Private Certifier Intersection

Bishakh Chandra Ghosh[1], Sikhar Patranabis[2], Dhinakaran Vinayagamurthy[2], Venkatraman Ramakrishna[2], Krishnasuri Narayanam[2], and Sandip Chakraborty[1]

[1]Indian Institute of Technology Kharagpur
[2]IBM Research, India
ghoshbishakh@gmail.com, sikhar.patranabis@ibm.com, dvinaya1@in.ibm.com, vramakr2@in.ibm.com, knaraya3@in.ibm.com, sandipc@cse.iitkgp.ac.in

September 30, 2022

Abstract

We initiate the study of Private Certifier Intersection (PCI), which allows mutually distrusting parties to establish a trust basis for cross-validation of claims if they have one or more trust authorities (certifiers) in common. This is one of the essential requirements for verifiable presentations in Web 3.0, since it provides additional privacy without compromising on decentralization. A PCI protocol allows two or more parties holding certificates to identify a common set of certifiers while additionally validating the certificates issued by such certifiers, without leaking any information about the certifiers not in the output intersection. In this paper, we formally define the notion of multi-party PCI in the Simplified-UC framework for two different settings depending on whether certificates are required for any of the claims (called PCI-Any) or all of the claims (called PCI-All). We then design and implement two provably secure and practically efficient PCI protocols supporting validation of digital signature-based certificates: a PCI-Any protocol for ECDSA-based certificates and a PCI-All protocol for BLS-based certificates. The technical centerpiece of our proposals is the first secret-sharing-based MPC framework supporting efficient computation of elliptic curve-based arithmetic operations, including elliptic curve pairings, in a black-box way. We implement this framework by building on top of the well-known MP-SPDZ library using OpenSSL and RELIC for elliptic curve operations, and use this implementation to benchmark our proposed PCI protocols in the LAN and WAN settings. In an intercontinental WAN setup with parties located in different continents, our protocols execute in less than a minute on input sets of size 40, which demonstrates the practicality of our proposed solutions.

1 Introduction

In the traditional web (Web 2.0), users are dependent on a limited set of identity and service providers and public Certificate Authorities (CAs) [fir] to initiate trusted interactions. Recent trends in decentralization towards Web 3.0 aim to remove such dependencies on centralized service providers. A prominent problem in the decentralized

web revolves around identity and trust. Decentralized identifiers (DIDs) [RSL+20] and Verifiable Credentials (VCs) [SLC21] enable parties to own and control their identities. This implies a self-sovereign ability to create, update, and selectively share identity records. Importantly, one can prove properties (or *claims*) about themselves without relying on centralized/federated identity providers or a canonical trusted set of CAs [RSL+20, did20, TR16], as long as the VC issuer (also called a trust anchor [ind]) is trusted by both the prover and the verifier of a claim. In a nutshell, existing DID and VC recommendations give users the ability to control their privacy while engaging in a trusted decentralized interaction. But, there are scenarios where these recommendations cannot adequately safeguard user privacy unless we introduce new privacy-preserving mechanisms. In its most general form, the scenario we are concerned about involves two parties wishing to establish a trust basis for future interactions. Service providers in the Semantic Web have encountered such situations, and mechanisms for trust negotiation [WYS+02] were proposed to minimize privacy compromise without sacrificing decentralization, albeit for a specific model of service provider-consumer interaction. In grid computing, service-level agreements (SLAs) [CFK+02] followed a similar template. This challenge has returned to salience in today's Web3 world, where private and independent blockchain systems have business imperatives to interoperate [ABG+19]. The interaction model common to these scenarios involves no a priori trust between the interacting parties, though they may, unbeknownst to each other, possess VCs (or more generally *certificates*) from common trust anchors (or more generally *certifiers*) attesting to different claims.

A trust basis for interoperation can be established between two parties if they can determine that they both possess valid certificates attesting to certain claims, and that these certificates are issued by one or more certifiers that they both trust. But this is hard to do in the absence of a priori trust or knowledge of the counterparty's intentions, or without compromising one's privacy. We can see why this is so by applying the standard VC recommendation, whereby one party makes a Verifiable Presentation (VP) [SLC21] to another, to our scenario. In a typical VC use case, the relationship between credential presenter and verifier is asymmetric, as the verifier is typically a well-known entity from whom the presenter seeks service or approval. The presenter knows at least one certifier that is trusted by it and the verifier. Typically, this requires the verifier to publish its complete list of certifiers so the presenter can determine ones that are commonly trusted by both parties [BFGP22]. But in our interaction model, the relationship between parties is symmetrical, as they are both trying to simultaneously prove something to the other. In a standard VP, the presenter is willing to share credentials (albeit selectively) with the verifier. But, if we use this asymmetric VP-based solution in our scenario where neither party knows anything about the other a priori, the revelation of credentials by the party that presents first will automatically give more leverage to the counterparty (verifier), which learns more about the presenter than it reveals.

A naïve adaptation of an asymmetric solution (such as [BFGP22]) to our symmetric setting would require both parties to reveal to each other the list of certifiers from which they have valid certificates, and then identify if there is a mutually trusted certifier. This entails complete loss of privacy for both parties, but especially for an honest party if the other behaves maliciously. There are strong reasons why revealing one's complete list of certifiers might not be in one's interest. A business-oriented certifier, for instance, might not like its clientele to be visible to its market competitors. Consider a blockchain interoperability scenario, where shipment carriers on different trade networks certify their respective networks' participants, e.g., Maersk Shipping Company (on the TradeLens network [tra]) and the American Bureau of Shipping (ABS). But as Maersk and ABS are

market competitors, they may not necessarily want their clients (the certificate holders) to reveal their respective associations [GVR+22]. Knowing the clientele of Maersk may benefit ABS, and vice versa; hence there is a privacy cost to revealing certifier lists in a symmetrical interaction unless those certifier lists are identical.

The other privacy violation aspect is from the perspective of the certificate holder. Every certificate possessed indicates an affiliation with some real world entity, often a well-known one; this could include government agencies, political organizations, NGOs, etc., and such affiliations might be sensitive information that could potentially be misused. And here lies the biggest hazard in the naïve trust basis establishment solution: one of the two interacting parties could be malicious and is trying to fish for information about its counterparty's affiliations. A simple attack would be for the malicious party to offer a long list of certifiers, regardless of whether it possesses valid certificates from them, and have the honest counterparty reveal its true certifier list. Now the malicious party knows, and can misuse, the honest party's affiliations, without revealing its own true affiliations. In the context of trust anchors (TAs) in the DID & VC world, where any entity can issue a VC and there does not exist a canonical list or registry of global TAs, it would not be a hard task for a malicious counterparty to list as many of them as possible to mount the attack we just described. Therefore, we can identify a compelling need to maintain certifier privacy and authenticity, which are not addressed by the naïve solution for determining common certifiers. This motivates us to ask the following question:

Can parties owning certificates efficiently identify a common set of certifiers without leaking anything else?

In particular, the parties should not learn any information about certifiers that may be in the lists of other parties but are not in the intersection.

1.1 Our Contributions

Private Certifier Intersection (PCI). In this paper, we initiate the study of Private Certifier Intersection (PCI) – a cryptographic primitive that aims to answer the above question in the affirmative. Informally speaking, a PCI protocol allows a set of mutually distrusting certificate-holding parties to achieve a privacy-preserving trust negotiation with the following objectives: (i) find an intersection among the set of certifiers across the parties, (ii) ensure that the certificates issued by these certifiers are valid, and (iii) reveal no information about the certifiers that may be in the lists of individual parties but are not in the intersection.

Comparison with Private Set Intersection. At a first glance, the classic *Private Set Intersection* (PSI) problem [PSZ14,CLR17], where the intersection of two private sets must be determined without a trusted mediator, bears a strong resemblance to PCI (also see Figure 1). In both PCI and PSI, a set of mutually distrusting parties holding private sets of entities aim to compute the intersection between their sets without revealing any additional information about the elements in their individual sets that are not in the intersection. However, the non-triviality of PCI arises from the need to additionally validate the certificates issued by the certifiers in the intersection. In this sense, one can think of PCI as a form of "predicated" PSI, where the inclusion of a common certifier in

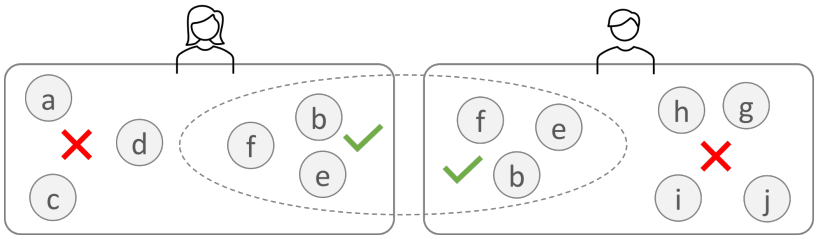


Figure 1: Private Set Intersection (PSI): Match Values

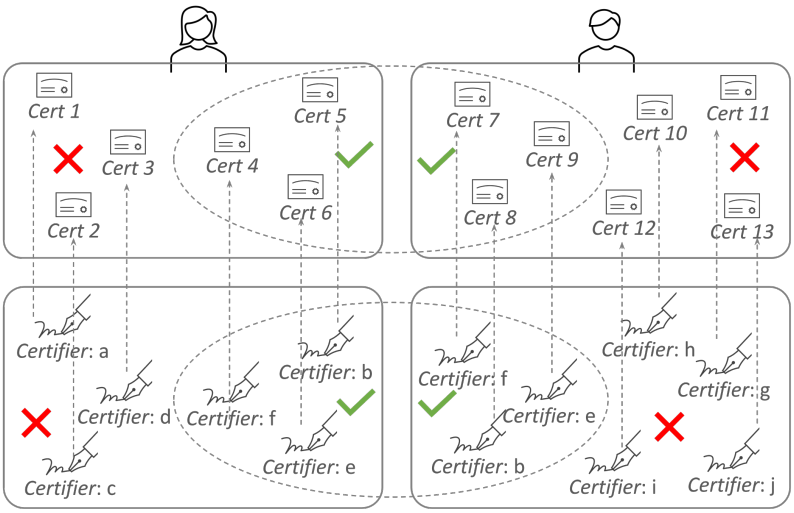


Figure 2: Private Certifier Intersection (PCI): Match Certificates with Common Issuers

the final output set is predicated on the certificates issued by this certifier to each of the parties being valid (see Figure 2 for an illustration). We argue in this paper that realizing an efficient PCI protocol with ideal security guarantees requires novel techniques beyond simply using PSI as a building block. Consider the hazard we encountered earlier in the naïve solution to establish a trust basis. Using standard PSI, a malicious party could simply supply a long (or universal) list of certifiers as input and determine the list of certifiers of the other (honest) party. To avoid this hazard, we need to enforce the ability of participants to prove that they possess genuine certificates issued by thir claimed certifiers. There is no obvious way to do this using standard PSI, and therefore PCI requires novel mechanisms that are not congruent to PSI's mechanisms. We refer the reader to Section 1.3 for additional related work.

Achieving Semi-Honest PCI. It turns out that in the setting of semi-honest corruptions (i.e., when the participating parties behave honestly as prescribed in the protocol), one can easily achieve a secure PCI protocol by using any secure PSI protocol in a black-box way. Consider the following simple construction: each party first locally "filters" its private list of certifiers based on the validity of the certificates issued by such certifiers, and then uses this filtered list of certifiers as its input to an execution of a PSI protocol to securely identify their intersection. Correctness is immediate, since, assuming honest behavior, the filtered list for each party only contains certifiers issuing valid certificates. Security follows from the security of the underlying PSI protocol.

Upgrading to Malicious Security. Unfortunately, in the setting of malicious corruptions (i.e., when the participating parties can deviate arbitrarily from the protocol), it is seemingly hard to achieve a secure PCI protocol by simply using certification validation and a (maliciously secure) PSI as individual black-boxes. To begin with, we cannot rely on the parties to filter the local sets of certifiers correctly; in fact, the parties can prepare arbitrary sets of certifiers, including those for which it does not have valid certificates.

For example, in the setting of two-party PCI, if one party (say Alice) provides a "universal set" of certifiers as input to a PSI protocol, it can learn the complete set of certifiers of the other party (say Bob). This attack may not be feasible in a general PSI setting where listing the entire range of values in an input set may be infeasible or prohibitively expensive, but is quite feasible in a PCI setting where the range of certifiers (trusted authorities) is limited. Therefore, it is crucial for both Alice and Bob to verify that the other is not faking its input set, and so the validity of certificates and the signatures within must be proven by both parties during the protocol. This is challenging because neither Alice nor Bob knows a priori which set of certifiers it needs to supply proof for (indeed, this is the objective of PCI), and providing more proof than strictly required (i.e., revealing certifiers outside the intersection) would violate privacy goals. Therefore, we must somehow intertwine certificate validation with a PSI-like protocol to achieve PCI. In other words, a maliciously secure PCI protocol cannot be achieved securely without a mechanism that somehow intertwines certificate validation with the subsequent PSI protocol.

Theoretically, a maliciously secure PCI protocol can be achieved as follows: run a maliciously secure multi-party computation (MPC) protocol for the functionality that: (i) filters the certifier list for each party to identify the certifiers issuing valid certificates attesting to the relevant claims, and (ii) computes the intersection between these filtered sets. This solution is highly inefficient in practice for essentially all widely used cryptographically secure certification mechanisms. For example, the most common method of generating certificates is to sign the claim using a digital signature algorithm. In this case, claim validation would require us to perform signature verifications inside the MPC protocol, which is prohibitively expensive for popular digital signature schemes such as ECDSA [ANS99, JMV01] and BLS [BLS01, BDN18, BGLS03], that rely on elliptic curve-based finite-field arithmetic operations. Implementing such a verification algorithm inside a maliciously secure MPC protocol would involve non-black-box usage of the various elliptic-curve (EC) operations, i.e., we would have to express these operations as (potentially complicated) binary/arithmetic circuits with gate operations over $\{0, 1\}$ or over some finite field F_p. Such a maliciously secure MPC protocol is likely to incur huge computational and communication overheads in practice.

Need for Efficient Protocols. The above discussion motivates specialized PCI protocols that efficiently enable computing the intersection of certifier-sets while: (i) achieving the desired security guarantees in the setting where a majority of the parties could be maliciously corrupt, and (ii) minimizing non-black-box usage of the operations in the certificate validation algorithm. In this paper, we design and implement two concrete PCI protocols – based on the ECDSA signature scheme and the BLS signature scheme – that achieve the above goal while supporting different variations of claim validation (we expand on this later). While our protocols broadly follow the generic approach outlined above, the main novelty lies in how we validate signatures while using the underlying elliptic curve-based operations in a black-box manner. For an (informal) comparison, the

generic MPC-based solution is expected to incur $O(xd)$ computation/communication cost, where x is the corresponding cost of our protocols, and d is the average depth of the arithmetic circuits representing EC operations (e.g., $d = 256$ for constant-time scalar multiplication over curve-ED25519 and curve-BLS12-381).

1.2 Overview of Contributions

In this section, we provide an informal overview of our key technical contributions.

Defining PCI. We formalize the security guarantees expected of a (multi-party) PCI protocol using the simplified universal composability (SUC) framework due to Canetti, Cohen, and Lindell [CCL15] in the real/ideal world paradigm. We consider two variations of PCI protocols in this paper:

- Validate-Any PCI: A PCI-Any protocol outputs the set of common certifiers for which each party has at least one valid certificate attesting to *any* one of its (publicly known) claims.
- Validate-All PCI: A PCI-All protocol outputs the set of common certifiers for which each party has valid certificates attesting to *all* of its (publicly known) claims.

We also consider a variant of validate-any PCI which we call validate-any PCI with disclosed claims (abbreviated as PCI-Any-DC) where, for each common certifier in the output set, the parties additionally learn the set of claims attested by the certifier. We refer to Section 2 for a formal description.

MPC for Elliptic Curve Pairings. As a fundamental building block of our proposed PCI protocols, we introduce a new secret-sharing based MPC framework that is tuned for elliptic curve pairings. Our overall approach is to design a secret-sharing based MPC protocol that efficiently supports basic elliptic curve operations (i.e., point addition and scalar multiplication) and elliptic curve bilinear pairing operations as fundamental building blocks. We build upon the SPDZ secret-sharing based MPC protocol [DPSZ12, DKL+13] to achieve the first secret-sharing based MPC framework that seamlessly supports elliptic curve pairing operations as fundamental gate-level building-blocks with malicious security against a dishonest majority of adversarial parties. A technical cornerstone of our framework is the round-preserving upgradation of SPDZ from basic field operations to the significantly more complicated elliptic curve operations, including pairings. Our framework allows us to directly use standardized and open-source implementations of elliptic curve libraries [ope, Lyn, AGM+], thereby leveraging both the performance improvements/optimizations as well as the protections against evolving implementation-level attacks that such libraries usually offer. We believe that this is a contribution of independent interest.

Efficient Two-Party PCI. We use our proposed MPC framework to design the following provably secure yet practically efficient two-party PCI protocols:

- A two-party PCI-Any-DC protocol using the ECDSA signature scheme [ANS99] – an elliptic-curve-based digital signature scheme which is standardized and widely

adopted in multiple real-world applications including X.509 public key infrastructure in the Internet, TLS [MBG+06], DNSSEC [HW12], etc. Moreover, ECDSA is a candidate signature scheme in verifiable credentials [SLC21] which is one of the target applications of PCI. Choosing ECDSA also allows us to use its standard implementation in the OpenSSL [ope] library for EC group operations. This naturally motivates designing a PCI protocol supporting ECDSA-based certification of claims.

- A two-party PCI-All protocol using the BLS signature scheme [BLS01, BDN18, BGLS03]– an elliptic-curve pairing-based digital signature that is popularly used in blockchain applications and is in the process of being standardized [BGW+20]. We design a PCI-All protocol supporting BLS-based certification of claims that exploits the signature-aggregation capabilities of BLS to perform efficient validation of certificates over all of the public claims of each party.

The starting point of our protocols is the generic maliciously secure PCI protocol outlined earlier, with several optimizations to obviate or minimize expensive elliptic curve operations inside the MPC protocol. In our ECDSA-based PCI-Any-DC protocol, we develop techniques that enable securely yet efficiently performing the expensive algebraic operations (such as field inversion) and non-algebraic operations (such finding the x-coordinate of an elliptic curve point) required by the ECDSA verification algorithm *outside* the MPC protocol. The protocol is then implemented using our proposed MPC framework, which allows performing ECDSA signature validations while using all elliptic curve operations in a black-box manner. We also discuss how to upgrade this protocol to full-fledged PCI-Any where the claims are no longer disclosed publicly (see Section 4 for details).

Trivially extending the approach used in our ECDSA-based PCI-Any-DC protocol to design a PCI-All protocol would require iterating through all of the public claims, and validating the signatures on these claims by a specific certifier. This results in a claim validation complexity that grows with the number of claims. We overcome this challenge by designing a PCI-All protocol using BLS-based signature-aggregation that only requires a single (aggregate-)signature verification per certifier inside the MPC protocol. We introduce additional optimizations that exploit the deterministic nature of the BLS signature to further reduce the number of elliptic curve pairing operations inside MPC to just one per certifier, which is then implemented in a black-box manner using our proposed MPC framework over pairings.

Implementation and Evaluation. We extend MP-SPDZ [Kel20] to implement our proposed secret-sharing framework supporting elliptic curve operations including bilinear pairings. For the black-box operations on elliptic curves we use OpenSSL [ope] and RELIC [AGM+] libraries. We then implement ECDSA-based PCI-Any-DC and BLS-based PCI-All protocols. We make the source code of our implementation available at `https://github.com/ghoshbishakh/pci` for independent benchmarking. We provide a detailed analysis of the performance of the individual components of our MPC framework, followed by the end-to-end performance evaluation of the protocols in realistic setups by placing parties in three geographic regions across two continents. In an intercontinental WAN setup with parties located in different continents, our PCI-Any-DC and PCI-All protocols execute in less than a minute on input sets of size 40. This demonstrates the practicality of our proposed solutions. We refer to Section 6 for details.

1.3 Related Work

Private Set Intersection (PSI). Private set intersection (PSI) [PSZ14] has been extensively studied, with a wide range of solutions based on garbled circuits [HEK12], homomorphic encryption [CLR17], oblivious transfer [PSZ14], and other techniques [PSSZ15, RR17, KS05, FNP04, DSMRY09, CT10, CM20]. However, as outlined earlier, there is no straightforward way of using PSI as a black-box to achieve PCI, particularly in the face of malicious adversarial corruptions, due to the additional requirement of certificate validation.

PSI over Certified Sets. Private intersection of "certified sets", introduced in [CZ09], is an augmentation of PSI with the additional requirement that the input claim-sets are certified by some certification authority (CA). However, this primitive has fundamentally different privacy goals as compared to PCI; it assumes that the information of the CAs is public and that the two parties agree apriori on which CAs they mutually trust. Conversely, in the case of PCI, the CAs (certifiers) are, in fact, the input to the protocol (and thus cannot be made public apriori) while the claims are public. We could also have a variant of PCI where the claims are additionally private; we leave it as an open question to investigate this variant further.

HIAC. Hidden-issuer anonymous credentials (HIAC), introduced in [BFGP22], is an elegant cryptographic primitive that allows a credential holder to prove its claim(s) to a verifier without disclosing the identity of the credential issuer (i.e., the certifier). However, HIAC inherently requires the set of certifiers trusted by the verifier to be published as an "aggregator", thereby revealing the identity of each such certifier. Hence, while one could use HIAC to solve the same problem at PCI, such an adaptation would only achieve *one-sided privacy* since of the parties would have to make its list of certifiers publicly available. On the other hand, PCI aims to enable *two-sided privacy* by allowing the two parties to negotiate their common certifiers while preserving the privacy of *both* individual lists, and while simultaneously validating the certificates issued to *both* the parties.

IHABC. Issuer-Hidden Attribute-Based Credential [BEK+21] is another related system in which a user can prove a credential issued to it without revealing which issuer among a set of issuers acceptable to the verifier issued that credential. Similar to HIAC, this system also provides one-sided privacy while revealing the certifier set of the verifier (PCI, on the other hand, ensures privacy of both the parties' list of certifiers). Moreover, the concrete solution presented in [BEK+21] uses a trusted setup, which is costly in practice and is not a requirement for any of our PCI solutions.

Secret Handshake. The "secret handshake" family of protocols [BDS+03, AKB07] enable (role-based) authenticated key exchange between parties without revealing any information beyond the common group memberships shared by the parties. These protocols, however, differ fundamentally from PCI in the sense that: (a) they do not capture the notion of validating certificates and claims (which is one of the core requirements addressed by PCI), and (b) the process of issuing membership credentials is part of the protocol itself (in PCI, the process of issuing credentials/certificates is not considered part of the primitive).

2 Private Certifier Intersection (PCI)

In this section, we formally define Private Certifier Intersection (PCI). We begin by introducing some notations and background material. We subsequently formalize the functionality and security guarantees that a PCI protocol should satisfy.

General Notations. We write $x \leftarrow \chi$ to represent that an element x is sampled uniformly at random from a set/distribution $\mathcal{X}$. The output x of a deterministic algorithm $\mathcal{A}$ is denoted by $x = \mathcal{A}$ and the output x' of a randomized algorithm $\mathcal{A}'$ is denoted by $x' \leftarrow \mathcal{A}'$. For $a, b \in \mathbb{N}$ such that $a, b \geq 1$, we denote by $[a, b]$ the set of integers lying between a and b (both inclusive). We refer to $\lambda \in \mathbb{N}$ as the security parameter, and denote by $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ any generic (unspecified) polynomial function and negligible function in λ, respectively.[1]

PCI Notations. Let $\mathcal{ID}$ be a set of identities corresponding to the certifiers. Given a claim $\mathsf{m} \in \mathcal{M}$ by a party P, a certifier with identity id can issue a certificate $\sigma \in \mathcal{C}$, such that there exists a relation $\mathbf{R}$ that satisfies the following:

$$\mathbf{R}(\mathsf{id}, \sigma, \mathsf{m}) = 1 \text{ iff } \sigma \text{ is a valid certificate by } \mathsf{id} \text{ on } \mathsf{m}$$

A natural instantiation of the certification process outlined above is a digital signature, where the certificate issuance corresponds to the signing algorithm and the relation $\mathbf{R}$ corresponds to the verification algorithm, with σ being the signature on a claim m under the signing key corresponding to id. Looking ahead, our proposed realizations of PCI protocols in this paper will use this digital signature-based instantiation of the certification process.

We now introduce some additional notations for ease of exposition, these notations will be useful in understanding our definitions for PCI. Let S be a set of (identity, certificate, claim) tuples of the form

$$S = \{(\mathsf{id}_j, \sigma_j, \mathsf{m}_j) \in \mathcal{ID} \times \mathcal{C} \times \mathcal{M}\}_{j \in [1,n]}$$

where N is the number of tuples in the set S. We define the following projection functions on the set S:

$$\begin{aligned}
\mathsf{id}(S) &:= \{\mathsf{id} : \exists \sigma, \mathsf{m} \text{ s.t. } (\mathsf{id}, \sigma, \mathsf{m}) \in S\} \\
\mathsf{m}(S) &:= \{\mathsf{m} : \exists \mathsf{id}, \sigma \text{ s.t. } (\mathsf{id}, \sigma, \mathsf{m}) \in S\} \\
\overline{\mathsf{m}}(S) &:= (\mathsf{m}_j)_{(\mathsf{id}_j, \sigma_j, \mathsf{m}_j) \in S}
\end{aligned}$$

Here, $\overline{\mathsf{m}}(S)$ is a list/multiset of the claims corresponding to each tuple in the set S.

2.1 Defining a Two-Party PCI Protocol

We now formally define a PCI protocol in the two-party setting, which is the focus of this paper. Our definitions naturally extend to multiple parties, as discussed subsequently.

[1] Note that a function $f : \mathbb{N} \to \mathbb{N}$ is said to be negligible in λ if for every positive polynomial p, $f(\lambda) < 1/p(\lambda)$ when λ is sufficiently large.

Two-Party PCI. A two-party PCI protocol Π involves parties P_1 and P_2, where each party P_i for $i \in \{1, 2\}$ inputs a tuple of the form $\mathsf{inp}_i = \big(\mathsf{inp}_{i,1}, \mathsf{inp}_{i,2}\big)$, where:

- The *private* input $\mathsf{inp}_{i,1}$ is a set of (identity, certificate, claim) tuples of the form

$$\mathsf{inp}_{i,1} = \{(\mathsf{id}_{i,j}, \sigma_{i,j}, \mathsf{m}_{i,j}) \in \mathcal{ID} \times \mathcal{C} \times \mathcal{M}\}_{j \in [1, N_i]}$$

  where N_i is the number of tuples in $\mathsf{inp}_{i,1}$ from party P_i.

- The *public* input $\mathsf{inp}_{i,2}$ is a set of claims of the form $\{\widehat{\mathsf{m}}_{i,j} \in \mathcal{M}\}_{j \in [1, N'_i]}$, where N'_i is the number of tuples in $\mathsf{inp}_{i,2}$ from party P_i.

Note that a party P_i can produce multiple certificates from the same certifier on same or different claims. Additionally, a party P_i can also request certifications on the same claim from multiple certifiers. Hence, in the most general setting, a party's input could have multiple tuples with the common id or a common m. Also note that the public input for P_1 is known to P_2 at the start of the protocol and vice versa.[2]

Remark. A couple of remarks on the definition follow:

1. One could have a variant of PCI with the claims being private. This work considers the above defined variant with the claims being public. We leave it to future work for instantiating PCI with private claims.
2. Our definition lets a (corrupt) party provide claims in the public input that are different from those in the tuple in the private input. One could also restrict the public input $\mathsf{inp}_{i,2}$ to be $\mathsf{m}(\mathsf{inp}_{i,1})$, which is the expected behaviour of the honest parties.

At the end of the protocol Π, each party P_i receives as output a set of certifiers. In this paper, we consider different variations of (two-party) PCI protocols that produce different kinds of output sets, that we outline below:

- **Validate-Any:** In this flavor of PCI protocol, denoted by $\mathsf{PCI\text{-}Any}$, both parties P_1 and P_2 receive as output the set of certifiers $\mathsf{out}_{\mathsf{PCI\text{-}Any}}$, such that an identity $\mathsf{id} \in \mathsf{out}$ if and only if both P_1 and P_2 have valid certificates on some $\mathsf{m}_1 \in \mathsf{inp}_{i,2}$ and $\mathsf{m}_2 \in \mathsf{inp}_{i,2}$, respectively, such that both the certificates are issued by id. More formally, for each $i \in \{1, 2\}$, we define the following Boolean predicate:

$$\begin{aligned}\mathbf{R}_{\mathsf{PCI\text{-}Any},\mathsf{inp}_i}(\mathsf{id}) = 1 \text{ if and only if } \exists \mathsf{m} \in \mathsf{inp}_{i,2} :\\ \exists (\mathsf{id}, \mathsf{m}, \sigma) \in \mathsf{inp}_{i,1} \text{ s.t. } \mathbf{R}(\mathsf{id}, \mathsf{m}, \sigma) = 1\end{aligned}$$

  Then we have

$$\begin{aligned}\mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \mathsf{inp}_2) = \big\{\mathsf{id} \in \mathsf{id}(\mathsf{inp}_{1,1}) \cap \mathsf{id}(\mathsf{inp}_{2,1}) :\\ \mathbf{R}_{\mathsf{PCI\text{-}Any},\mathsf{inp}_1}(\mathsf{id}) = \mathbf{R}_{\mathsf{PCI\text{-}Any},\mathsf{inp}_2}(\mathsf{id}) = 1\big\}\end{aligned}$$

[2] We assume that these sets are shared between P_1 and P_2 via some apriori mechanism that is not within the purview of the PCI protocol itself.

- **Validate-Any with Disclosed Claims:** We also consider a weaker variant of the aforementioned validate-any PCI protocol (denoted by $\mathsf{PCI\text{-}Any\text{-}DC}$), where the parties additionally learn the following: (i) the claim $\mathsf{m}_{i,j}$ corresponding to each tuple $(\mathsf{id}_{i,j}, \sigma_{i,j}, \mathsf{m}_{i,j}) \in \mathsf{inp}_{i,1}$ for each party P_i, (ii) for each id in the output set of certifiers $\mathsf{out}_{\mathsf{PCI\text{-}Any}}$, each party learns the set of (public) claims on which the other party has a valid certificate issued by id. Note that no information is revealed about any (valid/invalid) certificates that the parties might have that are issued by some $\mathsf{id}' \notin \mathsf{out}_{\mathsf{PCI\text{-}Any}}$. Formally, for each $i \in \{1, 2\}$, we define the function

$$\mathsf{m}_{\mathsf{inp}_i}(\mathsf{id}) = \big\{\mathsf{m} : \exists(\mathsf{id}, \mathsf{m}, \sigma) \in \mathsf{inp}_{i,1} \text{ s.t. } \mathbf{R}(\mathsf{id}, \mathsf{m}, \sigma) = 1\big\}$$

  Then the output set $\mathsf{out}_{\mathsf{PCI\text{-}Any\text{-}DC}}$ is described formally as follows

$$\mathsf{out}_{\mathsf{PCI\text{-}Any\text{-}DC}}(\mathsf{inp}_1, \mathsf{inp}_2) = \Big(\big\{\overline{\mathsf{m}}(\mathsf{inp}_{i,1})\big\}_{i\in[1,2]}, \big\{\big(\mathsf{id}, \{\mathsf{m}_{\mathsf{inp}_i}(\mathsf{id})\}_{i\in\{1,2\}}\big) : \mathsf{id} \in \mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \mathsf{inp}_2)\big\} \Big)$$

  $\mathsf{PCI\text{-}Any\text{-}DC}$ is relevant in most real-world scenarios since the parties would know the claims of the counterparty that they want to validate, and vice versa. Moreover, traditional VC interactions also work on disclosed claims (see Section 1).

- **Validate-All:** In this flavor of PCI protocol, denoted by $\mathsf{PCI\text{-}All}$, both parties P_1 and P_2 receive as output the set of certifiers $\mathsf{out}_{\mathsf{PCI\text{-}All}}$, such that for each $\mathsf{id} \in \mathsf{out}_{\mathsf{PCI\text{-}All}}$, P_1 and P_2 have valid certificates issued by id on *all* of the (public) claims in their input sets $\mathsf{inp}_{1,2}$ and $\mathsf{inp}_{2,2}$, respectively. More formally, for each $i \in \{1, 2\}$, we define the following Boolean predicate:

$$\mathbf{R}_{\mathsf{PCI\text{-}All},\mathsf{inp}_i}(\mathsf{id}) = 1 \text{ if and only if } \forall \mathsf{m} \in \mathsf{inp}_{i,2} : \exists(\mathsf{id}, \mathsf{m}, \sigma) \in \mathsf{inp}_{i,1} \text{ s.t. } \mathbf{R}(\mathsf{id}, \mathsf{m}, \sigma) = 1$$

  Then we have

$$\mathsf{out}_{\mathsf{PCI\text{-}All}}(\mathsf{inp}_1, \mathsf{inp}_2) = \big\{\mathsf{id} \in \mathsf{id}(\mathsf{inp}_{1,1}) \cap \mathsf{id}(\mathsf{inp}_{2,1}) : \mathbf{R}_{\mathsf{PCI\text{-}All},\mathsf{inp}_1}(\mathsf{id}) = \mathbf{R}_{\mathsf{PCI\text{-}All},\mathsf{inp}_2}(\mathsf{id}) = 1\big\}$$

2.2 Security of Two-Party PCI

We now define the security guarantees expected of a PCI protocol in the two-party setting. Informally, we require that in any PCI protocol Π, party P_1 (resp. party P_2) learns nothing about the inputs of party P_2 (resp. party P_1) except what is revealed by the output out of the protocol Π, and the sizes N_1 and N_2 of the input sets of P_1 and P_2. In the rest of this section, we formalize this security guarantee using the simplified universal composability (SUC) framework due to Canetti, Cohen, and Lindell [CCL15] in the real/ideal world paradigm. We consider a *dishonest majority* in our definitions, wherein the adversary can corrupt one of the two participating parties. For ease of exposition, we assume without loss of generality that P_1 and P_2 are the corrupt party and the honest party, respectively.

Ideal Functionality for Two-Party PCI. We begin by formally defining the first component of our simulation-based security definition, namely the ideal functionality $\mathcal{F}_{\mathsf{PCI}}$, as described in Figure 3. This functionality $\mathcal{F}_{\mathsf{PCI}}$ formally defines what each party is meant to learn at the completion of the protocol.

$\underline{\mathcal{F}_{\mathsf{PCI}}(\mathsf{mode} \in \{\mathsf{Any}, \mathsf{Any\text{-}DC}, \mathsf{All}\})}$

- For $i \in \{1, 2\}$, let the input of party P_i be $\mathsf{inp}_i = (\mathsf{inp}_{i,1}, \mathsf{inp}_{i,2})$, where

$$\mathsf{inp}_{i,1} = \{(\mathsf{id}_{i,j}, \sigma_{i,j}, \mathsf{m}_{i,j}) \in \mathcal{ID} \times \mathcal{C} \times \mathcal{M}\}_{j \in [1, N_i]}$$
$$\mathsf{inp}_{i,2} = \{\widehat{\mathsf{m}}_{i,j} \in \mathcal{M}\}_{j \in [1, N'_i]}$$

  The honest party P_2 provides its input directly to $\mathcal{F}_{\mathsf{PCI}}$, while the input of the corrupt party P_1 is provided to $\mathcal{F}_{\mathsf{PCI}}$ by the simulator $\mathcal{S}$.

- $\mathcal{F}_{\mathsf{PCI}}$ computes $\mathsf{out}_{\mathsf{PCI\text{-}mode}}$, where for $\mathsf{mode} \in \{\mathsf{Any}, \mathsf{Any\text{-}DC}, \mathsf{All}\}$, we have

$$\mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \mathsf{inp}_2) = \big\{\mathsf{id} \in \mathsf{id}(\mathsf{inp}_{1,1}) \cap \mathsf{id}(\mathsf{inp}_{2,1}) : \mathbf{R}_{\mathsf{PCI\text{-}Any}, \mathsf{inp}_1}(\mathsf{id}) = \mathbf{R}_{\mathsf{PCI\text{-}Any}, \mathsf{inp}_2}(\mathsf{id}) = 1\big\}$$
$$\mathsf{out}_{\mathsf{PCI\text{-}Any\text{-}DC}}(\mathsf{inp}_1, \mathsf{inp}_2) = \Big(\big\{\overline{\mathsf{m}}(\mathsf{inp}_{i,1})\big\}_{i \in [1,2]}, \big\{ \big(\mathsf{id}, \{\mathsf{m}_{\mathsf{inp}_i}(\mathsf{id})\}_{i \in \{1,2\}}\big) : \mathsf{id} \in \mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \mathsf{inp}_2)\big\}\Big)$$
$$\mathsf{out}_{\mathsf{PCI\text{-}All}}(\mathsf{inp}_1, \mathsf{inp}_2) = \big\{\mathsf{id} \in \mathsf{id}(\mathsf{inp}_{1,1}) \cap \mathsf{id}(\mathsf{inp}_{2,1}) : \mathbf{R}_{\mathsf{PCI\text{-}All}, \mathsf{inp}_1}(\mathsf{id}) = \mathbf{R}_{\mathsf{PCI\text{-}All}, \mathsf{inp}_2}(\mathsf{id}) = 1\big\}$$

- $\mathcal{F}_{\mathsf{PCI}}$ sends $\big(\mathsf{out}_{\mathsf{PCI\text{-}mode}}, N_1, \mathsf{inp}_{1,2}\big)$ to the simulator $\mathcal{S}$.
- If $\mathcal{S}$ responds with an abort, $\mathcal{F}_{\mathsf{PCI}}$ aborts.
- Otherwise, $\mathcal{F}_{\mathsf{PCI}}$ sends to P_1 and P_2 the tuple

$$\big(\mathsf{out}_{\mathsf{PCI\text{-}mode}}, N_1, N_2, \mathsf{inp}_{1,2}, \mathsf{inp}_{2,2}\big)$$

Figure 3: Ideal functionality $\mathcal{F}_{\mathsf{PCI}}$ in the two-party setting

The Real/Ideal World Paradigm. The real and ideal worlds are as follows.

- In the real world, the honest parties follow the specified protocol Π and interact with the maliciously corrupt parties who are controlled by an adversary $\mathcal{A}$.
- In the ideal world, the honest parties and an adversary called the simulator (denoted by $\mathcal{S}$) interact with the ideal functionality $\mathcal{F}_{\mathsf{PCI}}$ (described in Figure 3), where $\mathcal{S}$ controls the same set of maliciously corrupt parties as in the real world.

Informally speaking, in the ideal world, $\mathcal{S}$ learns nothing beyond what $\mathcal{F}_{\mathsf{PCI}}$ reveals to the corrupt parties. As is the norm in the SUC model, we also introduce an additional adversary called the **environment** $\mathcal{Z}$ that interacts with adversary $\mathcal{A}$ in the real world, and with the simulator $\mathcal{S}$ in the ideal world. Its goal is to distinguish these two interactions. If $\mathcal{Z}$ cannot distinguish between the real and ideal worlds, we say that the real-world PCI protocol Π securely emulates the ideal functionality $\mathcal{F}_{\mathsf{PCI}}$. We now formalize this paradigm below.

Real World. In the real world, the following participants engage in the protocol Π:

- The honest party P_2 that receives its input from the environment $\mathcal{Z}$ and honestly follows the protocol Π.
- A real-world adversary $\mathcal{A}$ that controls the corrupt party P_1, and interacts with P_2 and the environment $\mathcal{Z}$.
- The environment $\mathcal{Z}$ that provides P_2 with its input, and interacts with the real-world adversary $\mathcal{A}$. The environment $\mathcal{Z}$ also receives the output of P_2, and eventually outputs a bit $b \in \{0,1\}$.

Ideal World. In the ideal world, the following participants interact with the ideal functionality $\mathcal{F}_{\mathsf{PCI}}$ described in Figure 3.

- The honest party P_2 that receives its input from the environment $\mathcal{Z}$ and directly forwards this input to $\mathcal{F}_{\mathsf{PCI}}$.
- An ideal-world simulator $\mathcal{S}$ that sends inputs to $\mathcal{F}_{\mathsf{PCI}}$ on behalf of the corrupt party P_1 and receives back the corresponding output from $\mathcal{F}_{\mathsf{PCI}}$. $\mathcal{S}$ also interacts with the environment $\mathcal{Z}$, with the aim of making this interaction indistinguishable from the interaction between the real world $\mathcal{A}$ and the environment $\mathcal{Z}$.
- The environment $\mathcal{Z}$ that provides P_2 with its input, and interacts with the simulator $\mathcal{S}$. As in the real world, $\mathcal{Z}$ also receives the output of P_2, and eventually outputs a bit $b \in \{0,1\}$.

For any two-party PCI protocol Π, any adversary $\mathcal{A}$, any simulator $\mathcal{S}$, and any environment $\mathcal{Z}$, define the following random variables:

- $\mathsf{real}_{\Pi,\mathcal{A},\mathcal{Z}}$: denotes the output of the environment $\mathcal{Z}$ after interacting with the adversary $\mathcal{A}$ during an execution of the real-world protocol Π.
- $\mathsf{ideal}_{\mathcal{F}_{\mathsf{PCI}},\mathcal{S},\mathcal{Z}}$: denotes the output of the environment $\mathcal{Z}$ after interacting with the simulator $\mathcal{S}$ in the ideal world.

Definition 1 (Secure Two-Party PCI)**.** A PCI protocol Π securely emulates the ideal functionality $\mathcal{F}_{\mathsf{PCI}}$ described in Figure 3 if for any security parameter $\lambda \in \mathbb{N}$ and any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that, for any PPT environment $\mathcal{Z}$,

$$\left|\Pr\left[\mathsf{real}_{\Pi,\mathcal{A},\mathcal{Z}} = 1\right] - \Pr\left[\mathsf{ideal}_{\mathcal{F}_{\mathsf{PCI}},\mathcal{S},\mathcal{Z}} = 1\right]\right| \leq \mathsf{negl}(\lambda)$$

2.3 Multi-Party PCI

In this section, we extend the definition of two-party PCI in Section 2 to the more general setting of multi-party PCI involving n parties $P_1, \ldots, P_n$. Similar to a two-party PCI protocol, in a multi-party PCI protocol, each party P_i for $i \in [1,n]$ inputs a tuple of the form $\mathsf{inp}_i = \left(\mathsf{inp}_{i,1}, \mathsf{inp}_{i,2}\right)$. We consider the following analogous variations of multi-party PCI protocols:

- **Validate-Any:** Each party P_i for $i \in [1, n]$ receives as output the set

$$\mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n) = \big\{\mathsf{id} \in \bigcap_{i \in [1,n]} \mathsf{id}(\mathsf{inp}_{i,1}) : \\ \forall i \in [1, n] \;\; \mathbf{R}_{\mathsf{PCI\text{-}Any},\mathsf{inp}_i}(\mathsf{id}) = 1\big\}$$

  where $\mathbf{R}_{\mathsf{PCI\text{-}Any},\mathsf{inp}_i}(\cdot)$ for each $i \in [1, n]$ is as defined in the two-party case.

- **Leaky Validate-Any:** Each party P_i for $i \in [1, n]$ receives as output the set

$$\mathsf{out}_{\mathsf{PCI\text{-}Any\text{-}DC}}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n) = \Big\{ \big(\mathsf{id}, \{\mathsf{m}_{\mathsf{inp}_i}(\mathsf{id})\}_{i \in [1,n]}\big) : \\ \mathsf{id} \in \mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n)\Big\}$$

  where $\mathsf{m}_{\mathsf{inp}_i}(\cdot)$ for each $i \in [1, n]$ is again as defined in the two-party case.

- **Validate-All:** Each party P_i for $i \in [1, n]$ receives as output the set

$$\mathsf{out}_{\mathsf{PCI\text{-}All}}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n) = \big\{\mathsf{id} \in \bigcap_{i \in [1,n]} \mathsf{id}(\mathsf{inp}_{i,1}) : \\ \forall i \in [1, n] \;\; \mathbf{R}_{\mathsf{PCI\text{-}All},\mathsf{inp}_i}(\mathsf{id}) = 1\big\}$$

  where $\mathbf{R}_{\mathsf{PCI\text{-}All},\mathsf{inp}_i}(\cdot)$ for each $i \in [1, n]$ is again as defined in the two-party case.

Security of Multi-Party PCI. We now define the security guarantees expected of a multi-party PCI protocol. Similar to the two-party setting, informally, we require that in any multi-party PCI protocol Π, each party P_i learns nothing about the inputs of the other parties P_j for $j \neq i$ except what is revealed by the output out of the protocol Π, and the size N_j of the input set of party P_j. We again formalize this security guarantee using the simplified universal composability (SUC) framework due to Canetti, Cohen, and Lindell [CCL15] in the real/ideal world paradigm. Our definition is a natural generalization of the security definition of two-party PCI in Section 2 to the multi-party setting. We again consider a *dishonest majority* in our definitions, wherein the adversary can corrupt upto $(n-1)$ parties in an n-party PCI protocol.

Ideal Functionality for Multi-Party PCI. We begin by formally defining the ideal functionality $\mathcal{F}_{\mathsf{PCI}}^{(n)}$ for n-party PCI, as described in Figure 4. This ideal functionality is basically a generalization of the ideal functionality $\mathcal{F}_{\mathsf{PCI}}$ for two-party PCI (defined earlier in Figure 3) to the n-party setting.

The real and ideal worlds are defined analogously to the two-party setting. Now, for any multi-party PCI protocol Π, any adversary $\mathcal{A}$, any simulator $\mathcal{S}$, and any environment $\mathcal{Z}$, define the following random variables:

- $\mathsf{real}_{\Pi,\mathcal{A},\mathcal{Z}}$: denotes the output of the environment $\mathcal{Z}$ after interacting with the adversary $\mathcal{A}$ during an execution of the real-world protocol Π.
- $\mathsf{ideal}_{\mathcal{F}_{\mathsf{PCI}}^{(n)},\mathcal{S},\mathcal{Z}}$: denotes the output of the environment $\mathcal{Z}$ after interacting with the simulator $\mathcal{S}$ in the ideal world.

$\underline{\mathcal{F}_{\mathsf{PCI}}^{(n)}(\mathsf{mode} \in \{\mathsf{Any}, \mathsf{Any\text{-}DC}, \mathsf{All}\})}$

- Let $H \subseteq [1,n]$ be the set of honest parties, and let $C \subseteq [1,n]$ be the set of corrupt parties.
- For each $i \in [1,n]$, let the input of party P_i be $\mathsf{inp}_i = (\mathsf{inp}_{i,1}, \mathsf{inp}_{i,2})$, where

$$\mathsf{inp}_{i,1} = \{(\mathsf{id}_{i,j}, \sigma_{i,j}, \mathsf{m}_{i,j}) \in \mathcal{ID} \times \mathcal{C} \times \mathcal{M}\}_{j \in [1,N_i]}$$
$$\mathsf{inp}_{i,2} = \{\widehat{\mathsf{m}}_{i,j} \in \mathcal{M}\}_{j \in [1,N_i']}$$

  For each $i \in H$, the input of an honest party P_i is provided directly to $\mathcal{F}_{\mathsf{PCI}}^{(n)}$ by P_i, while for each $i \in C$, the input of a corrupt party P_i is provided to $\mathcal{F}_{\mathsf{PCI}}^{(n)}$ by the simulator $\mathcal{S}$.
- $\mathcal{F}_{\mathsf{PCI}}^{(n)}$ computes $\mathsf{out}_{\mathsf{PCI\text{-}mode}}$, where for $\mathsf{mode} \in \{\mathsf{Any}, \mathsf{Any\text{-}DC}, \mathsf{All}\}$, we have

$$\mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n) = \Big\{\mathsf{id} \in \bigcap_{i \in [1,n]} \mathsf{id}(\mathsf{inp}_{i,1}) : \forall i \in [1,n] \;\; \mathbf{R}_{\mathsf{PCI\text{-}Any}, \mathsf{inp}_i}(\mathsf{id}) = 1\Big\}$$
$$\mathsf{out}_{\mathsf{PCI\text{-}Any\text{-}DC}}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n) = \Big\{\Big(\mathsf{id}, \{\mathsf{m}_{\mathsf{inp}_i}(\mathsf{id})\}_{i \in [1,n]}\Big) : \mathsf{id} \in \mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n)\Big\}$$
$$\mathsf{out}_{\mathsf{PCI\text{-}All}}(\mathsf{inp}_1, \ldots, \mathsf{inp}_n) = \Big\{\mathsf{id} \in \bigcap_{i \in [1,n]} \mathsf{id}(\mathsf{inp}_{i,1}) : \forall i \in [1,n] \;\; \mathbf{R}_{\mathsf{PCI\text{-}All}, \mathsf{inp}_i}(\mathsf{id}) = 1\Big\}$$

- $\mathcal{F}_{\mathsf{PCI}}^{(n)}$ sends $\big(\mathsf{out}_{\mathsf{PCI\text{-}mode}}, \{N_i, \mathsf{inp}_{i,2}\}_{i \in H}\big)$ to the simulator $\mathcal{S}$.
- If $\mathcal{S}$ responds with an abort, $\mathcal{F}_{\mathsf{PCI}}^{(n)}$ aborts.
- Otherwise, $\mathcal{F}_{\mathsf{PCI}}^{(n)}$ sends $\big(\mathsf{out}_{\mathsf{PCI\text{-}mode}}, \{N_i, \mathsf{inp}_{i,2}\}_{i \in [1,n]}\big)$ to all the parties.

Figure 4: Ideal functionality $\mathcal{F}_{\mathsf{PCI}}^{(n)}$ for multi-party PCI

Definition 2 (Secure Multi-Party PCI)**.** A multi-party PCI protocol Π securely emulates the ideal functionality $\mathcal{F}_{\mathsf{PCI}}^{(n)}$ described in Figure 4 if for any security parameter $\lambda \in \mathbb{N}$ and any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that, for any PPT environment $\mathcal{Z}$, we have

$$\Big|\Pr\left[\mathsf{real}_{\Pi,\mathcal{A},\mathcal{Z}} = 1\right] - \Pr\left[\mathsf{ideal}_{\mathcal{F}_{\mathsf{PCI}}^{(n)},\mathcal{S},\mathcal{Z}} = 1\right]\Big| \leq \mathsf{negl}(\lambda)$$

2.4 Generic Construction of Multi-Party PCI

In this section, we describe a generic approach to achieving a semi-honest secure (multi-party) PCI-mode protocol for mode $\in \{\mathsf{Any}, \mathsf{Any\text{-}DC}, \mathsf{All}\}$ given any semi-honest secure private set intersection (PSI) protocol. We then discuss some challenges that we face when attempting to upgrade this generic construction to provide malicious security.

Semi-Honest Secure PCI. We show how to construct a semi-honest secure multi-party PCI-Any protocol $\pi_{\mathsf{PCI\text{-}Any,Generic}}$ given a semi-honest secure multi-party PSI protocol π_{PSI}. The constructions of PCI-Any-DC and PCI-All follow analogously.

Suppose that each party P_i for $i \in [1, n]$ inputs a tuple of the form $\mathsf{inp}_i = \left(\mathsf{inp}_{i,1}, \mathsf{inp}_{i,2}\right)$, where $\mathsf{inp}_{i,1} = \{(\mathsf{id}_{i,j}, \sigma_{i,j}, \mathsf{m}_{i,j})\}_{j\in[1,N_i]}$ and $\mathsf{inp}_{i,2} = \{\widehat{\mathsf{m}}_{i,j} \in \mathcal{M}\}_{j\in[1,N'_i]}$. The parties proceed as described in Algorithm 1. At a high level, the protocol proceeds in two phases:

- **Phase-1: Filtering.** In this phase, each party filters its input set of (identity, certificate, claim) tuples to identify the subset of identities under which it has a valid certificate on *at least one* public claim.
- **Phase-2: PSI.** The parties then run the secure PSI protocol with these filtered subset of identities as inputs and output the resulting set as the output of the PCI-Any protocol.

Correctness is immediate. Since semi-honest corruption precludes the possibility of malicious behavior, semi-honest security of the overall protocol follows immediately from the semi-honest security of the underlying PSI protocol. Finally, it is straightforward to appropriately modify this protocol for: (i) PCI-Any-DC by additionally including in the filtered subset of identities the set of public claims for which each party has valid certificates under each identity, and (ii) PCI-All by changing Phase-1 to identify the subset of identities under which a party has a valid certificates on *all* of its public claims.

Algorithm 1: $\pi_{\mathsf{PCI\text{-}Any,Generic}}$ **from** π_{PSI}

1 **for** $i := 1 \ldots n$ **do**

2 Each party P_i locally computes a filtered set of identities as:

$$\mathsf{inp}'_i = \left\{\mathsf{id} \in \mathsf{id}(\mathsf{inp}_{1,1}) : \mathbf{R}_{\mathsf{PCI\text{-}Any},\mathsf{inp}_1}(\mathsf{id}) = 1\right\}$$

3 The parties $P_1, \ldots, P_n$ then run the PSI protocol π_{PSI} on the filtered input sets $\left(\mathsf{inp}'_1, \ldots, \mathsf{inp}'_n\right)$ to compute an output set

$$\mathsf{out}_{\mathsf{PSI}} = \bigcap_{i\in[1,n]} \mathsf{inp}'_i.$$

4 The parties output $\mathsf{out}_{\mathsf{PCI\text{-}Any}} := \mathsf{out}_{\mathsf{PSI}}$ as the output of the PCI-Any protocol.

Challenges for Malicious Security. The key non-triviality of achieving a secure PCI protocol arises in the setting of malicious corruption, where the generic solution

fails (even assuming a maliciously secure PSI protocol) since we can no longer enforce that parties execute Phase-1 honestly. We illustrate this in the simple setting of 2-party PCI. Suppose that in Phase-1 of the generic solution, a malicious P_1 chooses to include in its filtered subset an identity id under which: (i) P_1 does not have a single valid certificate, but (ii) P_2 has one or more valid certificates. Then, the output of Phase-2 allows P_1 to learn more information about the input set of P_2 than is allowed by the ideal functionality $\mathcal{F}_{\mathsf{PCI}}$. Clearly, this is true even if the underlying PSI protocol were maliciously secure.

"Tying" Validation to PSI. In order to enforce malicious security, we need to ensure that for each id in the final result set, each party P_i *proves* to all of the other parties that its input set contains a valid signature under id on some/all of its public claims. This is seemingly hard to achieve efficiently while using certificate validation and the PSI protocol as individual black-boxes since, prior to executing the PSI protocol, the parties do not know the set of identities for which such a proof is required. The parties could choose to provide proofs for all of their inputs, but this leaks more information about their input sets than allowed by $\mathcal{F}_{\mathsf{PCI}}$. To solve this issue, we require a mechanism that somehow "ties" certificate validation to the subsequent PSI protocol, rather than treating these as individual phases.

Maliciously Secure PCI. To upgrade our generic solution for the semi-honest setting outlined in Algorithm 1 to a malicious security setting, we use the following natural approach - run both phases of Algorithm 1 inside a maliciously secure MPC protocol [Yao82, DPSZ12]. In particular, Phase-1, which involves validation of claims and creation of filtered identity sets for each party, now happens inside the MPC protocol, and is tied to Phase-2 where the intersection of the identities from the parties is computed[3].

Non-Black-Box Claim Validation. Our generic maliciously-secure MPC protocol is theoretically feasible, but is highly inefficient in practice for almost all widely used cryptographically secure certification mechanisms. For example, the verification algorithms for popular digital signature schemes such as ECDSA [ANS99, JMV01] and BLS [BLS01, BDN18, BGLS03] rely on elliptic curve-based finite-field arithmetic operations. Implementing such a verification algorithm inside a maliciously secure MPC protocol would involve non-black-box usage of the various elliptic-curve operations, which is likely to incur huge computational and communication overheads in practice. The above discussion motivates specialized PCI protocols that efficiently yet securely realize $\mathcal{F}_{\mathsf{PCI}}$ against malicious corruptions while minimizing non-black-box usage of the underlying certificate validation algorithm. In this paper, we design and implement two concrete PCI protocols - a PCI-Any-DC protocol based on the ECDSA signature scheme and a PCI-All protocol based on the BLS signature scheme - that achieve the above goal.

[3]Note that Phase-2 can be implemented a simple intersection functionality without the use of a private version in PSI since it's already run inside an MPC.

3 MPC for Elliptic Curve Pairings

As a fundamental building block of our proposed PCI protocols, we introduce a new secret-sharing based MPC framework that is tuned for elliptic curve pairings. In this section, we describe this framework.

On Using Secret-Sharing based MPC. We begin by observing that both the ECDSA and BLS signature schemes fundamentally rely on elliptic curve (EC)-based finite-field arithmetic operations over F_p. Informally speaking, both standard EC operations (i.e., point addition and scalar multiplication over the EC group) and pairing based operations (i.e., algebraic operations over the output group of an EC pairing) share a common algebraic structure with the underlying field F_p (up to group homomorphisms). It turns out that secret-sharing based MPC protocols offer us precisely the desired amount of flexibility to manoeuvre over the algebraic structure of these groups without having to use the group representation/operations in a non-black-box manner. In particular, our overall approach (at a high level) is to design a secret-sharing based MPC protocol that supports EC operations and pairing operations as fundamental building blocks (similar in flavor to addition/multiplication "gates" in standard secret-sharing based MPC over F_p). This enables us to directly use black-box implementations of such operations without having to express them explicitly in terms of the underlying F_p operations.

On Choosing SPDZ. As a concrete instance of secret-sharing based MPC, we use the SPDZ secret sharing based protocol [DPSZ12] with malicious security against a dishonest majority of adversarial parties. The SPDZ protocol has been widely studied with several extensions [DPSZ12, DKL+13, CDE+18, DOK+20], optimizations [KOS16, KPR18], and robust open-source implementations available [ACC+, Kel20]. In addition, SPDZ naturally supports finite-field arithmetic operations over F_p, which also suits our requirements and overall approach, as outlined above.

Black-Box Usage of Standard Elliptic Curve Libraries. One of our main contributions is augmenting the SPDZ framework as well as the SPDZ open-source implementation to seamlessly support basic elliptic curve operations as well as elliptic curve pairing operations as fundamental gate-level building-blocks. This allows us to directly use standardized and open-source implementations of elliptic curve libraries [ope, Lyn, AGM+]. This is crucial from the point of view of both practical performance and real-world security, since we can immediately leverage both the performance improvements/optimizations as well as the protections against evolving implementation-level attacks that such libraries usually offer. To the best of our knowledge, such a framework was not available before, and this is an independent contribution since it enables an easy implementation of EC pairing-based MPC protocols.

We detail our proposed extension to SPDZ to support both basic EC and EC pairing operations in this section. We then describe our corresponding implementation and integration with the existing SPDZ open-source implementation in Section 6.

Our Framework for MPC over EC Pairings. We now detail our framework for designing secret-sharing based MPC protocols over EC pairings.Our framework can be

broadly divided into three-tiers, where each tier builds upon the preceding one.

- **Tier-1:** This tier of our framework supports the basic operations over F_p for some prime p.
- **Tier-2:** This tier of our framework supports group operations over any generic group $\mathcal{G}$ with order p. We use this tier to implement basic EC operations over the source groups of an EC pairing (i.e., point addition and scalar multiplication), as well as the group operations over the output group of the EC pairing (i.e., multiplication and exponentiation).
- **Tier-3:** This tier of our framework supports EC pairing operations, subject to the restriction that the pairing map e takes its inputs from two source groups $\mathcal{G}_1$ and $\mathcal{G}_2$, both of which have order p, and produces an output in a target group $\mathcal{G}_T$, also of order p.

While each tier supports a different set of operations, we exploit the fact that each tier shares a common algebraic structure (up to group homomorphisms), and we can manoeuvre over this structure to progressively support more complicated operations. We now describe each of these tiers in greater details below.

3.1 Tier-1: MPC for Basic F_p Operations

Our starting point is a secret-sharing based MPC engine for operating over secret-shared inputs in some field F_p that implements the ideal functionality $\mathcal{F}[F_P]$ as described in Figure 5. This engine can be realized directly using SPDZ (the SPDZ-based realization ensures security against both semi-honest and malicious corruption of parties by using an additional authentication mechanism to enforce honesty of operations over secret-shared values). We use the representation $[x]$ for any $x \in F_p$ to denote that the value x is secret-shared, i.e., no individual party has access to x, but each party has access to some share of x (for simplicity, we will assume that this notation incorporates the additional authentication components required to ensure malicious security).

Linearity-Preservation. Fundamentally, we require that the secret-shared representation $[x]$ is "linearity-preserving", i.e., for any $x, y, z, \alpha, \beta \in F_p$ such that $u = \alpha \cdot x + \beta \cdot y + z$, given the secret shares $[x]$ and $[y]$ and the public values z, α, β, the parties can compute a secret-sharing of u "for free" as

$$[u] = \alpha \cdot [x] + \beta \cdot [y] + z$$

Note that, in the case of malicious security, we also need this property to be preserved for the authentication components.

Additional Functionalities. We additionally require two deterministic functionalities to be supported by the MPC engine:

1. A functionality that "opens" a secret shared value $[x]$, i.e., reconstructs and distributes the value x to all or a subset of the parties.

$\underline{\mathcal{F}[F_P]}$

Init-F: On input (init, F_p) from all parties, the functionality stores $(domain, F_p)$. A list of identifiers is established for F_p, if not already done before.
Input-F: On input $(\mathsf{inp}F, P_i, \mathsf{varid}, x)$ with $x \in F_p$ from P_i and $(\mathsf{inp}F, P_i, \mathsf{varid}, \phi_{F_p})$ from all other parties, with varid a fresh identifier, the functionality stores (varid, x) in the list of field identifiers.
Rand-F: On input $(\mathsf{rand}, \mathsf{varid})$ from all parties (if varid is not stored in memory), the functionality generates a uniformly random $a \in F_p$ and stores (varid, a) in the list of field identifiers.
Triple-F: On input $(\mathsf{triple}, \mathsf{varid}_1, \mathsf{varid}_2, \mathsf{varid}_3)$ from all parties (if none of the varid_i are stored in memory), the functionality generates a uniformly random $a, b \in F_p$ and computes $c = a \cdot b$ and then stores (varid_1, a), (varid_2, b) and (varid_3, c) in the list of field identifiers.
Add-F: On command $(\mathsf{add}F, \mathsf{varid}_1, \mathsf{varid}_2, \mathsf{varid}_3)$ from all parties where $\mathsf{varid}1$, varid_2 are in the list of field identifiers and varid_3 is not, the functionality retrieves (varid_1, x), (varid_2, y) from the list of field identifiers and stores $(\mathsf{varid}_3, x + y)$ in the list of field identifiers.
Mult-F: On command $(\mathsf{mult}F, \mathsf{varid}_1, \mathsf{varid}_2, \mathsf{varid}_3)$ from all parties where varid_1, varid_2 are in the list of field identifiers and varid_3 is not, the functionality retrieves (varid_1, x), (varid_2, y) from the list of field identifiers and stores $(\mathsf{varid}_3, x \cdot y)$ in the list of field identifiers.
Output-F: On input $(\mathsf{out}F, \mathsf{varid}, i)$ from all honest parties (if varid is present in the list of field identifiers), the functionality retrieves (varid, y) from the set of field identifiers and outputs it to the environment. The functionality waits for an input from the environment. If this input is $\mathsf{Deliver}$ then y is output to all parties if $i = 0$, or y is output to party P_i if $i \neq 0$. If the adversarial input is not equal to $\mathsf{Deliver}$ then ϕ is output to all parties.

Figure 5: Ideal functionality for MPC over field operations in F_p

2. A functionality that "multiplies" secret shared inputs, i.e., given two secret-shared inputs $[x]$ and $[y]$, produces a secret-shared output $[z]$ such that $z = x \cdot y$.

Finally, we require two randomized functionalities to be supported by the MPC engine:

1. A functionality that generates a secret-shared representation $[a]$ for a randomly sampled value $a \leftarrow F_p$.
2. A functionality that generates secret-shared representations of uniformly random multiplicative "triples", i.e., it generates $[a]$, $[b]$ and $[c]$ for $a, b \leftarrow F_p$ and $c = a \cdot b$.

We refer to $\mathcal{F}[F_P]$ described in Figure 5 for a formal description of these functionalities. Note that, for malicious security, we would need each of the above functionalities to also preserve (or, in the case of opening, validate) the authentication components of the output appropriately.

SPDZ-based Realization. While we can use any secret-sharing-based MPC engine that securely realizes $\mathcal{F}[F_P]$, we choose to use SPDZ as a concrete realization, with secu-

rity against a malicious corruption of the majority of the parties. We briefly recall here that, in addition to securely implementing $\mathcal{F}[F_P]$, SPDZ also implements a MAC-check based authentication mechanism for secret-shared values $[x]$ to achieve active security against malicious corruption of parties. We recall the details of this mechanism at a very high level; the low-level details are not important for understanding our proposed framework. Informally, in SPDZ, each party P_i for $i \in [1, n]$ holds a sharing of a global MAC-key $\alpha \in F_p$ (this sharing follows a slightly different mechanism; we omit the details as our framework is oblivious to the same). Any value $x \in F_p$ is shared as

$$[x] = (\delta, (x_1, \ldots, x_n), (\gamma_1(x), \ldots, \gamma_n(x))),$$

where for each $i \in [n]$, party P_i holds the tuple $(x_i, \gamma_i(x), \delta)$ and where the following invariant holds:

$$x = \sum_{i \in [n]} x_i, \quad \alpha \cdot (x + \delta) = \sum_{i \in [n]} \gamma_i(x).$$

The SPDZ *Opening Protocol.* we briefly recall how the "opening" protocol in SPDZ allows the parties to authenticate, via a MAC-check mechanism, that a secret-shared value has been opened correctly. The opening protocol for a secret-shared value $[x]$ involves the following steps:

- Each party P_i, upon receiving a reconstructed value x', uses its share α_i of the global MAC-key α, as well as $\gamma_i(x)$ and δ, to compute $\sigma_i = \gamma_i(x) - \alpha_i \cdot (x' + \delta)$.
- Each party P_i then broadcasts a commitment $\mathsf{Com}(\sigma_i)$ to all the other parties.
- Finally, each party P_i opens the commitments $\{\mathsf{Com}(\sigma_j)\}$ received from $\{P_j\}_{j \neq i}$, computes $\mathsf{chk} = \sum_{j \in [n]} \sigma_j$, and aborts if $\mathsf{chk} \neq 0$.

We use the term *partial opening* to refer the procedure that just publicly reconstructs the value x without going through the subsequent MAC-check procedure.

Suppose that a malicious adversary $\mathcal{A}$ manages to add an error ϵ during the reconstruction phase, i.e., we have $x' = x + \epsilon$. Suppose also that the adversary $\mathcal{A}$ commits to a subset of false $\{\sigma'_j\}_{j \in \mathcal{C}}$ values corresponding to the subset $\mathcal{C} \subset [n]$ of parties it corrupts. In order to bypass the MAC-check, the adversary $\mathcal{A}$ must ensure that

$$\sum_{j \in \mathcal{C}} (\sigma'_j - \sigma_j) = \alpha\epsilon.$$

However, this happens with probability no greater than $1/p$, since the global MAC value α is uniformly random in F_p and (information-theoretically) unknown to $\mathcal{A}$, and hence, $\mathcal{A}$ cannot bypass the MAC-check protocol except with negligible probability.

Additional Functionalities in SPDZ. We note that the randomized functionalities for generating secret-shared representations of singleton values or multiplicative triples are implemented by the offline phase of SPDZ [KOS16]. We omit the low-level details of these functionalities because they are not necessary to understand our framework and proposed protocols; it suffices to state that our framework uses the native implementations of these functionalities directly from SPDZ. We also directly use SPDZ's implementation of the functionality for multiplying secret-shared values, which is based on generating a random multiplicative triple and then using Beaver's re-randomization technique. We refer to [DPSZ12, DKL+13] for the details.

$\underline{\mathcal{F}[\mathcal{G}]}$

Init-G: On input $(\mathsf{init}, \mathcal{G})$ from all parties, the functionality stores $(domain, \mathcal{G})$. A list of identifiers is established for $\mathcal{G}$, if not already done before.
Input-G: On input $(\mathsf{inp}\mathcal{G}, P_i, \mathsf{varid}, g)$ with $g \in \mathcal{G}$ from P_i and $(\mathsf{inp}\mathcal{G}, P_i, \mathsf{varid}, \phi_{\mathcal{G}})$ from all other parties, with varid a fresh identifier, the functionality stores (varid, g) in the list of field identifiers.
Op-G: On command $(\mathsf{op}\mathcal{G}, \mathsf{varid}_1, \mathsf{varid}_2, \mathsf{varid}_3)$ from all parties where $\mathsf{varid1}$, varid_2 are in the list of group identifiers and varid_3 is not, the functionality retrieves (varid_1, g), (varid_2, h) from the list of group identifiers and stores $(\mathsf{varid}_3, g \cdot h)$ in the list of group identifiers, where $\cdot$ is the group operation.
Exp-G-P: On command $(\exp \mathcal{G}P, \mathsf{varid}_1, g, \mathsf{varid}_2)$ from all parties where varid_1 is in the list of field identifiers, $g \in \mathcal{G}$, and varid_2 is a fresh identifier in the list of group identifiers, the functionality retrieves (varid_1, x) from the list of field identifiers and stores (varid_2, g^x).
Exp-G-S: On command $(\exp \mathcal{G}S, \mathsf{varid}_1, \mathsf{varid}_2, \mathsf{varid}_3)$ from all parties where varid_1 is in the list of field identifiers, varid_2 is in the list of group identifiers, and varid_3 is a fresh identifier in the list of group identifiers, the functionality retrieves (varid_1, x) from the list of field identifiers and (varid_2, h) from the list of group identifiers and stores (varid_2, h^x).
Output-G: On input $(\mathsf{out}\mathcal{G}, \mathsf{varid}, i)$ from all honest parties (if varid is present in the list of group identifiers), the functionality retrieves (varid, g) from the set of group identifiers and outputs it to the environment. The functionality waits for an input from the environment. If this input is $\mathsf{Deliver}$ then g is output to all parties if $i = 0$, or g is output to party P_i if $i \neq 0$. If the adversarial input is not equal to $\mathsf{Deliver}$ then ϕ is output to all parties.

Figure 6: Ideal functionality for MPC over the group operations in $\mathcal{G}$, which includes basic EC operations and the operations over the output group of a pairing. We assume that $\mathcal{F}[\mathcal{G}]$ also includes all Tier-1 sub-functionalities in $\mathcal{F}[F_p]$, but we avoid re-writing them for modularity.

3.2 Tier-2: MPC over any Generic Group

In Tier-2, we aim to realize an MPC protocol over any generic group $\mathcal{G}$ with prime order p. More concretely, we require the MPC protocol to implement the ideal functionality $\mathcal{F}[\mathcal{G}]$ as described in Figure 6. Such a protocol would allow us to support basic EC operations (i.e., point addition and scalar multiplication) over the source groups of an EC pairing, as well as the operations over the target group of the EC pairing (i.e., group multiplication and exponentiation). Similar to Tier-1, we use a linearity-preserving representation $[\cdot]_{\mathcal{G}}$ for elements in $\mathcal{G}$ such that for any $g_1, g_2, g_3 \in \mathcal{G}$ and any $\alpha, \beta \in Z_p$ such that $h = g_1^{\alpha} \cdot g_2^{\beta} \cdot g_3$, given the secret shares $[g_1]_{\mathcal{G}}$ and $[g_2]_{\mathcal{G}}$ and the public values g_3, α, β, the parties can locally compute

$$[h]_{\mathcal{G}} = {[g_1]_{\mathcal{G}}}^{\alpha} \cdot {[g_2]_{\mathcal{G}}}^{\beta} \cdot g_3$$

Once again, in the case of malicious security, we need this property to be preserved for the authentication components.

Homomorphic Relation with Tier-1. We note that the aforementioned linearity-preservation property in $\mathcal{G}$ shares a similar algebraic structure with the tier-1 linearity-preservation property in F_p described earlier. Let $F_p = Z_p$, and let

$$g_1 = g^x, \quad g_2 = g^y, \quad g_3 = g^z, \quad h = g^u.$$

Then observe that the linearity-preservation property in Z_p with u

Additional Functionalities. We additionally require three deterministic functionalities to be supported by the MPC engine:

1. A functionality that "opens" a secret shared value $[g]_{\mathcal{G}}$, i.e., reconstructs and distributes the group element g to all or a subset of the parties.
2. A functionality that "exponentiates" a publicly available group element in $\mathcal{G}$ using a secret-shared value in Z_p, i.e., given a public $g \in \mathcal{G}$ and a secret-shared value $[x]$ for $x \in Z_p$, produces a secret-shared output $[h]_{\mathcal{G}}$ such that $h = g^x$.
3. A functionality that "exponentiates" a secret-shared group element in $\mathcal{G}$ using a secret-shared value in Z_p, i.e., given a secret-shared element $[g]_{\mathcal{G}}$ for $g \in \mathcal{G}$ and a secret-shared value $[x]$ for $x \in Z_p$, produces a secret-shared output $[h]_{\mathcal{G}}$ such that $h = g^x$.

We refer to $\mathcal{F}[\mathcal{G}]$ described in Figure 6 for a formal description of these functionalities. Once again, for malicious security, we would need each of the above functionalities to preserve (or, in the case of opening, validate) the authentication components of the output appropriately.

Tier-2 Extension of SPDZ. As a concrete instantiation of $\mathcal{F}[\mathcal{G}]$, we generalize the extensions to SPDZ for basic EC operations proposed in [STA19, DOK+20] to any generic group of order p. We briefly recall the details of the approach, albeit in its generalized form. At a high level, we exploit the homomorphic relationship between the additive group over Z_p and the group $\mathcal{G}$, which yields a natural way to map the linearity-preserving property of SPDZ over Z_p to its extension over $\mathcal{G}$. Informally speaking, for $h = g^x$ for some publicly available generator g of $\mathcal{G}$, let $[h]_{\mathcal{G}} := g^{[x]}$. Then, observe that the linearity-preservation property in $\mathcal{G}$ follows from the linearity-preservation property in Z_p, albeit implicitly in the exponent of the public group element g.

Concretely, any group element $g \in \mathcal{G}$ is shared as

$$[g]_{\mathcal{G}} = (\delta_{\mathcal{G}}, (g_1, \ldots, g_n), (\gamma_1(g), \ldots, \gamma_n(g))),$$

where for each $i \in [n]$, party P_i holds the tuple $(g_i, \gamma_i(x), \delta_{\mathcal{G}}) \in \mathcal{G} \times \mathcal{G} \times \mathcal{G}$, and where the following invariant holds:

$$g = \prod_{i \in [n]} g_i, \quad (g \cdot \delta_{\mathcal{G}})^{\alpha} = \prod_{i \in [n]} \gamma_i(g),$$

where α is the same global MAC-key as used in Tier-1.

Opening and MAC-Check in $\mathcal{G}$. The opening protocol for a secret-shared group element $[g]_{\mathcal{G}}$ is also analogous to the corresponding protocol for F_p where each party P_i does the following: (a) upon receiving a reconstructed value x', computes $\sigma_i = \gamma_i(g)/(g' \cdot \delta_{\mathcal{G}})^{\alpha_i}$, (b) broadcasts a commitment $\mathsf{Com}(\sigma_i)$ to all the other parties, and (c) opens the commitments $\{\mathsf{Com}(\sigma_j)\}$ received from $\{P_j\}_{j \neq i}$, computes $\mathsf{chk} = \prod_{j \in [n]} \sigma_j$, and aborts if $\mathsf{chk} \neq \mathrm{id}_{\mathcal{G}}$, where $\mathrm{id}_{\mathcal{G}}$ is the additive identity for the group $\mathcal{G}$. We can use a very similar argument as that in Tier-1 to prove that an adversary $\mathcal{A}$ cannot bypass this extended MAC-check protocol over $\mathcal{G}$, except with negligible probability.

Exponentiating a Public Element in $\mathcal{G}$. As mentioned in prior works [STA19], exponentiating a publicly available group element in $\mathcal{G}$ using a secret-shared value in Z_p is immediate; given a public group element g and a secret-sharing of x of the form

$$[x] = (\delta, (x_1, \ldots, x_n), (\gamma_1(x), \ldots, \gamma_n(x))),$$

one can easily compute a secret-sharing of $h = g^x$ as

$$[h]_{\mathcal{G}} = g^{[x]} := \left(g^{\delta}, (g^{x_1}, \ldots, g^{x_n}), \left(g^{\gamma_1(x)}, \ldots, g^{\gamma_n(x)}\right)\right).$$

Exponentiating a Secret-Shared Element in $\mathcal{G}$. In order to exponentiate a secret-shared group element $[g]_{\mathcal{G}}$ using a secret-shared value $[x]$, the parties use a protocol that naturally extends SPDZ's implementation of the functionality for multiplying secret-shared values (based on generating a random multiplicative triple and then using Beaver's re-randomization technique). Concretely, the parties follows the following steps:

- Generate $[a]$, $[b]$ and $[c]$ for $a, b \leftarrow Z_p$ and $c = a \cdot b$ using the triple-generation functionality in Tier-1
- Locally compute $[h_1]_{\mathcal{G}} = g^{[b]}$ and $[h_2]_{\mathcal{G}} = g^{[c]}$ using the exponentiation algorithm outlined above.
- Partially open the values $\epsilon = (x - a)$ and $h_3 = g/h_1$.
- Locally compute $[h_4]_{\mathcal{G}} = h_3^{[a]}$ (using the exponentiation algorithm outlined above) and $h_5 = h_3^{\epsilon}$.
- Locally compute $[h]_{\mathcal{G}} = [h_2]_{\mathcal{G}} \cdot \left([h_1]_{\mathcal{G}}\right)^{\epsilon} \cdot [h_4]_{\mathcal{G}} \cdot h_5$.

Note that the final local computation is allowed by the linearity-preserving property of the secret-sharing over $\mathcal{G}$; we omit the explicit details for simplicity.

Remark. We note here that while the aforementioned extension of SPDZ was proposed theoretically in prior works [STA19,DOK+20], it was done specifically for EC groups (in particular, [STA19] is the only prior work to propose protocols for scalar-multiplying public/secret-shared EC points with secret-shared scalars, and their treatment is entirely specific to plain EC groups). As already mentioned, our generalized approach allows us to make this engine usable for pairing-friendly EC curves, since we can instantiate this engine not only for the source groups of an EC pairing (which are both elliptic curve groups), but also for the target group of the EC pairing, which is not an EC group but a multiplicative group over some extension field of F_p. As it turns out, this is an important building block that eventually allows us to support EC pairing operations in Tier-3 of our framework.

$\underline{\mathcal{F}[\mathsf{Pair}]}$

Pair-G1-P: On command $(\mathsf{pair}\mathcal{G}P, g_1, \mathsf{varid}_1, \mathsf{varid}_2)$ from all parties where $g_1 \in \mathcal{G}_1$, varid_1 is in the list of group $\mathcal{G}_2$ identifiers, and varid_2 is a fresh identifier in the list of group $\mathcal{G}_T$ identifiers, the functionality retrieves (varid_1, g_2) from the list of $\mathcal{G}_2$ identifiers and stores $(\mathsf{varid}_2, e(g_1, g_2))$, where e is the pairing function.
Pair-G2-P: On command $(\mathsf{pair}\mathcal{G}P, \mathsf{varid}_1, g_2, \mathsf{varid}_2)$ from all parties where varid_1 is in the list of group $\mathcal{G}_1$ identifiers, $g_2 \in \mathcal{G}_2$, and varid_2 is a fresh identifier in the list of group $\mathcal{G}_T$ identifiers, the functionality retrieves (varid_1, g_1) from the list of $\mathcal{G}_1$ identifiers and stores $(\mathsf{varid}_2, e(g_1, g_2))$, where e is the pairing function.
Pair-S: On command $(\mathsf{pair}S, \mathsf{varid}_1, \mathsf{varid}_2, \mathsf{varid}_3)$ from all parties where varid_1 is in the list of group $\mathcal{G}_1$ identifiers, varid_2 is in the list of group $\mathcal{G}_2$ identifiers, and varid_3 is a fresh identifier in the list of group $\mathcal{G}_T$ identifiers, the functionality retrieves (varid_1, g_1) from the list of $\mathcal{G}_1$ identifiers, (varid_2, g_2) from the list of $\mathcal{G}_2$ identifiers and stores $(\mathsf{varid}_3, e(g_1, g_2))$.

Figure 7: Ideal functionality for MPC over the EC pairing operation with $\mathcal{G}_1$ and $\mathcal{G}_2$ as the input groups and $\mathcal{G}_T$ as the target group. We assume that $\mathcal{F}[\mathsf{Pair}]$ also includes all Tier-1 and Tier-2 sub-functionalities in $\mathcal{F}[F_p]$ and $\mathcal{F}[\mathcal{G}]$, but we avoid re-writing them for modularity.

3.3 Tier-3: MPC over EC Pairings

We now build upon the infrastructure set up in Tier-1 and Tier-2 and design the MPC engine to support EC pairing operations. In particular, for a bilinear pairing $e : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{G}_T$, we start with Tier-2 instances for each of the groups $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_T$ (all of which satisfy linearity-preserving and support the operations outlined earlier), and realize the following three deterministic functionalities for EC pairings:

1. An EC pairing functionality that pairs a publicly available group element in $\mathcal{G}_1$ with a secret-shared group element in $\mathcal{G}_2$, i.e., given a public $g_1 \in \mathcal{G}_1$ and a secret-shared group element $[g_2]_{\mathcal{G}_2}$ for $g_2 \in \mathcal{G}_2$, outputs a secret-shared output $[g_T]_{\mathcal{G}_T}$ such that $g_T = e(g_1, g_2)$.

2. An EC pairing functionality that pairs a secret-shared group element in $\mathcal{G}_1$ with a publicly available group element in $\mathcal{G}_2$, i.e., given a secret-shared group element $[g_1]_{\mathcal{G}_1}$ for $g_1 \in \mathcal{G}_1$ and a public $g_2 \in \mathcal{G}_2$, produces a secret-shared output $[g_T]_{\mathcal{G}_T}$ such that $g_T = e(g_1, g_2)$.

3. An EC pairing functionality that pairs a secret-shared group element in $\mathcal{G}_1$ with a secret-shared group element in $\mathcal{G}_2$, i.e., given a secret-shared element $[g_1]_{\mathcal{G}_1}$ for $g_1 \in \mathcal{G}_1$ and a secret-shared group element $[g_2]_{\mathcal{G}_2}$ for $g_2 \in \mathcal{G}_2$, outputs a secret-shared output $[g_T]_{\mathcal{G}_T}$ such that $g_T = e(g_1, g_2)$.

We refer to $\mathcal{F}[\mathsf{Pair}]$ described in Figure 7 for a formal description of these functionalities. Once again, for malicious security, we would need each of the above functionalities to preserve the authentication components of the output appropriately. We note here that this functionality supports both symmetric and asymmetric pairings (in the symmetric case, we simply instantiate the framework with $\mathcal{G}_1 = \mathcal{G}_2 = \mathcal{G}$).

Tier-3 Extension of SPDZ. One of our technical contributions is an extension of the SPDZ framework to support MPC protocols realizing $\mathcal{F}[\mathsf{Pair}]$, which we describe here.

Pairing with One Secret-Shared Input. We begin by describing how to compute an EC pairing when one of the input group elements is secret-shared and the other input group element is public. We realize this by exploiting the bilinear property of the EC pairing. Recall that if $e : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{G}_T$ is a bilinear pairing, then for any $g_1, h_1 \in \mathcal{G}_1$ and any $g_2, h_2 \in \mathcal{G}_2$, we have

$$e(g_1 \cdot h_1, g_2) = e(g_1, g_2) \cdot e(h_1, g_2),$$
$$e(g_1, g_2 \cdot h_2) = e(g_1, g_2) \cdot e(g_1, h_2).$$

Now, observe that to pair a publicly available group element in $\mathcal{G}_1$ with a secret-shared group element in $\mathcal{G}_2$, each party can just locally compute

$$[h_T]_{\mathcal{G}_T} = e\left(h_1, [h_2]_{\mathcal{G}_2}\right),$$

and this yields a valid secret-sharing of pairing output h_T because of: (a) the bilinearity property of e as described above, and (b) the linearity-preservation property of the secret-sharing mechanism over $\mathcal{G}_2$. Pairing a publicly available group element in $\mathcal{G}_2$ with a secret-shared group element in $\mathcal{G}_1$ is analogously straightforward, wherein each party locally computes

$$[h_T]_{\mathcal{G}_T} = e\left([h_1]_{\mathcal{G}_1}, h_2\right).$$

Pairing with Two Secret-Shared Inputs. We now propose a protocol that allows the parties to pair a secret-shared group element $[h_1]_{\mathcal{G}_1}$ with a secret-shared group element $[h_2]_{\mathcal{G}_2}$, the parties follows the following steps. The protocol is inspired by SPDZ's implementation of the functionality for multiplying secret-shared values (based on generating a random multiplicative triple and then using Beaver's re-randomization technique), but needs to be carefully adapted to the setting of EC pairings. Concretely, in our proposed protocol, the parties proceed as follows:

- Generate $[a]$, $[b]$ and $[c]$ for $a, b \leftarrow Z_p$ and $c = a \cdot b$ using the triple-generation functionality in Tier-1.
- Locally compute

  $$[u_1]_{\mathcal{G}_1} = g_1^{[a]}, \quad [u_2]_{\mathcal{G}_2} = g_2^{[b]}, \quad [u_3]_{\mathcal{G}_1} = g_1^{[c]},$$

  using the exponentiation algorithm for public group elements in the Tier-2 MPC engine for $\mathcal{G}_1$ and $\mathcal{G}_2$.
- Partially open the values $h_3 = h_1/u_1 \in \mathcal{G}_1$ and $h_4 = h_2/u_2 \in \mathcal{G}_2$.
- Locally compute

  $$[v_1]_{\mathcal{G}_T} = e\left([u_3]_{\mathcal{G}_1}, g_2\right), \qquad [v_2]_{\mathcal{G}_T} = e\left(h_3, [u_2]_{\mathcal{G}_2}\right)$$
  $$[v_3]_{\mathcal{G}_T} = e\left([u_1]_{\mathcal{G}_1}, h_4\right), \qquad v_4 = e(h_3, h_4).$$

- Locally compute $[h_T]_{\mathcal{G}_T} = [v_1]_{\mathcal{G}_T} \cdot [v_2]_{\mathcal{G}_T} \cdot [v_3]_{\mathcal{G}_T} \cdot v_4$.

Note that the final local computation is allowed by the linearity-preserving property of the secret-sharing over $\mathcal{G}_T$; we omit the explicit details for simplicity. To prove correctness, it suffices to prove that $h_T = v_1 \cdot v_2 \cdot v_3 \cdot v_4$; correctness of the sharing again follows immediately from: (a) the bilinearity property of e described above, and (b) the linearity-preservation property of the secret-sharing mechanism over $\mathcal{G}_1$ and $\mathcal{G}_2$. Observe that

$$\begin{aligned}
&v_1 \cdot v_2 \cdot v_3 \cdot v_4 \\
&\quad = e\left(u_3, g_2\right) \cdot e\left(h_3, u_2\right) \cdot e\left(u_1, h_4\right) \cdot e(h_3, h_4) \\
&\quad = e\left(g_1^c, g_2\right) \cdot e\left(h_1 \cdot g_1^{-a}, g_2^b\right) \cdot e\left(g_1^a, h_2 \cdot g_2^{-b}\right) \cdot e\left(h_1 \cdot g_1^{-a}, h_2 \cdot g_2^{-b}\right) \\
&\quad = e\left(g_1, g_2\right)^c \cdot e\left(h_1, g_2\right)^b \cdot e\left(g_1, g_2\right)^{-ab} \cdot e\left(g_1, g_2\right)^{-ab} \cdot e\left(g_1, h_2\right)^a \cdot e\left(h_1, h_2\right) \\
&\qquad\qquad\qquad\qquad \cdot e\left(h_1, g_2\right)^{-b} \cdot e\left(g_1, h_2\right)^{-a} \cdot e\left(g_1, g_2\right)^{ab} \\
&\quad = e(h_1, h_2) = h_T
\end{aligned}$$

We highlight here that our solution uses the group operations and the pairing operations of the pairing-friendly EC group as a black-box. This enables us to use the state-of-the-art libraries such as RELIC [AGM+] for implementing the pairing operations on top of the MP-SPDZ framework. To the best of our knowledge, this is first proposal and implementation of an MPC protocol that efficiently supports EC pairings, and is likely to have applications beyond PCI.

4 PCI-Any-DC using ECDSA signature scheme

In this section, we describe a concrete instantiation of two-party PCI-Any-DC using the ECDSA signature scheme. We subsequently discuss how to extend this scheme to support PCI-Any and PCI-All.

Notations. Let the elliptic curve group $\mathcal{G}$ of prime order p be defined over a field F_p as a set of points $(x, y) \in F_p \times F_p$. Though the EC group $\mathcal{G}$ is an additive group of points over the elliptic curve, we will continue to use the multiplicative notation to ensure uniformity throughout the paper. Hence, we will denote point addition between two points Q_1 and Q_2 as $Q_1 \cdot Q_2$, and the scalar multiplication between a point Q and $x \in Z_p$ as Q^x. Let $Q \in \mathcal{G}$ be the generator of the group $\mathcal{G}$ (base point in standard EC parlance), and therefore we have $Q^p = \mathcal{O}$, where $\mathcal{O}$ is the point at infinity (the identity element). For any $Q' \in \mathcal{G}$, we use $[Q']_{\mathcal{G}}$ to denote the linearity preserving secret-sharing of Q'.

The ECDSA Signature Scheme. We briefly recall the key generation, signing, and verification equations for ECDSA.

KeyGen(λ): On input a security parameter λ, the key generation algorithm samples a private signing key $x \leftarrow [1, p-1]$, and computes the public verification key $Y := Q^x$. The algorithm outputs the pair (x, Y).

Algorithm 2: PCl-Any-DC **using ECDSA**

Private inputs from P_1: $\mathsf{inp}_{1,1} = [(Y_{1,\ell}, s_{1,\ell}^{-1}, \mathsf{m}_{1,\ell})]_{\ell \in [1,N_1]}$

Each $Y_{1,\ell}$ is shared as $[Y_{1,\ell}]_{\mathcal{G}_2}$ using Input-G, and each $s_{1,\ell}^{-1}$ is shared as $\left[s_{1,\ell}^{-1}\right]$ using Input-F.

Public inputs from P_1: $\mathsf{inp}_{1,2} = [(r_{1,\ell}, R_{1,\ell}, \mathsf{m}_{1,\ell})]_{\ell \in [1,N_1]}$

Private inputs from P_2: $\mathsf{inp}_{2,1} = [(Y_{2,\ell}, s_{2,\ell}^{-1}, \mathsf{m}_{2,\ell})]_{\ell \in [1,N_2]}$

Each $Y_{2,\ell}$ is shared as $[Y_{2,\ell}]_{\mathcal{G}_2}$ using Input-G, and each $s_{2,\ell}^{-1}$ is shared as $\left[s_{2,\ell}^{-1}\right]$ using Input-F.

Public inputs from P_2: $\mathsf{inp}_{2,2} = [(r_{2,\ell}, R_{2,\ell}, \mathsf{m}_{2,\ell})]_{\ell \in [1,N_2]}$

P_1 validates each $R_{2,\ell} \neq \mathcal{O}$ and has-x coordinate $r_{2,\ell}$.

P_2 validates each $R_{1,\ell} \neq \mathcal{O}$ and has x-coordinate $r_{1,\ell}$.

▷ **Validate P_1's input signatures**

for $\ell := 1 \ldots N_1$ **do**

  $[u_{1,\ell}] := H(\mathsf{m}_{1,\ell}) \cdot \left[s_{1,\ell}^{-1}\right]$

  $[v_{1,\ell}] := r_{1,\ell} \cdot \left[s_{1,\ell}^{-1}\right]$

  $\left[C_\ell^1\right]_{\mathcal{G}} := \text{Exp-G-P}([u_{1,\ell}], Q) \cdot \text{Exp-G-S}([v_{1,\ell}], [Y_{1,\ell}]_{\mathcal{G}_2}) / R_{1,\ell}$

▷ **Validate P_2's input signatures**

for $\ell' := 1 \ldots N_2$ **do**

  $[u_{2,\ell'}] := H(\mathsf{m}_{2,\ell'}) \cdot \left[s_{2,\ell'}^{-1}\right]$

  $[v_{2,\ell'}] := r_{2,\ell'} \cdot \left[s_{2,\ell'}^{-1}\right]$

  $\left[C_{\ell'}^2\right]_{\mathcal{G}} := \text{Exp-G-P}([u_{2,\ell'}], Q) \cdot \text{Exp-G-S}([v_{2,\ell'}], [Y_{2,\ell'}]_{\mathcal{G}_2}) / R_{2,\ell'}$

▷ **Match certifier**

The parties agree on public random values $\mathsf{rnd}_1, \mathsf{rnd}_2 \leftarrow Z_p$.

for $\ell := 1 \ldots N_1$ **do**

  for $\ell' := 1 \ldots N_2$ **do**

    Generate secret-shared randomness $[\mathsf{rnd}_{\ell,\ell'}] \leftarrow$ Rand-F.

    $[C]_{\mathcal{G}} := [Y_{1,\ell}]_{\mathcal{G}} / [Y_{2,\ell'}]_{\mathcal{G}}$

    $[C']_{\mathcal{G}} := \left[C_\ell^1\right]_{\mathcal{G}} \cdot \left[C_{\ell'}^2\right]_{\mathcal{G}}^{\mathsf{rnd}_1} \cdot [C]_{\mathcal{G}}^{\mathsf{rnd}_2}$

    $\left[C''_{\ell,\ell'}\right]_{\mathcal{G}} := \text{Exp-G-S}([\mathsf{rnd}_{\ell,\ell'}], [C']_{\mathcal{G}})$

    Output-G$\left(\left[C''_{\ell,\ell'}\right]_{\mathcal{G}}\right)$

    If $C''_{\ell,\ell'} == \mathcal{O}$, then Output-G$([Y_{1,\ell}]_{\mathcal{G}})$

$\mathsf{Sign}(x, m)$**:** On input a signing key x and a message $m \in \{0,1\}^*$, the signing algorithm does the following: (i) samples a random $k \leftarrow [1, p-1]$, (ii) computes $R = (x, y) := Q^k$ (a random point on the curve), (iii) computes $r = x \mod p$ and $s = k^{-1}(H(m) + r \cdot x) \mod p$, where $\mathcal{H} : \{0,1\}^* \to [0, p-1]$ denotes a hash function, (iv) repeats (i)-(iii) until $r \neq 0$ and $s \neq 0$. The algorithm finally outputs the signature $\sigma = (r, s)$.

$\mathsf{Verify}(Y, \sigma, m)$**:** On input a verification key Y, a signature σ and a message m, the verification algorithm computes $u_1 = H(m) \cdot s^{-1} \mod p$, $u_2 = r \cdot s^{-1} \mod p$ and computes $R := (x', y') = Q^{u_1} \cdot Y^{u_2}$. The algorithm outputs 1 if $(x', y') \neq \mathcal{O}$ and $x' = r$, and outputs 0 otherwise.

Protocol Overview. The starting point of our protocol is the generic maliciously secure protocol outlined in the introduction where we have the certificate validation and creation of the filtered sets of identities followed by the intersection of the sets from the two parties. We note here that we could have a single certifier issue multiple certificates on multiple different claims, or multiple certificates some of the same claims. However, we prescribe the parties to select only one certificate from a single certifier on one claim, i.e., there is a single (certificate, claim) pair for each certifier input to the protocol. We also expect an honest party to only input valid certificates on its set of public claims (although this is not a strict requirement for our protocol).

Optimizing Verify*:* Our main effort here is to reduce or obviate the non-algebraic operations in the Verify algorithm. In addition to the additions and multiplications, Verify requires an inverse operation in F_p and the extraction of the x-coordinate of an EC point from the point description (which is a trivial task to do in the plaintext world but not so inside an MPC). To do this, we make two observations. First, we note that the unforgeability of the signature scheme is retained if s^{-1} is input instead of s; given a signature (r, s), it is trivial to compute (r, s^{-1}) and hence the unforgeability guarantees are equivalent for (r, s) and (r, s^{-1}). This way the inverse can be done outside the MPC and the parties can provide the corresponding s^{-1} as their secret inputs.

Second, in addition to r, we input the point $R = (r, y)$ by calculating the y-coordinate, and check that the signature verification procedure actually yields the point R (recall that the original ECDSA signature verification algorithm first reconstructs the point R and then extracts its x-coordinate r). If r and R were to be private inputs, the MPC algorithm would have to check that the r is the valid x-coordinate of R to prevent maliciously constructed inputs. We obviate this by making r and R public. Observe that, in the ECDSA signing algorithm, the point R is a uniformly random point in the group $\mathcal{G}$, thus R and its x-coordinate r are statistically independent of the corresponding public key. In other words, the public key is not revealed when r and R are provided, even if the universal set of public keys is available to the adversary. We also note that a malicious adversary cannot forge signatures by inputting an invalid point R' since, given the x-coordinate r and the public description of the elliptic curve group $\mathcal{G}$, one can efficiently compute the two possible EC points the form (r, y) in the group $\mathcal{G}$, and either of these would match the point R reconstructed by the verification algorithm if and only if the original signature (r, s) was valid. At this point, we can perform certificate verification inside MPC using the operations in Tier-2 of our proposed MPC engine.

Computing the Intersection: We now perform the intersection of the sets of public keys by subtracting the corresponding elliptic curve points (dividing in the multiplicative notation) and checking if it opens to the identity element (point at infinity). It is important to hide the difference value if it is not the identity; otherwise we leak information about the public keys which are not part of the output set, which is not an allowed leakage according to our definition. So, we randomize the difference before opening while retaining the identity value. Another optimization in our protocol is that we store the information on the validity of the certificates in $\left[C_l^1\right]_{\mathcal{G}}$s and $\left[C_{l'}^2\right]_{\mathcal{G}}$s and open them along with the variable $[C]_{\mathcal{G}}$ storing the equality of public keys, as a random linear combination of three variables corresponding to the validity of P_1's certificate, validity of P_2's certificate and the equality of the public keys of the certifiers. This opens to the identity element if and only if all of the three requirements are satisfied.

The detailed description of our PCI-Any-DC protocol for ECDSA is provided in Algorithm 2. Here, each party inputs tuples of (identifier, certificate, claim) with the above discussed modifications as its private input, and the corresponding claim and (r, R) for each tuple as its public input. Note that the validation of P_1's certificates and P_2's certificates will be executed in parallel by the MPC algorithm. We describe the protocol in the $\mathcal{F}[\mathcal{G}]$-hybrid model, i.e., we assume that each sub-functionality in $\mathcal{F}[\mathcal{G}]$ has a secure instantiation. This allows us to define and prove the protocols in a modular way. A concrete instance of the protocol would use the SPDZ-based instantiation described in Section 3 to perform ECDSA signature validations while using all operations over the EC group $\mathcal{G}$ in a black-box way.

4.1 Correctness and Security

Correctness of the protocol follows immediately. We state the following theorem for the security of the protocol:

Theorem 1. *Our proposed* PCI-Any-DC *protocol for ECDSA signatures as described in Algorithm 2 securely emulates* $\mathcal{F}_{\mathsf{PCI}}$(PCI-Any-DC) *(for the two-party setting).*

Proof. We prove Theorem 1 by constructing a PPT simulator $\mathcal{S}$ such that no PPT environment $\mathcal{Z}$, who corrupts one of the parties (say P_1 without loss of generality) and chooses the input for P_1, can distinguish with significant probability, a view obtained by running our proposed PCI-Any-DC protocol for ECDSA signatures in Algorithm 2 between a PPT adversary $\mathcal{A}$ and honest party (say P_2 without loss of generality), and a simulated execution of the protocol between $\mathcal{S}$ and $\mathcal{F}_{\mathsf{PCI}}$(PCI-Any-DC) (for the two-party setting). The environment $\mathcal{Z}$'s view consists of the intermediate messages sent and received by the adversary $\mathcal{A}$, the input he chose for the honest party P_2, along with output of P_2.

We now describe the construction of the simulator $\mathcal{S}$, which proceeds as follows:

Input Phase: The simulator internally runs the real-world adversary $\mathcal{A}$ to obtain the private and public inputs for the corrupt party P_1. Let the inputs be as follows.

Private inputs for P_1 : $\mathsf{inp}_{1,1} = [(Y_{1,\ell}, s_{1,\ell}^{-1}, \mathsf{m}_{1,\ell})]_{\ell \in [1,N_1]}$, where each $Y_{1,\ell}$ is shared as $[Y_{1,\ell}]_{\mathcal{G}_2}$ using Input-G, and each $s_{1,\ell}^{-1}$ is shared as $\left[s_{1,\ell}^{-1}\right]$ using Input-F.

Public inputs for P_1 : $\mathsf{inp}_{1,2} = [(r_{1,\ell}, R_{1,\ell}, \mathsf{m}_{1,\ell})]_{\ell \in [1,N_1]}$.

Invoking $\mathcal{F}_{\mathsf{PCI}}$(PCI-Any-DC)**:** The simulator $\mathcal{S}$ now invokes the ideal functionality $\mathcal{F}_{\mathsf{PCI}}$(PCI-Any-DC) using the inputs of the corrupt party P_1 (the input of the honest party P_2 is directly provided to $\mathcal{F}_{\mathsf{PCI}}$(PCI-Any-DC) by the environment $\mathcal{Z}$) to receive $(\mathsf{out}_{\mathsf{PCI\text{-}Any\text{-}DC}}(\mathsf{inp}_1, \mathsf{inp}_2), N_2)$, where N_2 is the number of inputs for the honest party P_2,

and

$$\mathsf{out}_{\mathsf{PCI\text{-}Any\text{-}DC}}(\mathsf{inp}_1, \mathsf{inp}_2) = \Big(\big\{\overline{\mathsf{m}}(\mathsf{inp}_{i,1})\big\}_{i\in[1,2]}, \\ \big\{\big(Y, \{\mathsf{m}_{\mathsf{inp}_i}(Y)\}_{i\in\{1,2\}}\big) : Y \in \mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \mathsf{inp}_2)\big\} \Big)$$

where

$$\mathsf{out}_{\mathsf{PCI\text{-}Any}}(\mathsf{inp}_1, \mathsf{inp}_2) = \big\{Y \in \mathsf{id}(\mathsf{inp}_{1,1}) \cap \mathsf{id}(\mathsf{inp}_{2,1}) : \\ \mathbf{R}_{\mathsf{PCI\text{-}Any},\mathsf{inp}_1}(Y) = \mathbf{R}_{\mathsf{PCI\text{-}Any},\mathsf{inp}_2}(Y) = 1\big\}$$

Simulating Honest Party's Inputs: The simulator $\mathcal{S}$ now simulates a dummy private and public input on behalf of the honest party P_2 for the rest of the protocol as follows:

Simulated private inputs for P_2: $\overline{\mathsf{inp}}_{2,1} = [(\overline{Y}_{2,\ell}, \overline{s}_{2,\ell}^{-1}, \mathsf{m}_{2,\ell})]_{\ell\in[1,N_2]}$, where:

- Each $\overline{Y}_{2,\ell}$ is sampled uniformly at random from $\mathcal{G}$ and input to the Input-G subfunctionality in $\mathcal{F}[\mathcal{G}]$.
- Each $\overline{s}_{2,\ell}^{-1}$ is sampled uniformly at random from Z_p and input to the Input-F subfunctionality in $\mathcal{F}[F_p]$.
- Each $\mathsf{m}_{2,\ell}$ is provided to $\mathcal{S}$ by $\mathcal{F}_{\mathsf{PCI}}(\mathsf{PCI\text{-}Any\text{-}DC})$.

Simulated public inputs for P_2: $\overline{\mathsf{inp}}_{2,2} = [(\overline{r}_{2,\ell}, \overline{R}_{2,\ell}, \mathsf{m}_{2,\ell})]_{\ell\in[1,N_2]}$, where:

- Each $\overline{R}_{2,\ell}$ is sampled uniformly at random from $\mathcal{G}$.
- Each $\overline{r}_{2,\ell}$ is set to be the x-coordinate of $\overline{R}_{2,\ell}$.
- Each $\mathsf{m}_{2,\ell}$ is provided to $\mathcal{S}$ by $\mathcal{F}_{\mathsf{PCI}}(\mathsf{PCI\text{-}Any\text{-}DC})$.

Proceeding with the Protocol: The simulator now proceeds exactly as in the real protocol described in Algorithm 2. We note here that for each $(\ell, \ell') \in [N_1, N_2]$, prior to the output stage in Line 25 of Algorithm 2, the entire computation of the protocol is local. Thus, the environment's view, up to this point, will not leak whether inputs used by honest players' are dummy inputs or the ones the environment provided. Note that in the meantime, the simulator $\mathcal{S}$ can query the respective sub-functionalities from $\mathcal{F}[\mathcal{G}]$ (which includes the sub-functionalities from $\mathcal{F}[F_p]$).

Handling Openings of $C''_{\ell,\ell'}$: We note that for each $(\ell, \ell') \in [N_1, N_2]$, Line 25 of Algorithm 2 involves openings that reveal $C''_{\ell,\ell'}$ values that are either $0_{\mathcal{G}}$ (the point of infinity, which is the additive identity of the group of EC points $\mathcal{G}$) or a uniformly random element in the group of EC points $\mathcal{G}$. We note that $\mathcal{S}$ knows precisely which (ℓ, ℓ') tuples result in the opening of a $C''_{\ell,\ell'}$ value that is equal to $0_{\mathcal{G}}$: this corresponds to an intersecting public key Y which $\mathcal{S}$ can figure out deterministically given $\mathsf{out}_{\mathsf{PCI\text{-}Any\text{-}DC}}(\mathsf{inp}_1, \mathsf{inp}_2)$. $\mathcal{S}$ now proceeds as follows:

- If (ℓ, ℓ') tuples result in the opening of a $C''_{\ell,\ell'}$ value that is equal to $0_{\mathcal{G}}$: Suppose the current execution using the dummy inputs for the honest party P_2 leads to a value $C''_{\ell,\ell'} = Q'$ for some EC point $Q' \in \mathcal{G}$. $\mathcal{S}$ modifies the simulated share of $C''_{\ell,\ell'}$ corresponding to the honest party P_2 by dividing (i.e., subtracting the EC point) Q' from it locally, and modifies the MAC value by dividing (i.e. subtracting the EC point) $Q^{r\alpha}$ from the original MAC value. This is possible since $\mathcal{S}$ knows the MAC key α.
- If (ℓ, ℓ') tuples result in the opening of a $C''_{\ell,\ell'}$ value that is a uniformly random element in the group of EC points $\mathcal{G}$: in this case, $\mathcal{S}$ samples $r \leftarrow Z_p$, randomizes the simulated share of $C''_{\ell,\ell'}$ corresponding to the honest party P_2 by multiplying (i.e. adding the EC point) Q^r to it locally, and modifies the MAC value by multiplying (i.e., adding the EC point) $Q^{r\alpha}$ to the original MAC value.

Both of the above steps are possible since $\mathcal{S}$ knows the MAC key α.

Handling Openings of $Y_{1,\ell}$ values: Finally, Line 26 of Algorithm 2 involves openings that reveal public keys $Y_{1,\ell}$. Note that here, it suffices for the simulator $\mathcal{S}$ to proceed exactly as in the real protocol, since the public keys in the input of the corrupted party P_2 are available to the simulator $\mathcal{S}$ in the clear, and were shared by $\mathcal{S}$ exactly as in the real protocol.

It is easy to see that the view of $\mathcal{Z}$ is identical to that in the $\mathcal{F}[\mathcal{G}]$-hybrid model. Hence, assuming a secure SPDZ-based instantiation of the $\mathcal{F}[\mathcal{G}]$-hybrid model using our proposed MPC framework for generic group operations, our ECDSA-based PCI-Any-DC protocol securely emulates $\mathcal{F}_{\mathsf{PCI}}$(PCI-Any-DC). This concludes the proof of Theorem 1.

4.2 Extensions of ECDSA-based PCI-Any-DC

In this section, we discuss various possible extensions of our ECDSA-based two-party PCI-Any-DC protocol, including extensions to PCI-Any and PCI-All, as well as extensions to the multi-party setting.

Extension to PCI-Any. One can naturally upgrade the above PCI-Any-DC protocol to a PCI-Any protocol that additionally guarantees privacy of the input claims for each party by treating the claims as part of the private input. More concretely, the claims would be secret-shared across the participating parties instead of being publicly available, and all operations on the input claims would have to be performed inside the MPC protocol. While the extension is conceptually simple, it incurs some additional costs. For instance, we can no longer directly use our proposed optimizations to reduce or obviate the non-algebraic operations in the Verify algorithm, and we would incur the additional cost of performing these operations *inside* the underlying MPC protocol. We would also incur the additional cost of hashing the claims *inside* the MPC protocol (since the claims would now be secret-shared as opposed to being publicly available). One could use an MPC-friendly family of hash functions [GRR+16], but this would be non-compliant with standardized implementations of ECDSA that typically do not use such hash function families. We leave it as an interesting future direction to investigate optimization strategies that would allow performing the above operations efficiently (i.e.,

outside the MPC protocol) while ensuring privacy of the input claims *and* maintaining compliance with standardized ECDSA implementations.

Extension to PCI-All. The above PCI-Any-DC protocol can also be extended naturally to PCI-All by iterating through all the claims to validate the certificates on these claims by a specific certifier. To enable this, the private inputs will be ordered in a 2-D grid, where each row corresponds to the certificates by a certifier on all the claims in $\mathsf{inp}_{i,1}$, and the protocol needs to validate $|\mathsf{inp}_{i,1}|$ certificates per certifier inside the MPC protocol. The complexity grows with the number of claims which seems unavoidable since the ECDSA signatures cannot be aggregated across different claims. Therefore in the next section, we introduce an optimized PCI-All protocol using the BLS signature scheme [BLS01] that only requires a single signature verification per certifier inside the MPC protocol.

Extension to Multi-Party PCI-Any-DC. Finally, we present a discussion on how to extend the above PCI-Any-DC protocol (and its upgradation to PCI-Any) from the two-party to the multi-party setting. We divide the discussion into three phases - the input phase, the certificate validation phase, and the certifier matching phase.

Input Phase. To begin with, we can directly replicate the input phase of our two-party PCI-Any-DC protocol in the n-party setting. Concretely, as in the original two-party protocol (Lines 1-6 of Algorithm 2), each of the n participating parties inputs tuples of (identifier, certificate, claim) (with the same modifications/optimizations as described in Section 4) as its private input, and the corresponding claim and (r, R) for each tuple as its public input.

Certificate Validation Phase. We again directly replicate the certificate validation phase of our two-party PCI-Any-DC protocol in the n-party setting. Concretely, as in the original two-party protocol (Lines 7-11 and 12-16 of Algorithm 2), the protocol validates the signatures for each of the n participating parties. The validation proceeds in parallel for each of the parties.

Certifier Matching Phase. This is where the n-party version of our protocol involves a non-trivial extension of the original two-party protocol (Lines 19-26 of Algorithm 2). Note that a trivial extension of the protocol would involve n-nested loops (one for each participating party), thereby yielding a protocol with computational and communication complexity $O\left(\prod_{i\in[1,n]} |\mathsf{inp}_i|\right)$, which is clearly undesirable (this is, in fact, approximately $O(c^n)$ times more expensive that the two-party protocol, assuming a minimum input size of $O(c)$ per party, i.e., the overheads grow exponentially in the number of parties). It turns out, however, that this trivial extension essentially matches the certifier public keys across "all" parties in a pair-wise fashion, which is clearly unnecessary. In fact, without loss of generality, it suffices to simply match each certifier public key input by party P_1 with the corresponding certifier public keys across parties $P_2, \ldots, P_n$. This reduces the required number of checks from $O\left(\prod_{i\in[1,n]} |\mathsf{inp}_i|\right)$ to $O\left(|\mathsf{inp}_1| \cdot \sum_{i\in[2,n]} |\mathsf{inp}_i|\right)$, which is significantly more efficient (in fact, we now incur approximately $O(n)$ times more checks that the two-party protocol). In fact, by choosing

P_1 to be the party with the smallest input size, we can optimize the overheads even further.

Concretely, in the certifier matching phase of the n-party version of our PCI-Any-DC protocol, we run a two-nested set of loops – an outer loop for party P_1 and $(n-1)$ inner loops for parties P_2 through P_n. Each inner loop performs essentially the same computation as in Lines 22-23 of Algorithm 2, except that we now accumulate the outcomes of each of these *intra-loop* computations into a global check variable C maintained across all of these inner loops. This two-layered accumulation step (requiring an intra-loop accumulation followed by an inter-loop accumulation), however, reduces to computing a Boolean formula in the conjunctive normal form (CNF). This is because we require a multiplication operation for the intra-loop accumulation (in order to check whether there is at least one matching certifier identity between P_1 and P_j) and an addition operation across the loops to compute the final intersection. Computing such a CNF formula necessitates using field elements for accumulation, which is not immediate from Algorithm 2, where the variable $C''_{\ell,\ell'}$ is actually a group element and is amenable to field operations. Hence, we need to hash $C''_{\ell,\ell'}$ into a corresponding field element $c_{\ell,\ell'}$, which incurs the additional cost of computing $O\left(|\mathsf{inp}_1| \cdot \sum_{i\in[2,n]} |\ inp_i|\right)$ instances of a collision-resistant hash function inside the MPC protocol[4]. Along the way, we also incur some other additional costs, such as sampling additional randomnesses for each inner loop and an n-party opening protocol for the final accumulation variable.

Assuming $O(n)$ overheads for each such operation, the overall computational and communication complexities scales as $O\left(n \cdot |\mathsf{inp}_1| \cdot \sum_{i\in[2,n]} |\mathsf{inp}_i|\right)$, which is approximately $O(n^2)$ times more expensive than the two-party protocol for realistic input sizes. In particular, we expect that for large values of n (i.e. for multi-party PCI involving a large number of parties), our proposed solution will be significantly more efficient that the naïve extension of the two-party solution outlined above (even taking into the hidden constants due to the additional overheads incurred by our proposed solution). We leave it an as open question to investigate additional optimizations (or alternative solutions) that could allow reducing the overheads even further.

Extension to Multi-Party PCI-Any. Finally, analogous to the two-party setting, one can naturally upgrade the above n-party PCI-Any-DC protocol to an n-party PCI-Any protocol that additionally guarantees privacy of the input claims for each party by treating the claims as part of the private input. More concretely, the claims would be secret-shared across all of the n participating parties instead of being publicly available, and all operations on the input claims would have to be performed inside the MPC protocol. As discussed in Section 4, this would incur additional costs of performing certain operations (such as field inversions, extraction of point coordinates, and hashing of claims) inside the MPC protocol. We again leave it as an interesting future direction to investigate optimization strategies that would allow performing the above operations efficiently (i.e., outside the MPC protocol) in the n-party setting.

[4]We can use an MPC-friendly hash here [GRR+16] to optimize the associated overheads

5 PCI-All using BLS signature

This section provides a concrete instantiation of the PCI-All protocol using the BLS signature scheme [BLS01, BDN18, BGLS03]. At a high level, we use the aggregatable feature of BLS signatures over different claims to minimize the number of signature verifications inside the PCI-All protocol. Note however that BLS signature verification involves EC pairings, which we handle in a black-box way using **Tier-3** (Section 3) of our proposed MPC engine.

Notations. Let $e : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{G}_T$ be a non-degenerate, efficiently computable bilinear pairing, where $\mathcal{G}_1, \mathcal{G}_2$ are elliptic curve groups and $\mathcal{G}_T$ is a multiplicative group, all of prime order p. Let Q_1 and Q_2 be generators of $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively, and hence $g_T = e(Q_1, Q_2)$ is a generator of $\mathcal{G}_T$.

The BLS Signature Scheme. We briefly describe the key generation, signing and verification algorithms of the BLS signature scheme, followed by the algorithms for signature aggregation (over multiple messages signed under the same verification key) and the verification of aggregate signatures.

$\mathsf{KeyGen}(\lambda)$**:** On input a security parameter λ, the key generation algorithm samples a private signing key $x \leftarrow [1, p-1]$ and computes the public verification key as $Y = Q_2^x \in \mathcal{G}_2$. The algorithms outputs the key pair (x, Y).

$\mathsf{Sign}(x, m)$**:** On input a signing key x and message m, the signing algorithm first computes $M = H(m) \in \mathcal{G}_1$ where $H : \{0,1\}^* \to \mathcal{G}_1$. The algorithm then computes and outputs the signature $\sigma = M^x \in \mathcal{G}_1$.

$\mathsf{Verify}(Y, \sigma, m)$**:** On input a verification key Y, a signature σ and a message m, the verification algorithm outputs 1 if $e(\sigma, Q_2) = e(M, Y)$, and 0 otherwise.

Signature aggregation: On input signature-message pairs $\{\sigma_i, m_i\}_{i\in[1,N]}$, the signature aggregation algorithm produces an aggregated signature $\sigma_{(m_1,\ldots,m_N)} = \prod_{i\in[1,N]} \sigma_i$.

Aggregated signature verification: On input a verification key Y, an aggregated signature $\sigma_{(m_1,\ldots,m_N)}$ and a list/multiset of messages $(m_1, \ldots, m_N)$, the aggregated signature verification algorithm outputs 1 if $e(\sigma_{(m_1,\ldots,m_N)}, Q_2) = \prod_{i\in[1,N]} e(M_i, Y)$ where $M_i = H(m_i)$. The algorithm outputs 0 otherwise.

Remark. We note here that BLS signature aggregation is susceptible to a rogue public key attack when aggregating signatures on the same message under different verification keys. However, the attack is not applicable when aggregating signatures over multiple messages signed under the same public verification key, and hence does not impact the security of our proposed protocol.

Algorithm 3: PCI-All **using BLS**

P_1 has $\mathsf{inp}_{1,1} = [(Y_{1,\ell_1}, \sigma_{1,\ell_1,\ell_2}, \mathsf{m}_{1,\ell_2})]_{\ell_1 \in [1,N_{1,1}], \ell_2 \in [1,N_{1,2}]}$ and $\mathsf{inp}_{1,2} = \{\mathsf{m}_{1,\ell_2}\}_{\ell_2 \in [1,N_{1,2}]}$

P_2 has $\mathsf{inp}_{2,1} = [(Y_{2,\ell_1}, \sigma_{2,\ell_1,\ell_2}, \mathsf{m}_{2,\ell_2})]_{\ell_1 \in [1,N_{2,1}], \ell_2 \in [1,N_{2,2}]}$ and $\mathsf{inp}_{2,2} = \{\mathsf{m}_{2,\ell_2}\}_{\ell_2 \in [1,N_{2,2}]}$

Private inputs from P_1: the aggregated tuples and the set of preempted pairings

(i) $\overline{\mathsf{inp}}_{1,1} = \left[\left(Y_{1,\ell}, \overline{\sigma}_{1,\ell}, \overline{M}_1\right)\right]_{\ell \in [1,N_{1,1}]}$

(ii) $\{z_{1,\ell} = e(\overline{M}_2, Y_{1,\ell})\}_{\ell \in [1,N_{1,1}]}$

where $\overline{\sigma}_{i,\ell} = \prod_{\ell_2 \in [1,N_{i,2}]} \sigma_{i,\ell,\ell_2}$ and $\overline{M}_i = \prod_{\ell \in [1,N_{i,2}]} H(\mathsf{m}_{i,\ell})$. Note that each $Y_{1,\ell}$ is secret-shared as $[Y_{1,\ell}]_{\mathcal{G}_2}$, each $\overline{\sigma}_{1,\ell}$ is secret-shared as $[\overline{\sigma}_{1,\ell}]_{\mathcal{G}_1}$, and each $z_{1,\ell}$ is secret-shared as $[z_{1,\ell}]_{\mathcal{G}_T}$.

Public inputs from P_1: $\mathsf{inp}_{1,2}$.

Private inputs from P_2: the aggregated tuples and the set of preempted pairings

(i) $\overline{\mathsf{inp}}_{2,1} = \left[\left(Y_{2,\ell}, \overline{\sigma}_{2,\ell}, \overline{M}_2\right)\right]_{\ell \in [1,N_{2,1}]}$

(ii) $\{z_{2,\ell} = e(\overline{M}_1, Y_{2,\ell})\}_{\ell \in [1,N_{2,1}]}$

Note that each $Y_{2,\ell}$ is secret-shared as $[Y_{2,\ell}]_{\mathcal{G}_2}$, each $\overline{\sigma}_{2,\ell}$ is secret-shared as $[\overline{\sigma}_{2,\ell}]_{\mathcal{G}_1}$, and each $z_{2,\ell}$ is secret-shared as $[z_{2,\ell}]_{\mathcal{G}_T}$.

Public inputs from P_2: $\mathsf{inp}_{2,2}$.

for $\ell := 1 \ldots N_{1,1}$ **do**

  $\left[z'_{1,\ell}\right]_{\mathcal{G}_T} :=$ Pair-G2-P$([\overline{\sigma}_{1,\ell}]_{\mathcal{G}_1}, Q_2)$

for $\ell' := 1 \ldots N_{2,1}$ **do**

  $\left[z'_{2,\ell'}\right]_{\mathcal{G}_T} :=$ Pair-G2-P$([\overline{\sigma}_{2,\ell'}]_{\mathcal{G}_1}, Q_2)$

The parties agree on public random $r \leftarrow Z_p$.

for $\ell := 1 \ldots N_{1,1}$ **do**

  for $\ell' := 1 \ldots N_{2,1}$ **do**

    Generate secret-shared randomness $[r_{\ell,\ell'}] \leftarrow$ Rand-F.

    Each party locally computes:

    $[c_{\ell,\ell'}]_{\mathcal{G}_T} := \left([z_{1,\ell}]_{\mathcal{G}_T} / \left[z'_{2,\ell'}\right]_{\mathcal{G}_T}\right) \cdot \left([z_{2,\ell'}]_{\mathcal{G}_T} / \left[z'_{1,\ell}\right]_{\mathcal{G}_T}\right)^r$

    $\left[c'_{\ell,\ell'}\right]_{\mathcal{G}_T} :=$ Exp-G-S $\left([r_{\ell,\ell'}], [c_{\ell,\ell'}]_{\mathcal{G}_T}\right)$

    Output-G$\left(\left[c'_{\ell,\ell'}\right]_{\mathcal{G}_T}\right)$

    if $c'_{\ell,\ell'} == 1_T$ **then**

      Output-G$\left([Y_{1,\ell}]_{\mathcal{G}_2}\right)$

Protocol Overview. We follow the same generic approach as in our ECDSA-based protocol, with some optimizations to reduce BLS signature verifications inside the MPC protocol. We note here that we could have a single certifier issue multiple certificates on the same claim for some of the claims. However, we prescribe the parties to select only one certificate from a single certifier on each claim, i.e., there is a single (certificate, claim) pair for each certifier per claim input to the protocol. We also expect an honest party to only input valid certificates on its set of public claims.

Reducing Claim Validation: As mentioned earlier, trivially extending the approach used in our ECDSA-based PCI-Any-DC protocol to design a PCI-All protocol would require iterating through all of the public claims, and validate the certificates on these claims by a specific certifier. This results in a claim validation complexity that grows with the number of claims, which is undesirable because the straightforward way of claim validation using BLS signatures would require computing two bilinear pairings inside the MPC protocol per validation, which is prohibitively expensive. Our main effort here is to reduce the number of pairing operations inside the MPC protocol as far as possible. To do this, we first use BLS signature aggregation over multiple claims signed under the same public verification key. Concretely, suppose that the private input $\mathsf{inp}_{i,1}$ for each (honest) party P_i is ordered in a 2-D grid of tuples of the form

$$\mathsf{inp}_{i,1} = \left[\left(Y_{i,\ell_1}, \sigma_{i,\ell_1,\ell_2}, \mathsf{m}_{i,\ell_2}\right)\right]_{\ell_1 \in [1,N_{i,1}], \ell_2 \in [1,N_{i,2}]}$$

with $N_{i,1}$ certifiers and $N_{i,2}$ claims to be validated, where row-ℓ_1 contains certificates of the form σ_{i,ℓ_1,ℓ_2} on the claim m_{i,ℓ_2}, signed by the certifier associated with the verification key Y_{i,ℓ_1}. The party P_i performs some pre-processing to aggregate the certificates in each row using the BLS signature aggregation algorithm as:

$$\overline{\sigma}_{i,\ell_1} = \prod_{\ell_2 \in [1,N_{i,1}]} \sigma_{i,\ell_1,\ell_2}, \quad \overline{M}_i = \prod_{\ell_2 \in [1,N_{i,1}]} H\left(\mathsf{m}_{i,\ell_2}\right)$$

and uses an *aggregated private input* of the form

$$\overline{\mathsf{inp}}_{i,1} = \left[\left(Y_{i,\ell_1}, \overline{\sigma}_{i,\ell_1}, \overline{M}_i\right)\right]_{\ell_1 \in [1,N_{i,1}]}$$

for the MPC protocol. This now reduces the number of pairing computations inside the MPC protocol to two per certifier (required to verify each aggregated certificate); in particular, the complexity no longer grows with the number of public claims to be validated.

The next optimization involves further reducing the number of pairing computations inside the MPC to one per certifier. Note that we could avoid the pairing computation that requires pairing the public key with the aggregated claim-hash by having each party pre-compute this and directly input it to the MPC protocol. Note, however, that doing this naïvely would break the "unforgeability" guarantee of our protocol because a malicious party could simply input the pairing of a (potentially) invalid signature with the group generator Q_2 to trivially satisfy the verification check. To counter this, we exploit the uniqueness of BLS signatures for a given (key, claim) pair as follows: each party preempts the output of pairing its own verification keys with the aggregated claim-hashes of the other party (this is possible since the claims are public), which in the case of an intersecting certifier (i.e. when the verification keys are the same), is identical to the pairing of the aggregated public claim-hashes with the other party's verification key. This enables performing certificate verification for one party by using the preempted pairing values computed by the other party. This obviates the need for computing one of the pairings inside the MPC protocol (since the preempted pairing computation is done outside the MPC), while also preserving security of the end-to-end protocol.

Computing the Intersection: In addition to certificate verification, the above step also enables computing the intersection of the identity sets between the two parties. In particular, we perform an equality check in $\mathcal{G}_T$ by simply dividing the corresponding group elements, and checking that the result opens to the identity element in $\mathcal{G}_T$. As in our ECDSA-based protocol, it is important to hide the output of this computation if it

is not the identity; otherwise we leak information about the public keys which are not part of the output set, which is not an allowed leakage according to our definition. So, we randomize the difference before opening while retaining the identity value.

The detailed description of our $\mathsf{PCI\text{-}All}$ protocol for BLS signatures is provided in Algorithm 3. Here, each party P_i inputs tuples of (identifier, aggregated certificate, aggregated claim-hash) as its private input $\mathsf{inp}_{i,1}$, and the corresponding claims for each tuple as part of its public input $\mathsf{inp}_{i,2}$ (for the honest parties, $\mathsf{inp}_{i,2}$ is expected to be simply the set of public claims as in the definition of $\mathsf{PCI\text{-}All}$ in Section 2). Each party also inputs the preempted pairing outputs as described earlier. We describe the protocol in the $(\mathcal{F}[\mathsf{Pair}])$-hybrid model, i.e., we assume that each sub-functionality in $\mathcal{F}[\mathsf{Pair}]$ has a secure instantiation. A concrete instance of the protocol would use the SPDZ-based instantiation described in Section 3 to perform BLS signature validations while using all operations over the EC groups $\mathcal{G}_1, \mathcal{G}_2$ and the target group $\mathcal{G}_T$ and the bilinear pairing e in a black-box way.

5.1 Correctness and Security

Correctness of the protocol follows immediately. We state the following theorem for the security of the protocol:

Theorem 2. *Our proposed* $\mathsf{PCI\text{-}All}$ *protocol for BLS signatures as described in Algorithm 3 securely emulates* $\mathcal{F}_{\mathsf{PCI}}(\mathsf{PCI\text{-}All})$ *(for the two-party setting).*

Proof. We prove Theorem 2 by constructing a PPT simulator $\mathcal{S}$ such that no PPT environment $\mathcal{Z}$, who corrupts one of the parties (say P_1 without loss of generality) and chooses the input for P_1, can distinguish with significant probability, a view obtained by running our proposed $\mathsf{PCI\text{-}Any\text{-}DC}$ protocol for ECDSA signatures in Algorithm 3 between a PPT adversary $\mathcal{A}$ and honest party (say P_2 without loss of generality), and a simulated execution of the protocol between $\mathcal{S}$ and $\mathcal{F}_{\mathsf{PCI}}(\mathsf{PCI\text{-}All})$ (for the two-party setting). The environment $\mathcal{Z}$'s view consists of the intermediate messages sent and received by the adversary $\mathcal{A}$, the input he chose for the honest party P_2, along with output of P_2.

We now describe the construction of the simulator $\mathcal{S}$, which proceeds as follows:

Input Phase: The simulator internally runs the real-world adversary $\mathcal{A}$ to obtain the private and public inputs for the corrupt party P_1. Let the inputs be as follows.

Private inputs for P_1 : aggregated tuples $\overline{\mathsf{inp}}_{1,1} = \left[\left(Y_{1,\ell}, \overline{\sigma}_{1,\ell}, \overline{M}_1\right)\right]_{\ell \in [1,N_{1,1}]}$ and the set of preempted pairings $\{z_{1,\ell} = e(\overline{M}_2, Y_{1,\ell})\}_{\ell \in [1,N_{1,1}]}$, where $\overline{\sigma}_{i,\ell} = \prod_{\ell_2 \in [1,N_{i,2}]} \sigma_{i,\ell,\ell_2}$ and $\overline{M}_i = \prod_{\ell \in [1,N_{i,2}]} H\left(\mathsf{m}_{i,\ell}\right)$. Each $Y_{1,\ell}$ is secret-shared as $[Y_{1,\ell}]_{\mathcal{G}_2}$, each $\overline{\sigma}_{1,\ell}$ is secret-shared as $[\overline{\sigma}_{1,\ell}]_{\mathcal{G}_1}$, and each $z_{1,\ell}$ is secret-shared as $[z_{1,\ell}]_{\mathcal{G}_T}$.

Public inputs for P_1 : $\mathsf{inp}_{1,2} = \{\mathsf{m}_{1,\ell_2}\}_{\ell_2 \in [1,N_{1,2}]}$.

Invoking $\mathcal{F}_{\mathsf{PCI}}(\mathsf{PCI\text{-}All})$: The simulator $\mathcal{S}$ now invokes the ideal functionality $\mathcal{F}_{\mathsf{PCI}}(\mathsf{PCI\text{-}All})$ using the inputs of the corrupt party P_1 (the input of the honest party P_2 is directly provided to $\mathcal{F}_{\mathsf{PCI}}(\mathsf{PCI\text{-}All})$ by the environment $\mathcal{Z}$) to receive $(\mathsf{out}_{\mathsf{PCI\text{-}All}}(\mathsf{inp}_1, \mathsf{inp}_2), N_2)$, where N_2 is the number of inputs for the honest party P_2, and

$$\mathsf{out}_{\mathsf{PCI\text{-}All}}(\mathsf{inp}_1, \mathsf{inp}_2) = \big\{Y \in \mathsf{id}(\mathsf{inp}_{1,1}) \cap \mathsf{id}(\mathsf{inp}_{2,1}) : \mathbf{R}_{\mathsf{PCI\text{-}All},\mathsf{inp}_1}(Y) = \mathbf{R}_{\mathsf{PCI\text{-}All},\mathsf{inp}_2}(Y) = 1\big\}$$

Simulating Honest Party's Inputs: The simulator $\mathcal{S}$ now simulates a dummy private and public input on behalf of the honest party P_2 for the rest of the protocol as follows:

Simulated private inputs for P_2: $\overline{\overline{\mathsf{inp}}}_{2,1} = \big[\big(\overline{Y_{2,\ell}}, \overline{\overline{\sigma}}_{2,\ell}, \overline{M}_2\big)\big]_{\ell \in [1,N_{2,1}]}$ and the set of preempted pairings $\{\overline{z_{2,\ell}} = e(\overline{M}_1, \overline{Y_{2,\ell}})\}_{\ell \in [1,N_{2,1}]}$, where:

- Each $\overline{Y_{2,\ell}}$ is sampled uniformly at random from $\mathcal{G}_2$ and input to the Input-G sub-functionality in $\mathcal{F}[\mathcal{G}]$ instantiated for $\mathcal{G}_2$.
- Each $\overline{\overline{\sigma}}_{2,\ell}$ is sampled uniformly at random from $\mathcal{G}_1$ and input to the Input-G sub-functionality in $\mathcal{F}[\mathcal{G}]$ instantiated for $\mathcal{G}_1$.
- Each preempted pairing $\overline{z_{2,\ell}}$ is computed as $e(\overline{M}_1, \overline{Y_{2,\ell}})$, where $\overline{M}_1 = \prod_{\ell \in [1,N_{2,2}]} H(\mathsf{m}_{2,\ell})$ is the aggregated hashed-claim corresponding to the corrupt party P_1.
- Each $\mathsf{m}_{2,\ell}$ in the simulated public inputs for P_2 is provided to $\mathcal{S}$ by $\mathcal{F}_{\mathsf{PCI}}(\mathsf{PCI\text{-}Any\text{-}DC})$, from which the simulator $\mathcal{S}$ computes $\overline{M}_2 = \prod_{\ell \in [1,N_{2,2}]} H(\mathsf{m}_{2,\ell})$.

Simulated public inputs for P_2: $\overline{\mathsf{inp}}_{2,2} = \{\mathsf{m}_{2,\ell})\}_{\ell \in [1,N_{2,2}]}$, where:

- Each $\mathsf{m}_{2,\ell}$ is provided to $\mathcal{S}$ by $\mathcal{F}_{\mathsf{PCI}}(\mathsf{PCI\text{-}All})$.

Proceeding with the Protocol: The simulator now proceeds exactly as in the real protocol described in Algorithm 3. We note here that for each $(\ell, \ell') \in [N_{1,1}, N_{2,1}]$, prior to the output stage in Line 18 of Algorithm 3, the entire computation of the protocol is local. Thus, the environment's view, up to this point, will not leak whether inputs used by honest players' are dummy inputs or the ones the environment provided. Note that in the meantime, the simulator $\mathcal{S}$ can query the respective sub-functionalities from $\mathcal{F}[\mathsf{Pair}]$ (which includes the sub-functionalities from $\mathcal{F}[F_p]$ and $\mathcal{F}[\mathcal{G}]$, initialized appropriately for $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_T$).

Handling Openings of $c'_{\ell,\ell'}$: We note that for each $(\ell, \ell') \in [N_1, N_2]$, Line 18 of Algorithm 3 involves openings that reveal $c'_{\ell,\ell'}$ values that are either $1_{\mathcal{G}}$ (identity element of the group $\mathcal{G}_T$) or a uniformly random element in $\mathcal{G}_T$. We note that $\mathcal{S}$ knows precisely which (ℓ, ℓ') tuples result in the opening of a $c'_{\ell,\ell'}$ value that is equal to $1_{\mathcal{G}_T}$: this corresponds to an intersecting public key Y which $\mathcal{S}$ can figure out deterministically given $\mathsf{out}_{\mathsf{PCI\text{-}All}}(\mathsf{inp}_1, \mathsf{inp}_2)$. $\mathcal{S}$ now proceeds as follows:

- If (ℓ, ℓ') tuples result in the opening of a $c'_{\ell,\ell'}$ value that is equal to $1_{\mathcal{G}_T}$: Suppose the current execution using the dummy inputs for the honest party P_2 leads to a value $c'_{\ell,\ell'} = h_T$ for some element $h_T \in \mathcal{G}_T$. $\mathcal{S}$ modifies the simulated share of $c'_{\ell,\ell'}$ corresponding to the honest party P_2 by dividing h_T from it locally, and modifies the MAC value by dividing h_T^{α} from the original MAC value. This is possible since $\mathcal{S}$ knows the MAC key α.
- If (ℓ, ℓ') tuples result in the opening of a $c'_{\ell,\ell'}$ value that is a uniformly random element in the group $\mathcal{G}_T$: $\mathcal{S}$ samples $r \leftarrow Z_p$, randomizes the simulated share of $c'_{\ell,\ell'}$ corresponding to the honest party P_2 by multiplying g_T^r to it locally, and modifies the MAC value by multiplying (i.e., adding the EC point) $g_T^{r\alpha}$ to the original MAC value.

Both of the above steps are possible since $\mathcal{S}$ knows the MAC key α.

Handling Openings of $Y_{1,\ell}$ values: Finally, Line 20 of Algorithm 3 involves openings that reveal public keys $Y_{1,\ell}$. Note that here, it suffices for the simulator $\mathcal{S}$ to proceed exactly as in the real protocol, since the public keys in the input of the corrupted party P_2 are available to the simulator $\mathcal{S}$ in the clear, and were shared by $\mathcal{S}$ exactly as in the real protocol.

It is easy to see that the view of $\mathcal{Z}$ is identical to that in the $\mathcal{F}[\mathcal{G}]$-hybrid model. hence, assuming a secure SPDZ-based instantiation of the $\mathcal{F}[\mathcal{G}]$-hybrid model using our proposed MPC framework for generic group operations, our ECDSA-based PCI-All protocol securely emulates $\mathcal{F}_{\mathsf{PCI}}$(PCI-All). This concludes the proof of Theorem 2.

5.2 Extension to the Multi-Party Setting

We can similarly extend our BLS-based PCI-All protocol to the n-party setting. The extension again follows the same strategy as outlined for the multi-party extension our ECDSA-based PCI-Any-DC scheme. In particular, as in the case of our PCI-Any-DC protocol, we directly replicate the input phase (and its associated pre-processing: Lines 1-6 of Algorithm 3) as well as the certificate validation phase (Lines 7-10 of Algorithm 3) of our two-party PCI-Any-DC protocol in the n-party setting, with these phases executed in parallel for each of the n participating parties.

Finally, we again exploit the fact it suffices to simply match each certifier public key input by party P_1 with the corresponding certifier public keys across parties $P_2, \ldots, P_n$ to design an n-party certifier matching phase with $O\left(|\mathsf{inp}_1| \cdot \sum_{i\in[2,n]} |\mathsf{inp}_i|\right)$ computational and communication overhead. Concretely, in the certifier matching phase of the n-party version of our PCI-All protocol, we run a two-nested set of loops – an outer loop for party P_1 and $(n-1)$ inner loops for parties P_2 through P_n. Each inner loop performs essentially the same computation as in Lines 14-16 of Algorithm 2, except that we again accumulate all of these computations into a global variable C maintained across all of these inner loops. Along the way, we similarly incur additional costs (such as hashing the accumulation variables into field elements, sampling additional randomnesses for each inner loop and running an n-party opening protocol for the final global variable), but once again, assuming $O(n)$ overheads for each such operation, the overall computational and communication complexities scale as $O\left(n \cdot |\mathsf{inp}_1| \cdot \sum_{i\in[2,n]} |\mathsf{inp}_i|\right)$. This is again

approximately $O(n^2)$ times more expensive than the two-party protocol for realistic input sizes, and is expected to be significantly more efficient that a naïve extension of the two-party solution. We again leave it an as open question to investigate additional optimizations (or alternative solutions) that could allow reducing the overheads even further.

6 Evaluation

This section details our implementation of the EC building blocks, the ECDSA-based PCI-Any-DC protocol, and the BLS-based PCI-All protocol. We independently benchmark the individual components of our protocols (including the protocols for EC operations) in a local server. We then evaluate the end-to-end performance of our PCI-Any-DC and PCI-All protocols in a LAN, an intra-continental WAN and an inter-continental WAN by spawning parties over three geographic regions across two continents.

6.1 Implementation Details

Our implementation builds on the MP-SPDZ [Kel20] framework to support the EC operations, including pairing described in Section 3. To the best of our knowledge, this is the first implementation of an MPC protocol that supports all the EC group operations as basic gates. In particular, we implement all the functionalities described in $\mathcal{F}[F_P]$, $\mathcal{F}[\mathcal{G}]$, and $\mathcal{F}[\mathsf{Pair}]$. The closest prior work [DOK+20] had implemented only two selected operations – Output-G and Exp-G-P. Our implementation of ECDSA PCI-Any-DC variant uses the standard OpenSSL (3.0) [ope] library for EC operations. For the BLS PCI-All variant, we use the RELIC toolkit [AGM+] to compute pairings and the EC operations on the corresponding groups. Both variants protect against malicious adversaries. As described earlier, our implementation builds on the SPDZ protocol with MASCOT [KOS16] pre-processing. Analyzing the single-threaded CPU bottlenecks of the protocols, we have incorporated multi-threading to parallelize parts that individual parties locally execute without involving any communication (such as steps 9, 10, 14, 15, 22, & 23 in Algorithm 2, and 8, 10, & 16 in Algorithm 3). The source code of the implementation is made available here – `https://github.com/ghoshbishakh/pci`.

6.2 Component-wise Performance Analysis

In this section we benchmark the individual operations of our proposed MPC framework for elliptic curve pairings. The different types of operations involved in the protocols can be categorized into (i) offline *pre-processing*, (ii) *input sharing*, (iii) *local operations* – performed by a party without any communication involved, e.g., Exp-G-P, (iv) *communication dependent operations* – which require inter-party communication, e.g., Exp-G-S, (v) *output* – which includes MAC-check. We perform experiments to analyze the performance of these different operations in terms of throughput (operations per second) and the impact of network latency on them. We separately compare the performance of local operations, followed by communication dependent operations including pre-processing, input sharing and output.

Table 1: Throughput (operations per second) for Local EC Operations using RELIC and OpenSSL

	RELIC - Ed25519	OpenSSL - Secp256k1
Op-G	2,254,758	459,801
Exp-G-P	7,281	2,175

Table 2: Throughput (operations per second) for Local EC Operations on Pairing-friendly Curves using RELIC

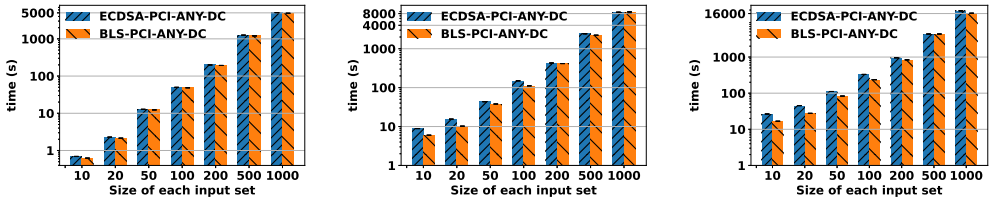
	BLS12-381	BLS12-446	BN-254	BLS12-638
Op-G : $\mathcal{G}_1$	1,079,688	834,877	687,906	435,223
Exp-G-P : $\mathcal{G}_1$	523,529	404,051	296,905	217,412
Op-G : $\mathcal{G}_2$	6,453	4,535	4,228	1,782
Exp-G-P : $\mathcal{G}_2$	3,684	2,683	1,990	1,019
Pair-G-P : $\mathcal{G}_1$, $\mathcal{G}_2$	960	689	508	307

Table 3: Throughput (operations per second) for Operations Requiring Communication

	RTT 1ms		RTT 100ms	
Pre-processing	967		267	
Input	261		245	
Output	457		363	
	Single Threaded	Multi Threaded	Single Threaded	Multi Threaded
Exp-G-S : $\mathcal{G}_1$	547	1,280	473	1,121
Exp-G-S : $\mathcal{G}_2$	277	554	257	554
Exp-G-S : $\mathcal{G}_T$	166	322	164	314
Pair-S: $\mathcal{G}_1$, $\mathcal{G}_2$	80	417	78	409

Platform Used. We used a workstation with dual Intel Xeon Gold 5118 2.30GHz CPUs, with 24 cores, and having 128 GB RAM. The system runs Ubuntu 18.04 operating system with Linux kernel version 4.15.

Local Operations. We start by benchmarking the local EC operations namely Op-G (point addition) and Exp-G-P (scalar multiplication with a point) separately for OpenSSL and RELIC. The throughput values (using a single thread) depicted in Table 1 make it evident that the performance of RELIC with Ed25519 [BDL+12] curve is significantly better than that of OpenSSL with Secp256k1 [sec] curve. Nevertheless, we use OpenSSL for our ECDSA-based implementation of PCI-Any-DC since it one of the most widely-used libraries implementing the ECDSA algorithm [DLK+14, NKS+17]. Following this, we evaluate the performance of EC operations on pairing-friendly curves with RELIC and carry out the experiments on four different curves, namely BLS12-381 [BLS02, WB19, YKS19], BN-254 [BN05], BLS12-446 [dlPVA22], and BLS12-638 [YKS19]. Table 2 summarizes the throughput for Op-G, Exp-G-P, and Pair-G-P for the above four curves. We observe that Op-G and Exp-G-P operations on $\mathcal{G}_2$ are much slower compared to that on $\mathcal{G}_1$, with Pair-G-P being the slowest operation by far. Among the curves benchmarked, BLS12-381 performs the best, and therefore we select this for the end-to-end experiments in Section 6.3.



(a) (b) (c)

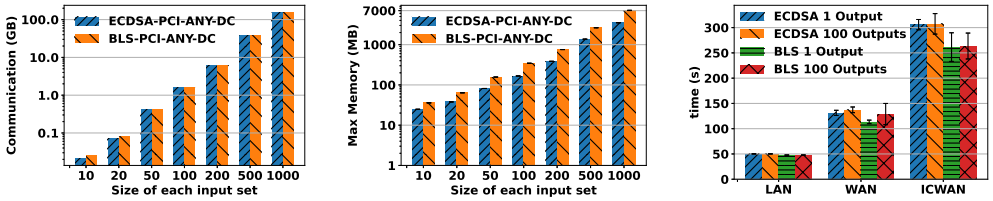
Figure 8: (a), (b) and (c) depict latency (in logarithmic scale) of ECDSA PCI-Any-DC vs BLS PCI-Any-DC in LAN, WAN and ICWAN setups respectively with commodity hardware.

Operations Requiring Communication. Moving to the more interesting benchmarks of the operations involving inter-party communication, namely *Pre-processing*, *Input*, *Output*, and *EC operations* Exp-G-S and Pair-S, we use two different setups – (a) a LAN setup with RTT between two parties being about 1ms, and (b) an emulated WAN setup with RTT of 100ms. In order to vary the link latency, we use the `tc` tool [tc] to manipulate the loopback interface. Table 3 shows the throughput observed in the single threaded and multi-threaded implementation for Exp-G-S and Pair-S. We observe that *Pre-processing* slows down significantly with increasing latency, so is *Output* but to a lesser extent. The throughput values of Exp-G-S and Pair-S operations slightly drop with increasing latency but, even with a high RTT of 100ms, multithreading significantly increases the throughput, indicating that CPU is a major bottleneck for these operations. This validates the expectation since Exp-G-S and Pair-S are performed in batches and involve only one round of communication in which a batch of tuples are partially opened (see Sections 3.2 and 3.3), thereby limiting the impact of network latency. However, if the batches are split (when a single batch becomes too large to handle), the impact of the communication latency will increase. Note that we perform this in a setup where bandwidth is sufficient enough to not be a bottleneck, and therefore, does not impact the benchmarks.

6.3 End-to-End Performance Analysis

In order to get real world performance metrics, we evaluate our implementations by placing the parties in the (a) same region – LAN, (b) different regions in the same continent – Continental WAN (WAN), and (c) different continents – Inter-continental WAN (ICWAN).

Platform Used. To gauge the practical performance of PCI on consumer hardware, we carried out the experiments on AWS EC2 `c6i.xlarge` virtual machine instances with only 4 vCPUs and 8 GB RAM. The instances were running the Ubuntu 22.04 operating system and were connected with a network having up to 12.5 Gbps bandwidth [aws]. For the ICWAN setup, we use instances located in Asia (ap-south-1) and North America (us-east-1), with an RTT latency of about 186ms. For WAN, we use two instances in the USA, one in east coast (us-east-1), and another in the west coast (us-west-1) with an RTT latency of about 62ms. For the LAN setup, we spawned the two parties in two



(a) (b) (c)

Figure 9: (a) and (b) represent total communication and maximum memory used respectively (in logarithmic scale). (c) presents the latency with different output intersection sizes.

separate VMs in the same data center (ap-south-1). We also performed experiments on more powerful hardware (48 vCPUs, 96 GB RAM), the results of which are reported in Section 6.4.

Overall Latency of PCI-Any-DC. We evaluate the end-to-end ECDSA and BLS-based PCI-Any-DC protocols, with each party's input set sizes varying from 10 to 1000. Here, the BLS PCI-Any-DC refers to the BLS PCI-All (Algorithm 3) with the parties using a single claim and its corresponding signature instead of the aggregated claim and signature. Figures Figure8a, Figure8b, and Figure8c show the mean and standard deviations of the latency in LAN, WAN, and ICWAN setups, respectively, taken over multiple runs. The y-axis shows the time taken in seconds in a logarithmic scale. For the input sets of size 10 from each party, the mean time taken is about 0.69 seconds, 8.8 seconds, and 26.4 seconds for the ECDSA PCI-Any-DC protocol in LAN, WAN, and ICWAN, respectively. In such a setting, the BLS PCI-Any-DC protocol takes 0.62 seconds, 5.9 seconds, and 16.6 seconds respectively. This is better than the ECDSA variant, albeit by a small margin because the ECDSA protocol requires additional Exp-G-S operations in the signature validation steps (lines 11 and 16 of Algorithm 2), which is not required in the BLS variant. Exp-G-S operation requires communication and hence is significantly expensive as analyzed in detail in Section 6.2. For 1000 inputs, both ECDSA and BLS PCI-Any-DC takes less than 84 minutes, 149 minutes, and 316 minutes in LAN, WAN, and ICWAN, respectively. Notably, in practice, the size of the centralized trusted set of all CAs on the web is around 200 [fir]; therefore, we expect the plausible set of certifiers for a party to be less than 200. Here the number of certifiers do not imply the global set of all possible certifiers, instead it is the number of certifiers that have issued certificates for a given claim to a user. For 200 inputs, both ECDSA and BLS PCI-Any-DC takes less than 3.5 minutes, 7 minutes, and 15 minutes in LAN, WAN, and ICWAN, respectively. This is improved further by using more powerful hardware, which we report in Section 6.4.

Communication and Memory Overhead of PCI-Any-DC. We observe that the volume of data communication across parties is deterministic and is defined by the size of their input sets as expected. Hence, there are no variations across the different runs and across LAN, WAN, and ICWAN. We report the communication bandwidth required for different input sizes in Figure9a. With input size of 10 from each party, the total volume of data communicated is 22 MB for ECDSA and 25 MB with BLS PCI-Any-DC.

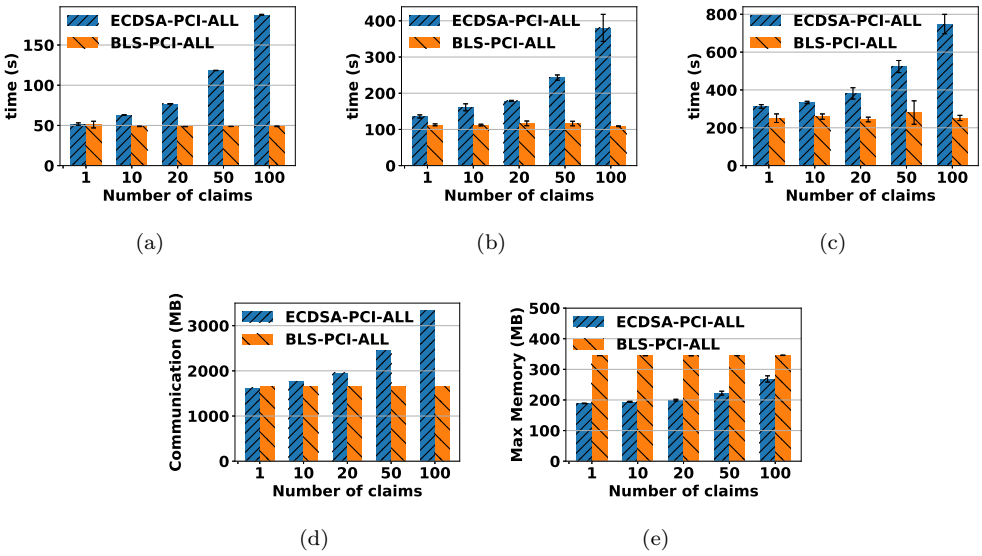
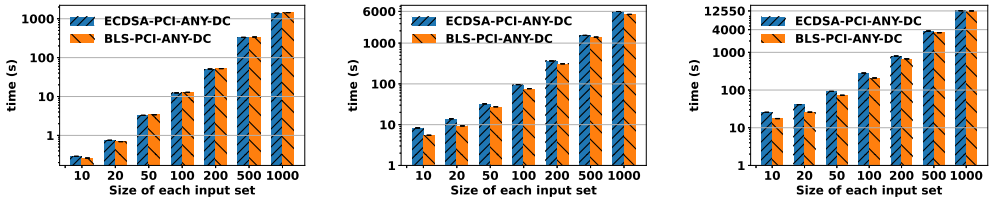


Figure 10: (a), (b) and (c) depict latency of ECDSA PCI-All vs BLS PCI-All with 100 certifiers and 1 to 100 claims as input from each party in (a) LAN (b) Continental WAN (c) Inter-continental WAN setups respectively. (d) and (e) presents the total data communicated and maximum memory consumption of PCI-All respectively.

With input sizes of 1000, the total communication goes up to 152.8 GB and 153.4 GB for ECDSA and BLS PCI-Any-DC, respectively. Unlike data communication overhead where ECDSA and BLS variants are close, the memory consumption of BLS is consistently higher as depicted in Figure9b. For 1000 inputs, ECDSA PCI-Any-DC requires around 3.4 GB memory (maximum usage during the runtime), whereas the BLS variant uses around 6.8 GB.

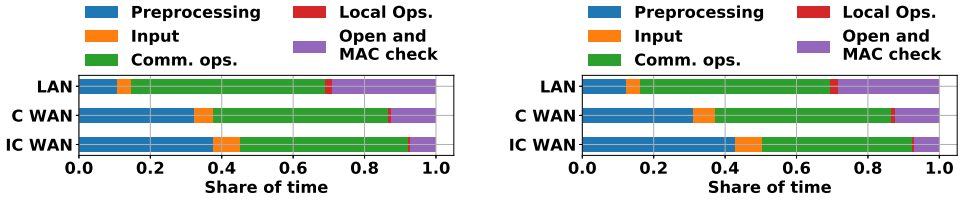
Latency of PCI-Any-DC with Varying Output Size. We evaluate the impact of varying overlap in the input certifier sets of the parties implying varying size of output intersection set. Figure9c represents the end-to-end latency of both ECDSA and BLS PCI-Any-DC while keeping the number of input from each party constant at 100, and varying the output size from 1 to 100. We observe that compared to the output size 1, the end-to-end latency for 100 outputs is higher by a very small margin on an average in all the settings, namely, LAN, WAN, and ICWAN. This is because of the differences in the number of outputs from the protocol that has to be opened (line 26 of Algorithm. 2, and line 20 of Algorithm. 3). We note, however, that no additional information is leaked outside what is permitted by the definition of PCI-Any-DC (Section 2) from the difference in the latency, since the intersection set is already known to the parties one step prior to this opening phase (line 25 of Algorithm. 2, and line 18 of Algorithm. 3).

Comparing Latency of BLS PCI-All and ECDSA PCI-All. In order to evaluate the gains of using BLS signature aggregation for PCI-All over the ECDSA implementation, we use a (somewhat artificial) construction of ECDSA-based PCI-All which iterates through all the claims to validate the certificates on them (see Section 4). We evaluate



(a) (b) (c)

Figure 11: (a), (b) and (c) depict latency in logarithmic scale of ECDSA PCI-Any-DC vs BLS PCI-Any-DC in (a) LAN, (b) WAN and (c) Inter-continental WAN setups respectively using powerful hardware.



(a) (b)

Figure 12: (a) and (b) Represents latency of different phases of the ECDSA PCI-Any-DC and BLS PCI-Any-DC respectively with 100 inputs from each party.

the end-to-end latency by keeping the input set size of each party constant at 100, and increasing the number of claims from 1 to 100. The results in Figure10a, Figure10b, and Figure10c depict the mean and the standard deviation of the overall latency in LAN, WAN and ICWAN setups, respectively, taken over multiple runs. While the BLS PCI-All consistently takes about 50 seconds, 115 seconds and 250 seconds for any number of claims (from 1 to 100) in LAN, WAN, and ICWAN setups, respectively, the time taken by ECDSA PCI-All gradually increases with the increase in the number of claims. ECDSA PCI-All takes on an average 188 seconds, 380 seconds, and 748 seconds for 100 claims in LAN, WAN and ICWAN, respectively. This clearly highlights the gains of using BLS construction of PCI-All.

Communication and Memory Overhead of PCI-All. The volume of data communicated between the two parties for the above scenario is depicted in Figure10d. With increasing number of claims, the communication overhead increases for ECDSA PCI-All, whereas it stays constant for BLS PCI-All which is the expected outcome. For 100 claims, the volume of data communicated by ECDSA PCI-All is 3333 MB, and by BLS PCI-All it is 1658 MB. Memory consumption of ECDSA PCI-All also increases with the increasing number of claims as represented by Figure10e. For 100 certifiers, with 100 claims for each party, the memory usage by ECDSA PCI-All is about 268 MB, and the same by the BLS variant is 345 MB. Overall, the memory consumption overhead of the BLS implementation is more than the ECDSA implementation for up to a reasonable number of claims such as 100.

6.4 End-to-End Performance Analysis on Powerful Hardware

In this section we present the end-to-end performance of PCI-Any-DC on devices with high compute and memory resources in LAN, WAN and ICWAN settings. The results of the same experiments but using less powerful devices that are representative of consumer hardware are presented in Section 6.3.

Platform Used. The experiments are carried out on AWS EC2 `c6i.12xlarge` virtual machine (VM) instances with 48 vCPUs and 96 GB RAM, running an Ubuntu 22.04 operating system. The instances are connected with a network having up to 18.75 Gbps bandwidth [aws]. For the ICWAN setup, we use instances located in Asia (ap-south-1) and North America (us-east-1), with an RTT latency of about 184ms. For WAN, we use two instances in the USA, one in east coast (us-east-1), and another in the west coast (us-west-1) with an RTT latency of about 68ms. For the LAN setup, we spawned the two parties in two separate VMs in the same data center (ap-south-1).

Overall Latency of PCI-Any-DC. We evaluate the end-to-end latency of ECDSA PCI-Any-DC and BLS PCI-Any-DC protocols, with each party's input set sizes varying from 10 to 1000. Here BLS PCI-Any-DC refers to the BLS PCI-All protocol (Algorithm 3), but with the parties giving a single claim and certificate as input instead of the aggregated claims and certificates. The mean (and standard deviation) of the latency in LAN, WAN, ICWAN setups are presented in Figure11a, Figure11b, and Figure11c respectively. The y-axis shows the time taken in seconds in logarithmic scale. In the LAN setting with 1000 inputs from each party, both ECDSA and BLS PCI-Any-DC take about 24 minutes, which is $\sim 71\%$ less than when using less powerful hardware having 4 vCPUs and 8GB memory (see Section 6.3). However, when the network latency increases in the ICWAN setting, the advantage of more CPU and memory resources reduces. For 1000 inputs from each party, ECDSA PCI-Any-DC takes around 211 minutes and BLS PCI-Any-DC takes 209 minutes. This is $\sim 33\%$ less than when using less powerful hardware where both ECDSA and BLS variant take less than 316 minutes. While the better hardware configuration improves the overall latency of both ECDSA and BLS based PCI-Any-DC, the volume of data communicated between the parties and the memory consumption stays unchanged from what we observed for less power hardware in Figure9a and Figure9b of Section 6.3.

Phase-wise Latency Analysis. Next we analyze the latency of different phases of the protocols. Figure12a and 12b represent the shares of time taken by different phases, namely pre-processing, input sharing, communication dependent operations, local operations and output (with MAC check) for PCI-Any-DC and PCI-All respectively, with 100 inputs from each party. We observe that pre-processing, output and input sharing phases have the greatest impact with increases in latency from LAN to ICWAN. The Exp-G-S, and Pair-S operations (denoted as Comm. ops. in the figure) on the other hand are relatively stable with varying latency, but still take the largest share of the entire runtime for our input sizes. Local operations are the cheapest as expected, and their impact on the end-to-end latency drops to insignificant percentage shares as latency increases.

7 Conclusion and Future Directions

Enabling parties to establish trust by inferring their common certification authorities without revealing their other respective certifiers will emerge as a key privacy goal in any architecture built on decentralized identities and verifiable claims, including Web 3.0. In this paper, we introduced Private Certifier Intersection (PCI) – a cryptographic primitive that allows mutually distrusting parties to establish a trust basis for cross-validation of claims if they have one or more trust authorities (certifiers) in common. We formalized the security guarantees of PCI and proposed two provably secure and practically efficient PCI protocols supporting validation of digital signature-based certificates: a PCI-Any-DC protocol for ECDSA-based certificates and a PCI-All protocol for BLS-based certificates. Along the way we have introduced a novel framework for efficient secret-sharing-based MPC over elliptic curve pairings. We have implemented and benchmarked our PCI solutions to showcase their practical efficiency.

Our work gives rise to many interesting open questions. We leave it open to study PCI in the setting where claims are private, as well as to define and realize variants of PCI that outputs a priority list of certifiers. Designing PCI protocols supporting other signature schemes, including quantum-safe schemes, is another challenging direction of research.Our MPC framework over EC pairings can plausibly be leveraged for building MPC-based PCI supporting other EC-based signature schemes. While our PCI constructions based on ECDSA and BLS cannot be be immediately/trivially extended to other signature schemes, we expect that carefully designed and specifically optimized PCI constructions supporting other EC-based signature schemes can plausibly be realized by using our proposed MPC framework over EC pairings as a building block.

Acknowledgements

We thank the anonymous reviewers of NDSS 2023 for their helpful comments and suggestions.

References

[ABG+19] Ermyas Abebe, Dushyant Behl, Chander Govindarajan, Yining Hu, Dileban Karunamoorthy, Petr Novotny, Vinayaka Pandit, Venkatraman Ramakrishna, and Christian Vecchiola. Enabling enterprise blockchain interoperability with trusted data transfer (industry track). In *Proceedings of the 20th International Middleware Conference Industrial Track*, pages 29–35, 2019.

[ACC+] Abdelrahaman Aly, K Cong, D Cozzo, M Keller, E Orsini, D Rotaru, O Scherer, P Scholl, N Smart, T Tanguy, et al. Scale–mamba. `https://github.com/KULeuven-COSIC/SCALE-MAMBA`. (Last accessed: May 13, 2022).

[AGM+] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`.

[AKB07] Giuseppe Ateniese, Jonathan Kirsch, and Marina Blanton. Secret handshakes with dynamic and fuzzy matching. In *NDSS*, volume 7, pages 43–54, 2007.

[ANS99] X9 ANSI. 62: public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ecdsa). *Am. Nat'l Standards Inst*, 1999.

[aws] Amazon ec2 c6i instances. (Last accessed: August 23, 2022).

[BDL+12] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012.

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 435–464. Springer, 2018.

[BDS+03] Dirk Balfanz, Glenn Durfee, Narendar Shankar, Diana Smetters, Jessica Staddon, and Hao-Chi Wong. Secret handshakes from pairing-based key agreements. In *2003 Symposium on Security and Privacy, 2003.*, pages 180–196. IEEE, 2003.

[BEK+21] Jan Bobolz, Fabian Eidens, Stephan Krenn, Sebastian Ramacher, and Kai Samelin. Issuer-hiding attribute-based credentials. In *International Conference on Cryptology and Network Security*, pages 158–178. Springer, 2021.

[BFGP22] Daniel Bosk, Davide Frey, Mathieu Gestin, and Guillaume Piolle. Hidden issuer anonymous credential. *Proceedings on Privacy Enhancing Technologies*, 4:571–607, 2022.

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *International conference on the theory and applications of cryptographic techniques*, pages 416–432. Springer, 2003.

[BGW+20] Dan Boneh, Sergey Gorbunov, Riad S. Wahby, Hoeteck Wee, and Zhenfei Zhang. BLS Signatures. Internet-Draft draft-irtf-cfrg-bls-signature-04, Internet Engineering Task Force, September 2020. Work in Progress.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[BLS02] Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *International conference on security in communication networks*, pages 257–267. Springer, 2002.

[BN05] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *International workshop on selected areas in cryptography*, pages 319–331. Springer, 2005.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In *Annual Cryptology Conference*, pages 3–22. Springer, 2015.

[CDE+18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spd $\mathbb{Z}_{2^k}$: efficient mpc mod 2^k for dishonest majority. In *Annual International Cryptology Conference*, pages 769–798. Springer, 2018.

[CFK+02] Karl Czajkowski, Ian Foster, Carl Kesselman, Volker Sander, and Steven Tuecke. Snap: A protocol for negotiating service level agreements and coordinating resource management in distributed systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 153–183. Springer, 2002.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *ACM CCS*, pages 1243–1255, 2017.

[CM20] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020*. Springer, 2020.

[CT10] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security, 14th International Conference, FC 2010*. Springer, 2010.

[CZ09] Jan Camenisch and Gregory M Zaverucha. Private intersection of certified sets. In *International Conference on Financial Cryptography and Data Security*, pages 108–127. Springer, 2009.

[did20] Did specification registries, 2020. (Last accessed: May 13, 2022).

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[DLK+14] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[dlPVA22] Antonio de la Piedra, Marloes Venema, and Greg Alpár. ABE Squared: Accurately benchmarking efficiency of attribute-based encryption. *Cryptology ePrint Archive*, 2022.

[DOK+20] Anders Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing dnssec keys via threshold ecdsa from generic mpc. In *European Symposium on Research in Computer Security*, pages 654–673. Springer, 2020.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[DSMRY09] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Efficient robust private set intersection. In *International Conference on Applied Cryptography and Network Security*, pages 125–142. Springer, 2009.

[fir] Ca certificates in firefox. (Last accessed: August 23, 2022).

[FNP04] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *International conference on the theory and applications of cryptographic techniques*, pages 1–19. Springer, 2004.

[GRR+16] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. Mpc-friendly symmetric key primitives. In *ACM CCS 2016*, pages 430–443. ACM, 2016.

[GVR+22] Bishakh Chandra Ghosh, Dhinakaran Vinayagamurthy, Venkatraman Ramakrishna, Krishnasuri Narayanam, and Sandip Chakraborty. Privacy-preserving negotiation of common trust anchors across blockchain networks (short paper). In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2022.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[HW12] Paul E. Hoffman and Wouter Wijngaards. Elliptic Curve Digital Signature Algorithm (DSA) for DNSSEC. RFC 6605, April 2012.

[ind] Hyperledger indy. (Last accessed: May 13, 2022).

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *ACM CCS*, 2020.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 158–189. Springer, 2018.

[KS05] Lea Kissner and Dawn Song. Privacy-preserving set operations. In *Annual International Cryptology Conference*, pages 241–257. Springer, 2005.

[Lyn] Ben Lynn. Pbc library-pairing-based cryptography. *http://crypto. stanford. edu/pbc/*.

[MBG+06] Bodo Moeller, Nelson Bolyard, Vipul Gupta, Simon Blake-Wilson, and Chris Hawk. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492, May 2006.

[NKS+17] Matus Nemec, Dusan Klinec, Petr Svenda, Peter Sekan, and Vashek Matyas. Measuring popularity of cryptographic libraries in internet-wide scans. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 162–175, 2017.

[ope] Openssl. (Last accessed: May 13, 2022).

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In {*USENIX*} *Security*, pages 515–530, 2015.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on {OT} extension. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 797–812, 2014.

[RR17] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In *ACM CCS*, pages 1229–1242, 2017.

[RSL+20] Drummond Reed, Manu Sporny, Dave Longley, Christopher Allen, Ryan Grant, Markus Sabadello, and Jonathan Holt. Decentralized identifiers (dids) v1.0, 2020. (Last accessed: May 13, 2022).

[sec] Sec 2: Recommended elliptic curve domain parameters. (Last accessed: May 13, 2022).

[SLC21] Manu Sporny, Dave Longley, and David Chadwick. Verifiable credentials data model v1.1, 2021. (Last accessed: May 13, 2022).

[STA19] Nigel P Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In *IMA International Conference on Cryptography and Coding*, pages 342–366. Springer, 2019.

[tc] tc - show / manipulate traffic control settings. (Last accessed: May 13, 2022).

[TR16] Andrew Tobin and Drummond Reed. The inevitable rise of self-sovereign identity. *The Sovrin Foundation*, 29(2016), 2016.

[tra] Tradelens. (Last accessed: August 29, 2022).

[WB19] Riad S Wahby and Dan Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *Cryptology ePrint Archive*, 2019.

[WYS+02] Marianne Winslett, Ting Yu, Kent E Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating trust in the web. *IEEE Internet Computing*, 6(6):30–37, 2002.

[Yao82] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[YKS19] Shoko Yonezawa, Tetsutaro Kobayashi, and Tsunekazu Saito. Pairing-friendly curves. *Network Working Group. Internet-Draft. January*, 2019.