to represents a function Func that is executed in SMPC.
TLS server $\mathcal{S}$: follows the TLS 1.3 protocol.
TLS client $\mathcal{P}$:
Offline phase:

- $\forall i \in [N]$, $\mathcal{P}_i$ samples $\alpha_i \leftarrow \mathbb{Z}_p^+$, computes $\alpha_i \cdot G$, and broadcast it to all other parties.
- $\mathcal{P}_1$ receives $\alpha_j \cdot G$ for $j \in \{2, ..., N\}$, computes $\alpha \cdot G = \sum_1^N \alpha_i \cdot G$

On receiving `ServerHello` **($\beta \cdot G$) from $\mathcal{S}$**:

- $\forall i \in [N]$, $\mathcal{P}_i$ computes $x_i := \alpha_i(\beta G)$, invokes $\mathcal{F}_{\mathsf{SMPC}}^{\mathsf{Sum}}(x_1, ..., x_n) := \sum_{i=1}^n x_i$ providing x_i as its private input. Each $\mathcal{P}_i$ receives $[\![\alpha\beta \cdot G]\!]_i$
- $\forall i \in [N]$, $\mathcal{P}_i$ provides $([\![\alpha\beta \cdot G]\!]_i, \mathsf{Hash}(\alpha G), \mathsf{Hash}(\beta G))$ as inputs and invokes $\mathcal{F}_{\mathsf{SMPC}}^{\mathsf{KDF}}$ to derive TLS handshake keys $\mathsf{hk} = (hk_1, hk_2)$, where $\mathcal{P}_i$ holds $([\![hk_1]\!]_i, [\![hk_2]\!]_i)$.

On receiving `ServerCert`, `ServerCertVerify`, `ServerFinished` **from $\mathcal{S}$**

- $\forall i \in [N]$, $\mathcal{P}_i$ reconstructs the handshake keys hk, follows the TLS 1.3 protocol to verify `ServerCert`, `ServerCertVerify`, and `ServerFinished`.
- $\forall i \in [N]$, $\mathcal{P}_i$ invokes $\mathcal{F}_{\mathsf{SMPC}}^{\mathsf{Sign}}$ to produce `ClientCertVerify`, $\mathcal{F}_{\mathsf{SMPC}}^{\mathsf{HMAC}}$ to produce the `ClientFinished` message, and send it to $\mathcal{S}$.
- $\forall i \in [N]$, $\mathcal{P}_i$ provides $[\![\alpha\beta \cdot G]\!]_i$ as inputs and invokes $\mathcal{F}_{\mathsf{SMPC}}^{\mathsf{KDF}}$ to derive session keys $\mathsf{sk} := (sk_1, sk_2)$ and IVs, where $\mathcal{P}_i$ holds $([\![sk_1]\!]_i, [\![sk_2]\!]_i)$.

During data exchange:

- $\forall i \in [N]$, $\mathcal{P}_i$ provides $([\![sk_1]\!]_i, [\![sk_2]\!]_i, [\![msg]\!])_i$ as inputs and invokes $\mathcal{F}_{\mathsf{SMPC}}^{\mathsf{AES-GCM-128}}$, which outputs $\mathsf{m} := (\mathsf{Enc}(msg), \mathsf{MAC}(msg))$ in plaintext to $\mathcal{P}_1$.
- $\mathcal{P}_1$ sends m to $\mathcal{S}$.

Fig. 6: The TLS-in-SMPC protocol.

protocol reveals the server handshake key and IV to all the MPCAuth 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 [31], which is a new property of TLS 1.3 that does not exist in TLS 1.2. This property is unnecessary in our setting in which the identity of the unmodified TLS server is known.

Several works have formally studied this problem and show that revealing the keys does not affect other guarantees of TLS [44, 89, 38, 59]. Interested readers can refer to these works for more information.

B. *Ideal functionalities*

Ideal functionality. In the ideal world, we model the TLS interaction with the unmodified, trusted TLS server as an ideal functionality $\mathcal{F}_{\mathsf{TLS}}$. We adopt the workflow of the standard secure message transmission (SMT) functionality $\mathcal{F}_{\mathrm{SMT}}$ defined in [51].

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

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

for this sid.

6) $\mathcal{F}_{\mathsf{TLS}}$ ignores other inputs and messages.

Multiparty computation functionality. In the hybrid model, we abstract SPDZ and AG-MPC as an ideal functionality $\mathcal{F}_{\mathsf{SMPC}}$, which provides the functionality of multiparty computation with abort. We require $\mathcal{F}_{\mathsf{SMPC}}$ to be reactive, meaning that it can take some input and reveal some output midway through execution, as specified in the function f being computed. A reactive SMPC can be constructed from a non-reactive SMPC scheme by secret-sharing the internal state among the N parties in a non-malleable manner, as discussed in [73]. $\mathcal{F}_{\mathsf{SMPC}}$ works as follows:

1) For each session sid, $\mathcal{F}_{\mathsf{SMPC}}$ waits for party $\mathcal{P}_i$ to send $(\mathtt{input}, sid, i, x_i, f)$, in which sid is the session ID, i is the party ID, x_i is the party's input, and f is the function to be executed.
2) Once $\mathcal{F}_{\mathsf{SMPC}}$ receives all the N inputs, it checks if all parties agree on the same f, if so, it computes the function $f(x_1, x_2, ..., x_N) \to (y_1, y_2, ..., y_N)$ and sends $(\mathtt{output}, sid, i, y_i)$ to party $\mathcal{P}_i$. Otherwise, it terminates this session and sends $(\mathtt{abort}, sid)$ to all the N parties.
3) If $\mathcal{F}_{\mathsf{SMPC}}$ receives $(\mathtt{Abort}, sid)$ from any of the N parties, it sends $(\mathtt{abort}, sid)$ to all the N parties.
4) $\mathcal{F}_{\mathsf{SMPC}}$ ignores other inputs and messages.

Restricted programmable random oracle. We use commitments in Section III-C to ensure that in Diffie-Hellman key exchange, the challenge $\alpha \cdot G$ is a random element. This is difficult to do without commitments because the adversary can control up to $N-1$ parties to intentionally affect the result of $\alpha \cdot G = \sum_{i=1}^{N} \alpha_i \cdot G$. In our security proof, we leverage a restricted programmable random oracle [49, 52], 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 $(\mathtt{Query}, sid, m)$ 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 $(\mathtt{result}, sid, m, h)$ to this party. Otherwise, it samples h from random, stores (m, h) in the list for sid, and returns $(\mathtt{result}, sid, m, h)$.
3) Both the simulator $\mathcal{S}$ and the real-world adversary $\mathcal{A}$ can send a message $(\mathtt{Program}, m, h)$ 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 $(\mathtt{Program}, r||(\alpha \cdot G - \sum_{i \in I} \alpha_i \cdot G), h)$ 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 $(\mathtt{GetHandshakeKeys}, sid)$ to $\mathcal{F}_{\mathsf{TLS}}$ to obtain the server/client handshake key and IV.
    b) $\mathcal{S}$ simulates the computation of the handshake keys in SMPC by pretending that the SMPC output is the handshake keys.
    c) $\mathcal{S}$ then simulates the remaining operations of key exchange in SMPC, which checks the server verification data and produces the client verification data.
4) $\mathcal{S}$ simulates the message encryption and decryption of the application messages by simply pretending the SMPC output is exactly the ciphertexts taken from actual TLS messages, also provided by $\mathcal{F}_{\mathsf{TLS}}$.
5) In the end, $\mathcal{S}$ outputs whatever the adversary $\mathcal{A}$ would output in the real world.

D. Proof of indistinguishability

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

1) In distributed generation of $\alpha \cdot G$, the only difference in the simulated output compared with Π's is that the honest party chooses its share as $\alpha \cdot G - \sum_{i \in I} \alpha_i G$ and uses a programmed hash value h for commitment. Since $\alpha \cdot G$ is sampled from random by the TLS client inside $\mathcal{F}_{\mathsf{TLS}}$, it has the same distribution as the $\alpha_i \cdot G$ sampled by an honest party. The properties of restricted programmable random oracle $\mathcal{F}_{\mathsf{rpRO}}$ show that no parties can detect that h has been programmed.

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

As a result, we have the following theorem.

Theorem B.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.*

C Security of MPCAuth's authentication protocols

In this section, we formalize the security goals of MPCAuth's authentication protocols, and provide proof sketches for each protocol.

Definition C.1 (N-wise authentication protocol)**.** An N-wise authentication protocol Π can be characterized by two phases: the enrollment phase and the authentication phase. The enrollment phase can be characterized by the following set of algorithms:

- $\mathsf{CreateProfile} \to (\mathsf{id}, \{\mathsf{profile}_i\}_{i\in[N]})$: The client generates an identifier and a set of authentication profiles. (e.g. shares of email address/SMS/passcode), and send it to the servers.
- $\mathsf{Enroll}_i(\mathsf{id}, \mathsf{profile}_i) \to \{0, 1\}$: Each server takes in the client's id and profiles, and stores them locally.

The authentication phase can be characterized by the following sets of algorithms:

- $\mathsf{GenerateChallenge}_i(\mathsf{id}, \mathsf{profile}_i) \to c_i$: Each server $\mathcal{P}_i$ generates a random challenge and sends it to the client.
- $\mathsf{Prove}(\mathsf{id}, w, \{c_i\}_{i\in[N]}) \to \{\pi_1, \pi_2, ..., \pi_n\}$: Client id takes her credential w, and the set of servers' challenges $\{c_i\}_{i\in[N]}$, outputs a set of N proofs where π_i is for $\mathcal{P}_i$.
- $\mathsf{Verify}_i(\mathsf{id}, c_i, \pi_i, \mathsf{profile}_i) \to \{0, 1\}$: Server $\mathcal{P}_i$ takes the client's id, the proof π_i, the challenge c_i it generates, and outputs 1 if it accepts and 0 if it rejects.

The protocol considers the client authenticated if $\mathsf{Verify}_i(\mathsf{id}, c_i, \pi_i, \mathsf{profile}_i) \to 1$ for all $i \in [N]$.

Definition C.2 (Security of an authentication protocol)**.** A N-wise authentication protocol Π is said to be secure if it satisfies the following properties:

- *Correctness*: Π is correct if $\forall i \in [N]$, an honest client id with credential w can produce proof π_i that $\mathsf{Verify}_i(\mathsf{id}, c_i, \pi_i, \mathsf{profile}_i)$ outputs 1 with overwhelming probability.
- *Soundness*: Π is sound if, for a malicious attacker without credential w, there is at least one party i such that $\mathsf{Verify}_i(\mathsf{id}, c_i, \pi, \mathsf{profile}_i)$ outputs 0 with overwhelming probability.

Formally, we model the soundness of MPCAuth's authentication protocol as a security game. In the game, there is an attacker, an honest user, and N servers labeled as $\mathcal{P}_1$ to $\mathcal{P}_N$. The honest user establishes a persistent TLS connection (modeled as an ideal functionality) with each server. The attacker compromises $N-1$ servers and can observe and modify all compromised servers' inputs, states, and network traffic. For email/SMS, there is an additional TLS server that the honest user and the N servers talk to. We assume this TLS server is trusted and uncompromised.

The attacker can ask the honest user to issue commands. However, the attacker cannot see the states of the honest user and the honest servers ($\mathcal{P}_1$ and the TLS server), as well as the network traffic between them. The security game goes as follows:

- The attacker chooses an authentication factor that it wants to attack. The attacker specifies the username, and lets the honest user runs MPCAuth's protocol to enroll in that factor using that username.
- The attacker can adaptively issue $O(poly(\lambda))$ commands asking the honest user to authenticate using MPCAuth's protocol.
- Finally, the attacker uses the same username to authenticate to the N servers. The attacker wins the game by convincing all servers to output 1 with non-negligible probability.

Without loss of generality, we assume that $\mathcal{P}_1$ is the uncompromised honest server. Since the attacker controls servers $\mathcal{P}_2, ..., \mathcal{P}_N$ and can simply let all of them output $\mathsf{Verify}_i(\mathsf{id}, c_i, \pi_i, \mathsf{profile}_i) = 1$. Soundness holds if and only if server P_1 rejects with overwhelming probability. Therefore, the attacker wins iff $\mathsf{Verify}_1(\mathsf{id}, c_1, \pi_1, \mathsf{profile}_1) = 1$ with non-negligible probability.

A. Proof sketch for the security of Π_{email} and Π_{SMS}

We now prove the security of Π_{email} and Π_{SMS}. as referenced in Section IV-A and Section IV-B. We assume that the TLS-in-SMPC protocol securely realized the ideal functionality $\mathcal{F}_{\mathsf{TLS}}$. As modeled in $\mathcal{F}_{\mathsf{TLS}}$, we assume that the TLS server that the N servers talk to is trusted and uncompromised. The attacker does not control this server and cannot see the network traffic between this server and the honest user. We also assume the attacker cannot access the honest user's email account or SMS messages, which we model as the honest user's internal states.

In both protocols, the credential is the one-time passcode $s = \bigoplus_i^N s_i$, and the proof for $\mathcal{P}_i$ is $\mathsf{PRF}(s, i)$. The email/SMS passcode protocol can be modeled as follows: The N servers use TLS-in-SMPC protocol to send the passcode s to the TLS server, and the TLS server forwards s to the honest user via a secure TLS channel. Given our above assumptions, the attacker cannot observe the passcode s. The attacker can see the challenges $s_2, ...s_N$ and $\mathsf{PRF}(s, 2), ..., \mathsf{PRF}(s, i)$ from the compromised servers. The honest server $\mathcal{P}_1$ accepts if and only if the attacker can produce $\mathsf{PRF}(s, 1)$.

Correctness. The protocol is correct by construction. An honest client with credential s is able to produce a proof $\pi_i = \mathsf{PRF}(s, i) = \pi'_i$ for $\mathcal{P}_i$.

Soundness. To break soundness, the attacker needs to produce $\mathsf{PRF}(s, 1)$ and send it to $\mathcal{P}_1$. As stated above, the attacker cannot observe the passcode s. By the definition of PRF, $\pi'_i = \mathsf{PRF}(s, i)$ is computationally indistinguishable to a random string $\{0,1\}^{O(\lambda)}$ for someone who does not know s. $\mathsf{PRF}(s, 2), ..., \mathsf{PRF}(s, i)$ therefore looks uniformly random to the attacker. The probability of the attacker producing the uniform random string $\mathsf{PRF}(s, 1)$ is negligible.

B. Proof sketch for the security of Π_{U2F}

In protocol Π_{U2F}, the credential is the U2F secret key sk, the challenge from $\mathcal{P}_i$ is s_i, and the proof for $\mathcal{P}_i$ is the tuple $\pi_i = (\mathsf{root}, \sigma_{\mathsf{root}}, \pi_i^{\mathsf{Merkle}}, \mathsf{cm}_i, r_i)$. During enrollment, the attacker observes pk. During authentication, the attacker observes all proof tuples except for $\mathcal{P}_1$'s tuple π_1. Therefore, it does not have access to s_1.

Correctness. An honest client with credential sk receives $\{s_i\}_{i\in[N]}$. It is able to commit to all s_i, create a Merkle tree over $\{\mathsf{cm}_i\}_{i\in[N]}$, and produce a valid signature over its root hash with sk.

Soundness. To break soundness, the attacker needs to produce the tuple $\pi_1 = (\mathsf{root}, \sigma_{\mathsf{root}}, \pi_1^{\mathsf{Merkle}}, \mathsf{cm}_1, r_1)$ to make $\mathcal{P}_1$ accepts, who performs the following three checks:

1) The signature σ_{root} is over the root hash root. The attacker sees both values from one of the compromised servers, so it can forward them to $\mathcal{P}_1$.
2) π_1^{Merkle} is a correct Merkle inclusion proof for leaf cm_1. The attacker can reconstruct the Merkle tree and produce this proof.
3) $\mathsf{cm}_1 = \mathsf{Commit}(s_1, r_1)$. The attacker observes cm_1, but does not have access to s_1 and r_1. The hiding property of the commitment scheme guarantees that cm_1 does not leak any information about s_1. The binding property of the commitment scheme guarantees the attacker cannot forge a different $\mathsf{cm}' \neq \mathsf{cm}_1$ that makes the honest server accept. Soundness holds as a result.

C. Proof sketch for the security of Π_{SQ}

In protocol Π_{SQ}, the credential is the security question and answer (q, a), and the proof for $\mathcal{P}_i$ is $(\llbracket \mathsf{Hash}(q) \rrbracket'_i, \llbracket \mathsf{Hash}(a) \rrbracket'_i)$. We model the private comparison protocol as a secure multi-party computation protocol implemented via AGMPC [83]. We model it as an ideal functionality $\mathcal{F}_{\mathsf{comparison}}$, which is special case of $\mathcal{F}_{\mathsf{SMPC}}$.

Correctness. $(\mathsf{Hash}(q), \mathsf{Hash}(a))$ is stored on the N servers as secret shares. A client holding the credential (q, a) is going to produce secret shares of the same hash $(\mathsf{Hash}(q), \mathsf{Hash}(a))$ so that the private comparison protocol outputs 1.

Soundness. The private comparison protocol, modeled as an ideal functionality $\mathcal{F}_{\mathsf{comparison}}$, guarantees that $\mathcal{P}_1$ accepts iff the attacker can produce a tuple that equals $(\mathsf{Hash}(q), \mathsf{Hash}(a))$. During enrollment and authentication, the attacker observes $(\llbracket \mathsf{Hash}(q) \rrbracket'_i, \llbracket \mathsf{Hash}(a) \rrbracket'_i)$ and $(\llbracket \mathsf{Hash}(q) \rrbracket_i, \llbracket \mathsf{Hash}(a) \rrbracket_i)$ for $2 \leq i \leq N$. The attacker cannot reconstruct $(\mathsf{Hash}(q), \mathsf{Hash}(a))$ given what it observes. The probability of the attacker producing the uniformly random tuple $(\mathsf{Hash}(q), \mathsf{Hash}(a))$ is negligible.

D. Proof sketch for the security of $\Pi_{Biometrics}$

In protocol Π_{SQ}, the credential is the biometric info $\mathbf{x}$, the authentication profile for $\mathcal{P}_i$ is the biometric feature share $\llbracket \mathbf{y}' \rrbracket$. The proof for $\mathcal{P}_i$ is the share of another biometric feature vector $\llbracket \mathbf{y}' \rrbracket$, where $\mathbf{y}' = M(\mathbf{x}')$. We assume for the same user, the model M always outputs feature vectors close to each other (below the threshold). And for different users, the model always output feature vectors far from each other (above the threshold). The l_2 distance protocol is a secure multi-party computation protocol modeled as an ideal functionality $\mathcal{F}_{\mathsf{l_2}}$. It is a special case of $\mathcal{F}_{\mathsf{SMPC}}$.

Correctness. Given the assumption, an honest user always produces feature vectors $\mathbf{y}$ and $\mathbf{y}'$ with a l_2 distance below the threshold. And $\mathcal{F}_{\mathsf{l_2}}$ guarantees correct execution of the l_2 distance check.

Soundness. The private l_2 distance check, modeled as an ideal functionality $\mathcal{F}_{\mathsf{l_2}}$, guarantees that $\mathcal{P}_1$ accepts iff the attacker can produce $\mathbf{y}'$ close to $\mathbf{y}$. Given our assumption, an attacker that does not have biometric info $\mathbf{x}$ cannot produce a feature vector $\mathbf{y}' = M(\mathbf{x})$ that is close to $\mathbf{y}$ from the model M, nor can it reconstruct the feature vector $\mathbf{y}$ by compromising $N - 1$ servers. Soundness holds as a result.

As a result, we have the following theorem:

Theorem C.3. *Assuming secure multiparty computation for private comparison and private l_2 distance test, random oracle, TLS-in-SMPC, and other standard cryptographic assumptions are securely realized, all MPCAuth's protocols (Π_{email}, $\Pi_{SMS}, \Pi_{U2F}, \Pi_{SQ}, \Pi_{Biometrics}$) are secure in the presence of a malicious attacker that statically compromises up to $N - 1$ out of the N parties.*