Secure Set-based Policy Checking and Its Application to Password Registration (full version)†

Changyu Dong[1] and Franziskus Kiefer[2]

[1] Department of Computer and Information Sciences
University of Strathclyde, Glasgow, UK
changyu.dong@strath.ac.uk
[2] Surrey Centre for Cyber Security
Department of Computer Science, University of Surrey, UK
mail@franziskuskiefer.de

Abstract. Policies are the corner stones of today's computer systems. They define secure states and safe operations. A common problem with policies is that their enforcement is often in conflict with user privacy. In order to check the satisfiability of a policy, a server usually needs to collect from a client some information which may be private. In this work we introduce the notion of secure set-based policy checking (SPC) that allows the server to verify policies while preserving the client's privacy. SPC is a generic protocol that can be applied in many policy-based systems. As an example, we show how to use SPC to build a password registration protocol so that a server can check whether a client's password is compliant with its password policy without seeing the password. We also analyse SPC and the password registration protocol and provide security proofs. To demonstrate the practicality of the proposed primitives, we report performance evaluation results based on a prototype implementation of the password registration protocol.

1 Introduction

Policies are widely used in the context of computer systems and security. A policy defines a set of rules, over elements such as resources and participants in a system. It governs the system's behaviour with the goal of keeping the system safe. This allows organisations to ensure that the system is always in a well defined and secure state. Policies can be used in, for example, access control, authentication, trust management, firewalls and many other places.

While policies offer security protection, they sometimes raise privacy concerns [9]. This is especially true in large distributed systems such as the Internet where there is no pre-established trust relationship between parties interacting with each other. One typical scenario is that a server wants to restrict access to certain resources and defines a policy so that only those who satisfy this policy can access those resources. Often to evaluate this policy, the server needs to collect some information from a client and check the information against the policy. This information can be sensitive, e.g. credentials that should be kept private or other personal information, thus the client may not want to release it to the server. This privacy problem motivates the notion of secure set-based policy checks (SPC) we are exploring in this work.

In an SPC protocol, a server holds a public policy based on some set-theoretical semantics and the client holds a set that represents required information. After running the protocol, the server

† A shortened version of this paper appears in the proceedings of the 14th International Conference on Cryptology and Network Security (CANS 2015) 10-12 December 2015, Morocco, Marrakesh.

gets only a single bit information, i.e. whether the client's set satisfies the policy, but nothing else about the client's set. Thus SPC allows the server to securely check the policy while protecting the client's privacy. SPC is a general building block that can be applied in many scenarios to allow privacy preserving policy checking. One particular example we will show in this paper is how to enforce password policies using SPC in password registration. We will discuss more applications such as policy checks for access control, friendship analysis and genome testing in section 7.

Contributions and Organisation In this paper, we propose secure set-based policy checking (SPC), a new privacy preserving protocol. SPC uses a generic and expressive representation of policies based on the notion of sets, thus can be applied in many policy based systems. We then show an efficient instantiation of SPC based on linear secret sharing schemes and the Oblivious Bloom Intersection protocol. These two building blocks rely mostly on arithmetic operations in small fields and symmetric cryptography. As a consequence, our SPC construction is very efficient. We believe the high efficiency will make SPC an attractive choice for applications that require privacy preserving policy checking. While SPC is interesting on its own, we further show how it can be used to solve real world problems. We develop a new password registration protocol that uses SPC so that the server can verify that a password chosen by a client is compliant with a password policy without seeing the password. We analyse the security and provide proofs of both the SPC protocol and the password registration protocol. We have implemented a prototype of the password registration protocol and evaluated the performance based on the implementation. The performance figure shows that our protocol is much more efficient than the password registration protocol (ZKPPC) from [16]. Furthermore, we sketch a few other application scenarios in which SPC can be used.

The paper is organised as follows: in Section 2, we briefly review related work; in Section 3, we introduce necessary preliminaries and cryptographic building blocks; in Section 4, we show the SPC protocol; in Section 5, we show the password registration protocol; performance evaluation results are given in Section 6; in Section 7, further applications of SPC are discussed; in Section 8, we conclude the paper and discuss possible future work. In the appendix we sketch security proofs for the protocols.

2 Related Work

Policy evaluation involving sensitive information has been a long established problem. Duma et al. [9] argued that uncontrolled exposure of private information is a major risk for Internet users and showed that policy evaluation can lead to undesirable information leakage. To counter the risk, one way is to define additional policies on the client side [25]. Those policies allow the release of sensitive information only if the server can convince the client that it is trustworthy. This approach does not prevent information from flowing out of the client's control, but rather provides some assurance that only trusted servers can see the information. Another approach is to use cryptographic protocols to allow privacy preserving policy checking. In this approach, information is not revealed and the server learns only the evaluation result. It is always possible to implement a protocol for policy checking using generic two party secure computation techniques such as garbled circuits [26] but the cost would be prohibitive. Some custom protocols have been built but they either work only for a certain policy language (e.g. [17]), or they are based on cryptographic primitives such as Ciphertext Policy Attribute-based Encryption (CP-ABE) that must have a trusted third party to generate keys for users based on their private information (e.g. [19]). In contrast, SPC can support a large class of policy languages and can work without a trusted party.

Password Registration To ensure high password entropy, servers often have policies on password complexity, e.g. a valid passwords must be a mixture of lower case, upper case, numeric characters and at least of a certain minimum length. Usually the server has to see the client's password in plaintext in order to check whether the password is compliant with the policy. However, revealing its password to the server may not be a desirable option for the client (see Section 5 for a further discussion). Recent work by Kiefer and Manulis [16] proposed the first protocol that allows blind registration of client passwords at remote servers. In the protocol the client sends only a cryptographic password verifier during the registration procedure. Although the server never sees the actual password, it can still enforce password policies. This protocol provides a feasible solution that solves the aforementioned problems. However, password policy checking in [16] relies heavily on zero-knowledge proofs, which is a costly cryptographic primitive and thus renders the protocol impractical.

3 Preliminaries

3.1 Policies and Linear Secret Sharing

In this paper, we consider a set-theoretical representation of policies, i.e. *monotone access structures* [14]. A policy P defines a pair $(\mathcal{S}, \Gamma_{\mathcal{S}})$ where $\mathcal{S}$ is a set and $\Gamma_{\mathcal{S}}$ is an access structure over $\mathcal{S}$. The access structure is a subset of the powerset $2^{\mathcal{S}}$, i.e. the access structure contains zero to many subsets of $\mathcal{S}$. We say an access structure $\Gamma_{\mathcal{S}}$ is monotonic if for each element in $\Gamma_{\mathcal{S}}$, all its supersets are also in $\Gamma_{\mathcal{S}}$. We say a set $\mathcal{C}$ satisfies a policy P, written as $P(\mathcal{C}) = \texttt{true}$, if $\mathcal{C} \in \Gamma_{\mathcal{S}}$. A set $\mathcal{C}$ that satisfies P is called an authorised set. Access structures capture many complex access control and authorisation policies. For example, $\mathcal{S}$ can be a set of credentials and $\Gamma_{\mathcal{S}}$ defines a monotone boolean formula of subsets of credentials that are required for authorisation.

It has long been known that an access structure can be mapped to a linear secret sharing scheme (LSSS) [14,2] by choosing a secret and split it into a set of shares according to a given access structure $\Gamma_{\mathcal{S}}$ defined over $\mathcal{S}$. Each share is then associated with an element in $\mathcal{S}$. For convenience, we will use $\mathfrak{s}_i \stackrel{\sim}{\in} \mathcal{S}$ to denote that $\mathfrak{s}_i$ is a share associated with some element s_i in a set $\mathcal{S}$. The following holds for a LSSS: (1) any set of shares can reconstruct the secret if the elements associated with the shares form an authorised set, and (2) any set of shares does not reveal any information about the secret if the elements associated with the shares do not form an authorised set. There are generic mechanisms to generate shares from access structures and reconstruct secrets from shares, e.g. see [14,2]. Using a LSSS, checking whether a set satisfies a policy is equivalent to checking whether a set of shares can reconstruct the secret.

3.2 Oblivious Bloom Intersection

The Oblivious Bloom Intersection (OBI) protocol by Dong et al. [8] is executed between a client and a server on the respective sets $\mathcal{C}$ and $\mathcal{S}$. Originally, the OBI protocol was designed for Private Set Intersection (PSI) such that at the end of the protocol, the client learns the intersection $\mathcal{C} \cap \mathcal{S}$ and the server learns nothing. As observed in [24], OBI can be extended to a Private Set Intersection with Data Transfer protocol. In this case, the server can associate each element $s_i \in \mathcal{S}$ with a data item d_i. At the end of the protocol, for each element in the intersection the client also receives the corresponding data item from the server. The protocol can be described at a high level as follows: let the server hold a set $\mathcal{S} = \{s_i\}$ and a data set $\mathcal{S}_d = \{d_i\}$. The two sets are of equal cardinality

and each (s_i, d_i) can be viewed as a key-value pair. The server generates a garbled Bloom filter $G_{\mathcal{S}}$ on $\mathcal{S}$ and $\mathcal{S}_d$ using [24, Algorithm 1]. The garbled Bloom filter encodes both $\mathcal{S}$ and $\mathcal{S}_d$ in a way such that querying the key $s_i \in \mathcal{S}$ against $G_{\mathcal{S}}$ returns the data item d_i and querying $s_j \notin \mathcal{S}$ returns a random string. Let the client hold a set $\mathcal{C}$. The client encodes the set into a conventional Bloom filter [5] $B_{\mathcal{C}}$. The client and the server run an oblivious transfer protocol using the Bloom filter and the garbled Bloom filter as inputs. As the result, the client receives a garbled Bloom filter $G_{\mathcal{C}\cap\mathcal{S}}$ that encodes the intersection $\mathcal{C} \cap \mathcal{S}$ and the data items associated with the elements in $\mathcal{C} \cap \mathcal{S}$. Then the client can query $G_{\mathcal{C}\cap\mathcal{S}}$ with each element $c_i \in \mathcal{C}$. If c_i is in the intersection then there must be some $s_j \in \mathcal{S}$ such that $c_i = s_j$ and the query result is d_j, the data item associated with s_j, otherwise the client gets a random string.

In this paper we use OBI so that the server can send a set of secret shares to the client based on the client's set $\mathcal{C}$ without knowing anything about $\mathcal{C}$. Although in general we can use any PSI with Data Transfer protocol (e.g. [12]), we choose OBI here because of its efficiency. OBI is very efficient due to the fact that it relies mostly on hash operations. The performance can be further improved by the modifications proposed by Pinkas et al. [21]. Note that although Pinkas et al. also proposed a new PSI protocol based on hashtable + oblivious transfer in [21] that is more efficient than OBI, the new PSI protocol cannot be used in our case because it does not support data transfer.

4 Secure Set-based Policy Checking (SPC)

In this section we introduce a new protocol called secure set-based policy checking (SPC). In SPC, a server holds a public policy P as defined in Section 3 and a client holds a private set $\mathcal{C}$. The goal is to allow the server to check whether $\mathcal{C}$ satisfies P without learning anything else about $\mathcal{C}$.

Definition 1 (Secure Set-based Policy Checking, SPC). *Set-based policy checking is executed between client C with a private set $\mathcal{C}$ and server S with a public policy $P = (\mathcal{S}, \Gamma_{\mathcal{S}})$. Server and client retrieve $P(\mathcal{C})$ as result. We call a set-based policy checking protocol secure iff it fulfils the following three notions.*

1. *Correctness: Honest execution of the protocol with $P(\mathcal{C}) =$ `true` is accepted by the server with overwhelming probability.*
2. *Client Privacy: Server S learns nothing about the client set $\mathcal{C}$ other than $P(\mathcal{C})$.*
3. *Soundness: A client C holding $\mathcal{C}$ with $P(\mathcal{C}) \neq$ `true` has negligible probability in getting S to accept the SPC execution.*

Definition 1 says in particular that an SPC protocol provides both participants with the result of $P(\mathcal{C})$ while the server learns nothing about $\mathcal{C}$ more than it can infer from the result and public information.

4.1 SPC Instantiation

An overview of the proposed protocol is depicted in Fig. 1, using LSSS and OBI. Let $P = (\mathcal{S}, \Gamma_{\mathcal{S}})$ be the server's policy defined over its set $\mathcal{S}$ and $\mathcal{C}$ be the client's set. The two parties want to check $P(\mathcal{C})$, i.e. whether $\mathcal{C}$ satisfies P. In the protocol, the server first chooses a random secret and splits it according to the policy. Then the server builds a garbled Bloom filter and runs the OBI protocol with the client. At the end of the protocol, the client receives a set of shares $\{\mathfrak{s}_i | \mathfrak{s}_i \widetilde{\in} \mathcal{S} \cap \mathcal{C}\}$, i.e. each $\mathfrak{s}_i$ received is associated with an element in $\mathcal{C} \cap \mathcal{S}$. If $P(\mathcal{C}) =$ `true`, then the client can recover

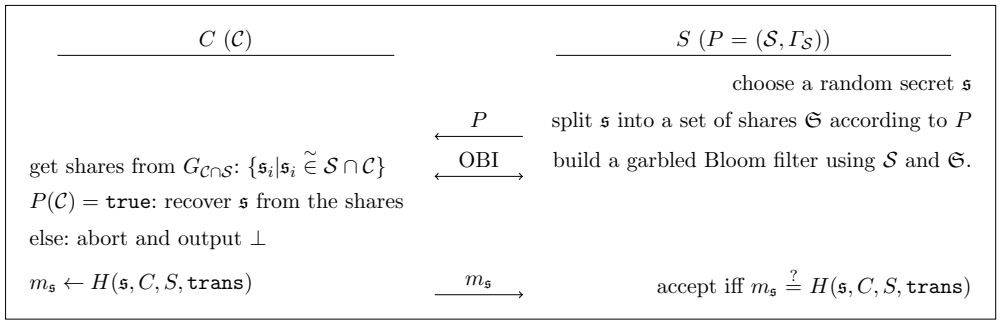

Fig. 1: Secure Set-based Policy Checking

the secret from the shares it has received, because $\mathcal{C} \cap \mathcal{S}$ must be an authorised set. If $P(\mathcal{C}) \neq \texttt{true}$ then the client will not receive enough shares that enable it to reconstruct the secret, and it learns nothing about the secret from the shares received. Therefore by checking whether the client can recover the secret, the server learns whether the client's set satisfies the policy. The protocol is defined as follows:

Public input: Both parties get a collision resistant hash function H, a LSSS scheme description, server policy $P = (\mathcal{S}, \Gamma_{\mathcal{S}})$, and security parameter λ.

1. The server first chooses a secret $\mathfrak{s}$ which is a random λ-bit string where λ is the security parameter. Then the server splits the secret into a set of shares $\mathfrak{S}$ according to its policy P using the LSSS scheme. Each share $\mathfrak{s}_i \in \mathfrak{S}$ is associated with an element in $\mathcal{S}$.
2. The server builds a garbled Bloom filter using $\mathcal{S}$ and $\mathfrak{S}$ as input such that each $s_i \in \mathcal{S}$ is a key and its associated secret share $\mathfrak{s}_i$ is the data value that is encoded in the garbled Bloom filter. The two parties then run the OBI protocol and the client using $\mathcal{C}$ as its input.
3. At the end of the OBI protocol the client gets a set of shares $\{\mathfrak{s}_i | \mathfrak{s}_i \tilde{\in} \mathcal{S} \cap \mathcal{C}\}$. If $\mathcal{C}$ satisfies policy P, then the shares obtained from the OBI protocol will allow the client to reconstruct the secret $\mathfrak{s}$, otherwise the client learns nothing about $\mathfrak{s}$ and aborts.
4. The client proves to the server that it knows $\mathfrak{s}$ by sending $m_{\mathfrak{s}} \leftarrow H(\mathfrak{s}, C, S, \texttt{trans})$ where $\mathfrak{s}$ is the secret, C and S are the identities of the two parties, and trans is the transcript of this execution. The sever checks whether $m_{\mathfrak{s}}$ is the same as it computed from its own state, if so then the client convinced the server that its set is compliant with policy P.

4.2 Security

Due to space limitations we only give lemmata and refer to the full version of this work for their proofs.

Lemma 1 (Correctness). *Let $\mathcal{C}$ and $\mathcal{S}$ denote sets from some universe and $P = (\mathcal{S}, \Gamma_{\mathcal{S}})$. Assuming the used OBI and LSSS algorithms are correct, then the SPC protocol from Figure 1 is correct, i.e. honest execution of the protocol with $P(\mathcal{C}) = \texttt{true}$ is accepted by the server with overwhelming probability.*

Lemma 2 (Privacy). *Let $\mathcal{C}$ and $\mathcal{S}$ denote sets from some universe, $P = (\mathcal{S}, \Gamma_{\mathcal{S}})$ a policy and $f_{\texttt{SPC}}(\mathcal{C}, \mathcal{S}) = (P(\mathcal{C}), P(\mathcal{C}))$. If the OBI protocol is secure and the LSSS is correct, the SPC protocol from Figure 1 securely realises $f_{\texttt{SPC}}$ in the presence of a malicious server or client.*

Lemma 2 proves that our SPC protocol ensures client privacy, i.e. does not leak any information about the client's set. We now give a lemma to show soundness of our SPC protocol that concludes the security analysis of the proposed SPC protocol.

Lemma 3 (Soundness). *Let $\mathcal{C}$ and $\mathcal{S}$ denote sets from some universe and $P = (\mathcal{S}, \Gamma_{\mathcal{S}})$ a policy. Assuming the used OBI and LSSS algorithms are secure and H is collision resistant, then the SPC protocol from Figure 1 is sound in the presence of a malicious client, i.e. the server accepts the protocol with negligible probability if $P(\mathcal{C}) \neq$* `true`.

5 A New Password Registration Protocol

Password-based authentication is the most common authentication mechanism for humans. Despite increasing attempts of replacing it from `https://fidoalliance.org/` and others, something has yet to be proposed to fully replace password-based authentication. There are many reasons why it is so difficult to transition away from passwords, e.g., low-cost, user-experience and scalability. For those reasons, passwords are likely to remain as a major authentication method in the foreseeable future. The current approach for remote registration of client passwords requires the client to send its password in plaintext to the server, which stores a value derived from the password (e.g., a hash value or a verifier) in a password database. The problem with this approach is that the server sees the plaintext password and the client has no control over what the server will do with it. At first glance, revealing the password to the server seems to be harmless, but a closer look shows the opposite. Research shows that people tend to reuse the same password across different websites [11,13,7]. In this case, a compromised or malicious server can easily break into other accounts belonging to the same client after seeing the plaintext password. Even if the server is honest, the client still has to worry about whether its password is protected properly by the server. Ideally passwords should be stored in a secure form that is hard to invert such that an attacker gaining access to the password database still has difficulties to recover the passwords. Currently, password-based authentication mechanisms in literature assume the server does this, i.e. the server is trusted to store and protect the password properly and securely. However, increasing number of successful password leaks [6,20,22] suggests that many servers fail to do so. It is desirable if the server does not see the plaintext password during registration. However, this will make it difficult for the server to check whether the password chosen by the client is complex enough or long enough.

In this section, we present a new password registration protocol as an application of SPC. The protocol allows a client to register its password *blindly* on a server while still allowing the server to check whether the password is compliant with a password policy. In the protocol, rather than sending the password in plaintext to the server, the client sends blinded characters of the password. The blinded characters enable the server to check policy compliance using an SPC protocol. If a password is valid, the blinded characters are aggregated into a verifier that is stored on the server and used in future authentication protocols. Since the blinded characters are generated with proper randomness, the client can be assured that the password is secure even if the password database is compromised (modulo unavoidable offline dictionary attacks).

5.1 Passwords and Password Policies

In this paper, we consider a password to be in the basic format of a finite length string of printable characters (ASCII, UTF-8, etc.). We do not consider other forms of passwords such as graphical

passwords [23]. It is a common practice to partition the password alphabet into character classes, e.g., upper case, lower case, symbols and digits. These character classes can be seen as disjoint subsets of the alphabet. A password policy is then defined to impose requirements for password complexity in terms of the minimum number of characters, minimal number of classes, and minimal number of characters in each class. For example, every valid password must have at least one character from each class and eight characters overall.

The connection between set-based policies from Section 3 and password policies is easy to see. Since password policies are defined in terms of thresholds and subsets over an alphabet that is a set of characters, they can be easily captured as access structures. It is also not difficult to see how SPC can be applied in the password policy checking setting, since a password can be seen as a set of characters. There is only a small gap: passwords are arbitrary strings and as such can have repeated characters. So the collection of characters in a password forms a multiset, not a set. The problem is that some policies might not be evaluated correctly using a multiset. For example, if a policy says "a valid password must have at least two symbols" and the client chooses "pa$$w0rd", using SPC directly the password will be considered invalid, even though it does contain two symbols. This can be solved by pre-processing the characters in the alphabet and passwords.

The idea of password pre-processing is to convert each character in the password into a unique symbol by appending an "index" to the character. So if the character '$' appears twice in a password, the first one becomes "$1" and the second one becomes "$2". Since $1 and $2 are two different strings, they are different elements when putting into a set. Therefore we can always convert a password into a set rather than into a multiset. The password pre-processing is performed by the client in the protocol. We define a function ψ for the client to convert a password (character string) into a set as follows. Let $\texttt{pwd} = c_1, \ldots, c_x$ denote a password of x characters. Function ψ repeats the following procedure for $i = 1$ to x: first create a substring of the password from the first character to the ith character (inclusive), then count how many times the ith character appears in this substring, then append the counter to the ith character and put the result into a set. For example, "pa$$w0rd" will be converted by ψ into $\{p1, a1, \$1, \$2, w1, 01, r1, d1\}$.

The alphabet pre-processing step is necessary because we use SPC to check whether a password satisfies a policy. The SPC protocol in section 4.1 is based on set intersection. So we need to intersect the password set and the alphabet set in order to check the password policy. The password set now contains indexed characters like "p1","$2" rather than the original characters. The alphabet needs to be converted into a set with indexed characters as well, otherwise the intersection of the password set and alphabet set will always be empty. This step is done by the server as follows. Let $A = A_1 \cup \cdots \cup A_m$ be the alphabet where A_i is a character class (digits, lower case, etc.). The server transforms it into $\mathcal{S}$ based on the password policy P. For each A_i, in the policy there is a threshold t_i that says at least t_i characters from A_i need to appear in the password. If $t_i = 0$, then the server just skips all characters in this class A_i because they do not have to be checked. Otherwise, the server creates an empty set $\mathcal{S}_i$ and appends an index (from 1 to t_i) to each character in A_i, and puts the t_i copies of indexed characters into S_i. For example, if A_i contains lower case characters and $t_i = 2$, then $\mathcal{S}_i = \{a1, a2, b1, b2, \ldots, z1, z2\}$. The server only needs to put t_i copies of indexed characters to the set, regardless how many times the character may appears in clients' passwords. The union of $\mathcal{S}_i$ is the set $\mathcal{S}$ to be used later in password registration. This step only needs to be done once as long as the policy does not change.

5.2 The Password Registration Protocol

An overview of the proposed password registration protocol is given in Fig. 2. To simplify the presentation, we assume the protocol is run over a secure channel, e.g., implemented as a server authenticated TLS channel. The secure channel will address common network-based attacks such as replay, eavesdropping and man-in-the-middle. We also assume there is a session mechanism to prevent the server from learning more information about the client's password by aborting the protocol in the last step and reruning the protocol using other policies. The protocol has two phases, a setup phase and a policy checking phase. In the setup phase the client commits to its password, and each party blinds its set. The blinded sets are later used in the policy checking phase where the server checks the password policy with a secure SPC protocol (cf. Section 4) using the blinded sets. If the password satisfies the policy, the server stores a password verifier for future authentication purposes.

Public input The server publishes a password policy $P = (\mathcal{S}, \Gamma_{\mathcal{S}})$ where $\mathcal{S}$ is a set of size w transformed from alphabet A according to Section 5.1 and $\Gamma_{\mathcal{S}}$ is a threshold access structure defined over $\mathcal{S}$. Other public parameters consist of a security parameter λ, a pseudorandom function family f_k, and three hash functions H_1, H_2, and H_3.

- **Setup Phase**
  1. The server runs $\texttt{KeyGen}(\lambda)$: pick two large equal length prime numbers p and q according to λ, compute $N = p \cdot q$, choose at uniformly random $e \in Z_N$ such that there is an integer d that satisfies $e \cdot d = 1 \bmod \phi(N)$, and output (e, d, N). Then the server sends (e, N) to the client.
  2. The client computes a key $k = H_1(\texttt{pwd})$ where $\texttt{pwd}$ is its password. The client uses the password pre-processing function ψ to generate $\mathcal{C} \leftarrow \psi(\texttt{pwd})$. The client computes $r_i = f_k(i)$ using the pseudorandom function f on key k as well as $u_i = H_2(c_i) \cdot r_i^e$ for each $c_i \in \mathcal{C}$. The result $(u_1, \ldots, u_v)$ is sent to the server, where v is the cardinality of $\mathcal{C}$.
  3. For each $i \in [1, v]$ the server computes $u'_i = u_i^d$ and returns $(u'_1, \ldots, u'_v)$ back to the client.
  4. Upon receiving $(u'_1, \ldots, u'_v)$, the client creates an empty set $\hat{\mathcal{C}}$ and for $i \in [1, v]$ puts $u'_i \cdot r_i^{-1} = (H_2(c_i))^d$ into $\hat{\mathcal{C}}$. The server creates an empty set $\hat{\mathcal{S}}$ and for $i \in [1, w]$ puts $(H_2(s_i))^d$ into $\hat{\mathcal{S}}$, where $s_i \in \mathcal{S}$ and w is the cardinality of $\mathcal{S}$. The set $\hat{\mathcal{S}}$ is partitioned into m subsets according to the character classes. The server also generates $\hat{P}$ from P by replacing $\mathcal{S}$ with $\hat{\mathcal{S}}$.
- **Policy Checking Phase** This phase is essentially an execution of the SPC protocol using $\hat{\mathcal{C}}$ and $\hat{P}$ as inputs. At the end of the SPC protocol the server learns whether the client's password satisfies the policy or not. If the SPC execution is successful, the server computes the hash of the product of the client's u_i values $h \leftarrow H_3(\prod_{i=1}^{v} u_i)$, and stores the password verifier $\texttt{ver} = (h, e, N, d, \boldsymbol{u})$, where (e, N, d) is generated in the first step of the setup phase and $\boldsymbol{u} = \{u_1, \ldots, u_v\}$ is the vector of client "commitments".

Note that in the first step of the setup phase, $\texttt{KeyGen}$ is essentially the RSA key generation algorithm with e chosen randomly rather than being a fixed small integer. In the protocol we use e as a salt so the verifiers will be different even if two users chooses the same password. Salting is also necessary in order to avoid rainbow table attacks where the attacker uses pre-computed values to speed up dictionary attacks.

Password Length Hiding (Enhanced Protocol) The protocol in Figure 2 leaks the password length to the server. By counting the number of blinded characters u_i, the server learns the password length v. This is intentional because this peripheral information leakage allows the server to efficiently

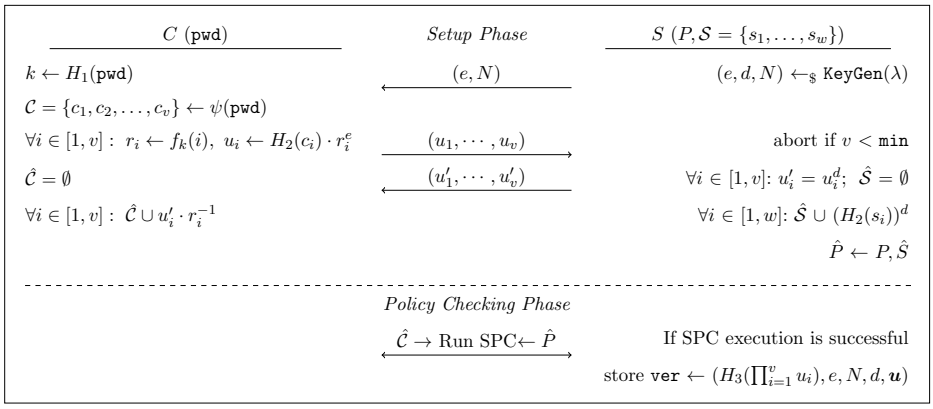

Fig. 2: Password Registration using secure SPC

enforce the minimal password length in the policy. However, in cases where the password length is considered sensitive, it can be hidden from the server at small additional cost.

The client generates a set $\mathcal{C}' \subseteq \mathcal{C}$ and uses it in the setup phase to generate $\hat{\mathcal{C}}$. $\mathcal{C}'$ contains only necessary characters to fulfil P. That is, the client first takes characters from $\mathcal{C}$ according to character class A_i and threshold t_i, and puts them in $\mathcal{C}'$. If the size of $\mathcal{C}'$ is smaller than the minimal password length `min`, the client pads it with other characters in $\mathcal{C}$ that are not in $\mathcal{C}'$ yet. In the setup phase, the client only uses characters in $\mathcal{C}'$ and obtains the corresponding $\hat{\mathcal{C}}$. In this process, the server learns the size of $\mathcal{C}'$ and can check whether this is equal to the minimal password length required by the policy. The client then uses this $\hat{\mathcal{C}}$ in the policy checking phase to convince the server about the password complexity. If the server accepts, all characters in $\mathcal{C} \setminus \mathcal{C}'$ that have not been sent to the server are put into an additional $u^* = r_{v+1}^e \cdot \prod u_i$ with $r_i \leftarrow f_k(i),\ u_i \leftarrow H_2(c_i) \cdot r_i^e$ for $c_i \in \mathcal{C} \setminus \mathcal{C}'$. This value u^* is then sent to the server and is multiplied with the other u_i values the server received in the setup phase. This product is then used to generate the verifier `ver`, i.e. $\texttt{ver} \leftarrow (H_3(r_{v+1}^e \prod_{i=1}^{v} u_i), e, N, d, \boldsymbol{u})$. Note that we require r_{v+1}^e as a multiplicand when computing u^*. Without this, the server could learn the client's password length when $\mathcal{C} \setminus \mathcal{C}' = \emptyset$ because the client would have nothing to send in this case.

5.3 Security Analysis

We now analyse the security of the password registration protocol. Note that in the password registration protocol, the two parties have different security requirements. For the server, privacy is not a concern since its input, the policy, is public. On the other hand, the server cares about the soundness of the protocol because an unsound protocol would allow a user to register an invalid password. For the client, privacy is the main concern. Soundness is trivial from the client's point of view. Since the policy is public, the client can check the policy by itself and can detect if the server cheats. We therefore refrain from using an over-complicated security definition and use the following comprehensible security model that is simpler. Let $\texttt{ver} \leftarrow \phi(\texttt{pwd}, r)$ denote a password verifier, computed from a password `pwd` and some randomness r, and $\psi(\texttt{pwd})$ a function to generate set $\mathcal{C}$ from password `pwd`.

1. Privacy: A malicious server must not be able to retrieve more information from the protocol than the password verifier and the result of the policy verification. Furthermore, the verifier must not give a malicious server advantage in terms of password guessing.

2. Soundness: The server accepts a password verifier $\texttt{ver} \leftarrow \phi(\texttt{pwd}, r)$ if and only if (i) the password is compliant with the server's policy, i.e. $P(\mathcal{C}) = \texttt{true}$ for $\mathcal{C} \leftarrow \psi(\texttt{pwd})$, and (ii) the verifier is uniquely defined by the password and some server known randomness, i.e. there exists no password $\texttt{pwd}' \neq \texttt{pwd}$ such that $\phi(\texttt{pwd}', r) = \texttt{ver}$ and it is not possible to find randomness $r' \neq r$ in polynomial time such that $\phi(\texttt{pwd}, r') = \texttt{ver}$.

Note that the strength of the privacy definition is in terms of dictionary attack resistance. This is an inherent problem of password-based protocols. All password-based protocols are susceptible to dictionary attacks if the server is considered as a potential adversary [18]. The reason is simple: for authentication purpose, the server holds a verifier derived from the client's password. An authentication protocol essentially takes the user's password as an input and compares it securely with the verifier. A malicious server can always run the protocol locally with itself playing the client's role using passwords enumerated from a dictionary. Since it is not realistic to assume any particular distribution of passwords, e.g. uniformly at random chosen passwords, the worst case security always depends on the hardness of dictionary attack and this is the strongest privacy notion possible. We will discuss what can be used to counter dictionary attack later in Section 5.4.

In the following we show that the enhanced version of the previously defined protocol satisfies those properties. Note that the simple version satisfies the same properties but in a weaker version, i.e. we would have to replace dictionary $\mathcal{D}_P$ in Lemma 4 with $\mathcal{D}_{P,|\texttt{pwd}|}$, where $\mathcal{D}_{P,|\texttt{pwd}|}$ denotes the dictionary that contains all passwords of size $|\texttt{pwd}|$ that are policy compliant with respect to P. Note that H_2 has to be modelled as random oracle here in order to use the one-more RSA assumption [3]. For the other two hash functions H_1 and H_3 it is sufficient to assume collision resistance. Due to space limitations we refer to the full version for proofs.

Lemma 4 (Privacy). *If f_k is a secure pseudorandom function family, H_1 is collision resistant, and H_2 a random oracle, the enhanced password registration protocol offers privacy with respect to a malicious server and dictionary $\mathcal{D}_P$, which contains all valid passwords with regard to the server policy.*

Lemma 5 (Soundness). *The enhanced password registration protocol is sound with respect to a malicious client under the one-more RSA assumption if H_1 and H_3 are collision resistant hash functions, and H_2 a random oracle.*

5.4 Password-authenticated Key Exchange for our Protocol

In order to use a password registered with our protocol for authentication, we require an appropriate password-based authentication or authenticated key exchange (PAKE) protocol. In this section we show how to use the verifier $\texttt{ver}$ in a common PAKE protocol. The approach we describe here is general and can be used with any PAKE protocol.

At the beginning of the authentication process, for a given client identifier the server retrieves the corresponding verifier $\texttt{ver} = (h, e, N)$ from the database and returns (e, N) to the client. Using (e, N) and the password $\texttt{pwd}$, the client can recompute all u_i values and thus $h' \leftarrow H_3(u_{v+1}^e \cdot \prod_{i=1}^{v} u_i)$ as described earlier. Note that depending on the used PAKE protocol we have to ensure that H_3 maps into an algebraic structure, suitable for use with the PAKE protocol. Now client and server run any PAKE protocol on password hash h. The password hash h retains information about individual characters as well as the order of characters in the password. The first is easy to see since h is computed from the product of blinded characters in the password. To see the second,

	$(P1, 20)$		$(P2, 20)$		$(P3, 20)$		$(P2, 10)$		$(P2, 40)$	
	Total	Pol-ck	Total	Pol-ck	Total	Pol-ck	Total	Pol-ck	Total	Pol-ck
ZKPPC [16]	81,287	81,268	66,944	66,925	38,496	38,477	7,710	7,699	453,574	453,529
Our Protocol	140	4	243	8	454	17	223	7	275	8
Improvement	580×		275×		80×		35×		1649×	

Table 1: Protocol Performance (Running Time in Milliseconds)

recall that each $u_i = H_2(c_i) \cdot r_i^e$ where $r_i = f_k(i)$, which is a pseudorandom number generated under a key k. The key k is derived from the password string $k \leftarrow H_1(\texttt{pwd})$. To counter offline dictionary attack, we can use a standard key derivation functions such as PBKDF2 [15] to compute H_1 such that the key k is derived by repeatedly applying a pseudorandom function with a salt. The verifier generation algorithm also provides additional protection to offline dictionary attack. The computation involves a large random e as a salt and requires slow public key operations. The additional salt and work load make offline dictionary attack even more difficult.

Because of the way the verifier is structured, in the authentication the server needs to send an additional message, the RSA public key (e, N), to the client. Often we can piggyback the messages in the PAKE protocol to avoid increasing communication round. For example, if we use the UC-secure PAKE protocol from Benhamouda et al. [4], the RSA public key (e, N) can be piggybacked on the server's message sent in the PAKE protocol. Thus we do not increase the round complexity and the protocol remains a one-round protocol.

6 Implementation and Evaluation

We implemented a prototype of our password registration protocol and measured the performance. To compare, we also implemented the password registration protocol (ZKPPC) proposed in [16]. Both implementations are in C and use OpenSSL 1.0.0 (`https://www.openssl.org`) for the underlying cryptographic operations. In the experiments, we set the security parameter to 80-bit. We used 1024-bit RSA keys and the SHA-1 hash function in our protocol. In the ZKPPC protocol we use the NIST P-192 elliptic curve. All experiments were run on a MacPro desktop with 2 Intel E5645 2.4GHz CPUs and 32 GB RAM. Note in the experiments our implemenation only uses one CPU core and less than 1 GB RAM.

The running time of the protocols are shown in Table 1. We measured the running time with different policies and password lengths. The passwords are printable ASCII strings. The alphabet is partitioned into 4 classes: digits, lower case, upper case and symbols. We used three policies $P1, P2$ and $P3$ in the experiments, which require at least one, two and four characters in all character classes respectively. In the first row of the table, the pairs indicate the policy and the password length that were used in the experiment, e.g. $(P1, 20)$ means policy $P1$ is used and the password was 20 characters long. The table shows the total running time as well as the time spent on checking the policies (Pol-ck) in the protocol. As we can see, the performance of our protocol is much better than the ZKPPC protocol. The main difference comes from policy checking time. Policy checking in ZKPPC is done by using a zero-knowledge proof of set membership protocol. The cost of the zero-knowledge proof protocol is $6 \cdot n \cdot \sum_{i=1}^{n} \omega_i$ exponentiations, where n is the password length in the experiments, and ω_i is the size of character class to which the ith character in the password belongs. In our protocol, policy checking is done by using the SPC protocol and the cost is mainly the OBI protocol which is based on symmetric cryptography. The cost of the OBI protocol is $4.32 \cdot |\hat{S}| \cdot \lambda$ hash operations, where λ is the security parameter. More concretely, in setting $(P1, 20)$,

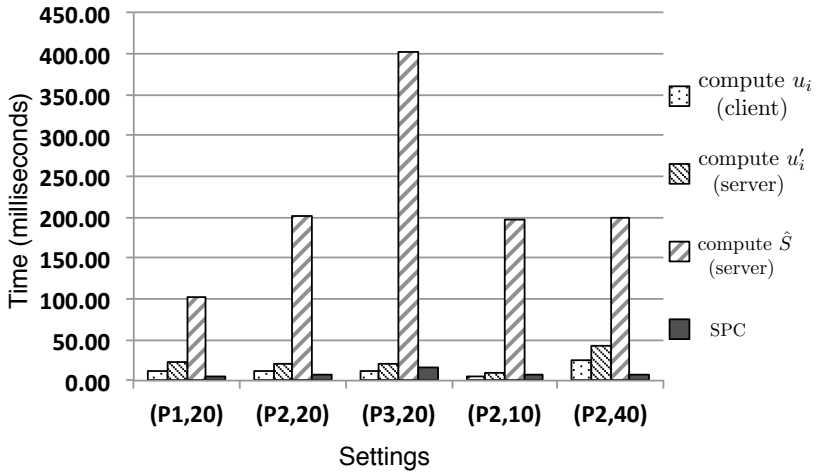

Fig. 3: Time Breakdown
– $(P1, 20)$ means policy $P1$ is used and the password was 20 characters long
– Policies $P1, P2$, and $P3$ require at least one, two and four characters in all classes

the zero-knowledge proof based policy checking requires around 200,000 exponentiations while our OBI based SPC requires only less than 33,000 hash operations.

We also show the running time for each step in our protocol (see Fig. 3). As we can see in the figure, the time for computing u_i and u_i' is linear in the password length, and the time for computing $\hat{\mathcal{S}}$ and executing SPC is linear in the size of $\hat{\mathcal{S}}$. The most costly step is in the setup phase when the server computes the encrypted version of the alphabet $\hat{S}$. A possible optimisation is to take this step offline. Since the computation of $\hat{S}$ does not depends on the client's password, the server can generate a random RSA key pair and pre-compute $\hat{S}$ before engaging with the client. The keys and pre-computed values can be stored together. Later when a client sends a registration request, the server can retrieve them and run the protocol. If this step is taken offline, then the online computation cost is small, usually no more than 100 milliseconds in a typical setting.

7 SPC Applications

SPC can be used in many different scenarios. In the previous section we gave a detailed example of using SPC for password-policy checking on password registration. In this section we describe other use-cases of the primitive.

Policy checks for Access Control In a role-based access control scenario [10] a user has to have a certain role in order to access a resource. In complex organisational structures it may be necessary to have a certain *combination* of roles in order to access a resource rather than just a single role. SPC can be used in this case to verify whether a client has necessary roles that allow it to access the resource. The server set $\mathcal{S}$ in this case contains secrets associated with each role S_i and the user's set $\mathcal{C}$ contains the client's secrets c_i. Access should be granted if and only if the SPC protocol is successful, i.e. the user can convince the server that he has all necessary roles.

Policies for Friendship Analysis One popular application of set based protocols is friendship analysis. This test should determine whether two parties become friends or not depending on the number of mutual friends. SPC can be used in this scenario as a very efficient alternative while increasing privacy. Using SPC further allows to build subsets in friend sets, such as colleagues, family etc.,

which in turn makes the friendship-test more "accurate" while leaking as little information about the friendship relations as possible.
Genome Testing Baldi et al. [1] propose protocols to perform privacy preserving genome testing, such as paternity tests. The tests can often be reduced to check a set of SNPs (Single Nucleotide Polymorphism) that are present in a patient's genome against some predefined sets of SNPs. Although it is not exactly policy checking, our SPC protocol can be used in this setting too.

8 Conclusion and Future Work

In this work we introduced a new notion called set-based policy checking (SPC), a new privacy preserving protocol. SPC allows a server to check whether a set held by a client is compliant with its policy, which is defined as a monotone access structure. At the end of the protocol, the server learns only a single bit of information, i.e. whether the client's set complies with the policy or not, and nothing else. We showcase the use of SPC in a new, highly efficient protocol for password registration that allows the server to impose a password policy on the client's password. To underline practicality and facilitate adoption we gave an efficient implementation of the password registration protocol together with an analysis. We further sketched other application scenarios of SPC.

Currently SPC is designed for public policies where the server makes its policy public to the client. Although in most real world policy-based systems, the policies are not considered private information, there are some scenarios in which the server may want to keep its policy private. As future work, we will investigate policy checking with hidden policies.

References

1. P. Baldi, R. Baronio, E. D. Cristofaro, P. Gasti, and G. Tsudik. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In *CCS'11*, pages 691–702. ACM, 2011.
2. A. Beimel. *Secure schemes for secret sharing and key distribution.* PhD thesis, Technion-Israel Institute of technology, Faculty of computer science, 1996.
3. M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. *J. Cryptology*, 16(3):185–215, 2003.
4. F. Benhamouda, O. Blazy, C. Chevalier, D. Pointcheval, and D. Vergnaud. New techniques for sphfs and efficient one-round pake protocols. In *CRYPTO'13*, volume 8042 of *LNCS*, pages 449–475. Springer-Verlag, 2013.
5. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
6. Dan Goodin. Hack of Cupid Media dating website exposes 42 million plaintext passwords. `http://goo.gl/sLcx4Y`, 2014. Accessed: 01/04/2015.
7. A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The Tangled Web of Password Reuse. In *NDSS*. The Internet Society, 2014.
8. C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: an efficient and scalable protocol. In *ACM Conference on Computer and Communications Security*, pages 789–800, 2013.
9. C. Duma, A. Herzog, and N. Shahmehri. Privacy in the Semantic Web: What Policy Languages Have to Offer. In *8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLIC) 2007*, pages 109–118, 2007.
10. D. F. Ferraiolo and D. R. Kuhn. Role-based access controls. *CoRR*, abs/0903.2171, 2009.
11. D. A. F. Florêncio and C. Herley. A large-scale study of web password habits. In *16th International Conference on World Wide Web, WWW 2007*, pages 657–666. ACM, 2007.
12. M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 1–19, 2004.
13. S. Gaw and E. W. Felten. Password management strategies for online accounts. In *SOUPS'06*, volume 149 of *ACM International Conference Proceeding Series*, pages 44–55. ACM, 2006.

14. M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.
15. B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), Sept. 2000.
16. F. Kiefer and M. Manulis. Zero-Knowledge Password Policy Checks and Verifier-Based PAKE. In *ESORICS'14*, volume 8713 of *LNCS*, pages 295–312. Springer, 2014.
17. J. Li, N. Li, and W. H. Winsborough. Automated trust negotiation using cryptographic credentials. In *CCS'05*, pages 46–57. ACM, 2005.
18. A. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
19. M. Nabeel and E. Bertino. Privacy-Preserving Fine-Grained Access Control in Public Clouds. *IEEE Data Eng. Bull.*, 35(4):21–30, 2012.
20. Nik Cubrilovic. RockYou Hack: From Bad To Worse. `http://goo.gl/u91YHV`, 2014. Accessed: 01/04/2015.
21. B. Pinkas, T. Schneider, and M. Zohner. Faster Private Set Intersection Based on OT Extension. In *USENIX'14*, pages 797–812, 2014.
22. Reuters. Trove of Adobe user data found on Web after breach: security firm. `http://goo.gl/cpZn6B`, 2014. Accessed: 01/04/2015.
23. X. Suo, Y. Zhu, and G. S. Owen. Graphical Passwords: A Survey. In *ACSAC'05*, pages 463–472, 2005.
24. Z. Wen and C. Dong. Efficient protocols for private record linkage. In *SAC'14*, pages 1688–1694. ACM, 2014.
25. W. H. Winsborough and N. Li. Towards practical automated trust negotiation. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002), 5-7 June 2002, Monterey, CA, USA*, pages 92–103, 2002.
26. A. C. Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, 1982.

A Definitions

One-more RSA assumption The one-more RSA assumption [3] indicates that the RSA problem is hard even if the adversary is given access to an RSA oracle. Formally, let $(N, e, d) \leftarrow_{\$} \texttt{KeyGen}(\lambda)$ denote the RSA key-generation algorithm, and $r_j \leftarrow_{\$} \mathbb{Z}_N^*$ be uniformly random integers in $\mathbb{Z}_N^*$ for $j \in [1, t+1]$. We say the one-more RSA problem is (λ, t)-hard on security parameter λ if for every probabilistic polynomial time adversary $\mathcal{A}$ we have

$$\Pr[\{x_i\}_{i\in[1,t+1]} \leftarrow \mathcal{A}^{(\cdot)^{d} \bmod N}(N, e, \lambda, r_1, \ldots, r_{t+1})] \leq \varepsilon(\lambda),$$

where $x_i^e = r_i \bmod N$, $\mathcal{A}$ made at most t queries to the RSA oracle $(\cdot)^{d} \bmod N$ and $\varepsilon(\cdot)$ is a negligible function.

Collision resistant hash functions Cryptographic hash functions are collision resistant if an attacker, given access to hash function H, has negligible probability of generating two inputs m_1 and m_2 $(m_1 \neq m_2)$ such that $H(m_1) = H(m_2)$.

B Security Proofs

B.1 Security Proofs of the SPC Protocol

Proof of Lemma 1 Assuming correct LSSS and OBI, correctness can be proved by the following argument. LSSS guarantees correct sharing of secret $\mathfrak{s}$, i.e. for any authorised set $S_i \in \Gamma_{\mathcal{S}}$ there is a set of shares associated with the elements in S_i that can reconstruct $\mathfrak{s}$ and for any other set the secret cannot be reconstructed. OBI guarantees that for each element in the intersection of $\mathcal{S} \cap \mathcal{C}$, the client is able to obtain the share associated with it. All elements in $\mathcal{C}$ that are not in the

intersection are irrelevant as by definition they are not in $\mathcal{S}$ thus not in any authorised sets. Then the client can reconstruct $\mathfrak{s}$ iff there exists a subset of $\mathcal{S} \cap \mathcal{C}$ that is an authorised set. If there is such an authorised set then $P(\mathcal{C}) = \texttt{true}$. Eventually, the hash value $m_{\mathfrak{s}}$ is the same on the client and server if the same key $\mathfrak{s}$, transcript $\texttt{trans}$, and participants identifiers C and S are used. □

Proof of Lemma 2 For the security proof we consider an ideal OBI function $f_{\texttt{OBI}}$, i.e. show computational indistinguishability between the world $\text{HYBRID}^{f_{\texttt{OBI}}}_{\Pi,A(z)}(x,y)$ and ideal world $\text{IDEAL}_{f_{\texttt{SPC}},B(z)}(x,y)$ to prove security. First, since we are in the $\text{HYBRID}^{f_{\texttt{OBI}}}$ world, every $\texttt{OBI}$ operation and message is forwarded to the $\texttt{OBI}$ functionality. Since we only require LSSS correctness here, we do not use the ideal functionality. In the following we show that the hybrid world $\text{HYBRID}^{f_{\texttt{OBI}}}_{\Pi,A(z)}$ is computationally indistinguishable from $\text{IDEAL}_{f_{\texttt{SPC}},B(z)}$ and hence the real world $\text{REAL}_{\Pi,A(z)}$. We give a simulator $\texttt{SIM}_S(z)$ that simulates a malicious server in the ideal world. Note that we omit auxiliary input z if not needed. We build $\texttt{SIM}_S$ that on input of the server's policy P, and access to the real world adversary A_S that plays server S, generates $\texttt{view}_S$ that is indistinguishable from $\texttt{view}_{AS}$ of A_S.

$\texttt{SIM}_S$ starts by invoking server A_S with P and z to receive share-generating matrix M and ρ from adversary A_S in the first protocol message. Then as input to the OBI functionality $\texttt{SIM}_S$ receives the server's OBI input from A_S, i.e. a set $\mathcal{S}'$ and shares $\mathcal{S}_d = \{\mathfrak{s}_i\}$. $\texttt{SIM}_S$ then sends $\mathcal{S}'$ to the trusted party. If the trusted party replies with $\perp$, the simulation terminates and $\texttt{SIM}_S$ outputs whatever A_S outputs. If the trusted party replies with $\texttt{true}$, simulator $\texttt{SIM}_S$ generates a set $\mathcal{C}$ with $P(\mathcal{C}) = \texttt{true}$ based on P and $\mathcal{S}'$. If the trusted party replies with $\texttt{false}$, simulator $\texttt{SIM}_S$ generates a set $\mathcal{C}$ with $P(\mathcal{C}) = \texttt{false}$ based on P and $\mathcal{S}'$. Building $\mathcal{C}$ from $\mathcal{S}'$ and P is straightforward. $\texttt{SIM}_S$ sends client and server input to the OBI functionality to retrieve the server's and client's view on the OBI execution. Further, $\texttt{SIM}_S$ tries to recover $\mathfrak{s}$ from the output of $f_{\texttt{OBI}}$ using the combination algorithm of LSSS. If the client can reconstruct $\mathfrak{s}$, $\texttt{SIM}_S$ generates $m_{\mathfrak{s}}$ using M, ρ, P and $\mathcal{S}'$, the (simulated) OBI transcript, $\mathfrak{s}$, C, and S and sends it to A_S. Otherwise, if $\texttt{SIM}_S$ cannot recover $\mathfrak{s}$ from the $f_{\texttt{OBI}}$ output, it terminates the session with $\perp$. Eventually $\texttt{SIM}_S$ outputs the transcript and whatever A_S returns on terminating as $\texttt{view}_S$.

We claim that the output of an honest client in the ideal execution is indistinguishable from the client's output in the real world. This is easy to see as the client always receives P and the possibility to compute $P(\mathcal{C})$ using either the public server set $\mathcal{S}$, or the output of the OBI/$f_{\texttt{OBI}}$ execution. Note that the evaluation of P depends on the set used by A_S in the OBI execution ($\mathcal{S}'$). Since this strategy is the same in the real and the ideal world, the claim follows. Indistinguishability of $\texttt{view}_S$ and $\texttt{view}_{AS}$ follows from the following observations. If the server's input to OBI is not correct, the protocol terminates. It is easy to see that the transcript containing P, M, ρ, and $\mathcal{S}$, and the OBI execution in $\texttt{view}_S$ and $\texttt{view}_{AS}$ is identical. Since the input to the hash function H is equivalent in both worlds, $\texttt{view}_S$ and $\texttt{view}_{AS}$ are indistinguishable. □

Proof sketch of Lemma 3 We prove security of Lemma 3 by showing that convincing S to accept the protocol despite the fact that $P(\mathcal{C}) \neq \texttt{true}$ implies a collision in H. First, it is clear that the attacker is not able to recover the correct secret $\mathfrak{s}$ from the OBI interaction with S since this would break either OBI or LSSS security. Now it follows directly that any attacker that is able to generate a message $m_{\mathfrak{s}}$ such that $m_{\mathfrak{s}} = H(\mathfrak{s}, C, S, \texttt{trans})$ found a collision in H as it does not know the correct secret $\mathfrak{s}$. □

B.2 Security Proofs of the Password Registration Protocol

Proof of Lemma 4 We first show that the server is simulatable and, i.e. the protocol realises the functionality f_{Π}, and then show that the verifier $\texttt{ver}$ is of no help when performing a dictionary attack.

We start with the simulation by building a simulator $\texttt{SIM}_S$, simulating a malicious server. $\texttt{SIM}_S$ starts by invoking adversary A_S with $(P, \mathcal{S})$ that is playing the role of server S in the protocol, and is provided with (e, N) as a result. Using (e, N), $\texttt{SIM}_S$ generates $(u_1, \ldots, u_l)$ as $u_i = \alpha_i r_i^e$ for $\alpha_i \leftarrow_\$ \mathbb{Z}_N^*$, $r_i \leftarrow f_{H_1(\texttt{pwd})}(i)$, and some $l = \texttt{min}$ where $\texttt{pwd} \in \mathcal{D}_P$, and returns it to A_S. The random oracle H_2 is honestly simulated by $\texttt{SIM}_S$. After receiving $(u'_1, \ldots, u'_l)$ the simulator builds $\hat{\mathcal{C}}$ according to protocol specification and uses it together with $\hat{P}$, which is provided by A_S, to simulate the SPC execution. Eventually, $\texttt{SIM}_S$ gives the result of the SPC execution as well as $u^* \leftarrow_\$ \mathbb{Z}_N^*$ to A_S and outputs whatever A_S returns on termination. It is easy to see that the client's view after the protocol is identical in the real and ideal world as the protocol execution is correct and all server values despite d are public. Further, the adversary's view is computationally indistinguishable from the simulator's output since all client messages have the same distribution in both worlds.

To see that a malicious server is not able to use the values retrieved in the protocol to perform a dictionary attack over $\mathcal{D}_P$ more efficient than without executing the protocol, i.e. we show that the adversary is not able to perform an attack on the retrieved elements $u_i, i \in [1, l]$ and u^* that contain information about the password, which is faster than a dictionary attack over $\mathcal{D}_P$. Since the client is essentially creating blind RSA signatures on $H_2(c_i)$ in u_i, those values are indistinguishable from random elements. While this would be true in the statistical sense if r_i would be chosen uniformly at random, this is not the case here. However, it is easy to see that in order to verify an element u_i, the server has to compute r_i, which either requires an offline dictionary attack on $\mathcal{D}_P$ to compute $k \leftarrow H_1(\texttt{pwd})$, or yields either a collision in H_1 or breaks pseudorandomness in f_k. Therefore, the fastest way for a server to retrieve the password is to perform an offline dictionary attack on $\mathcal{D}_P$. □

Proof of Lemma 5 First note that the used SPC protocol is secure and therefore guarantees that the server accepts iff the elements in $\hat{\mathcal{C}}$ are compliant with policy P with respect to $\hat{\mathcal{S}}$. We therefore only have to show that (i) the client is not able to use different elements in $\mathcal{C}$ than in $\hat{\mathcal{C}}$, i.e. the password $\texttt{pwd}$ actually satisfies P, and (ii) the password verifier $\texttt{ver}$ is uniquely defined by $(\texttt{pwd}, e, N, d, \boldsymbol{u})$, i.e. there exists no password $\texttt{pwd}' \neq \texttt{pwd}$ that generates the same verifier as $\texttt{pwd}$ and it is not possible to find randomness $(e', N', d') \neq (e, N, d)$ in polynomial time that generates the same verifier as (e, N, d).

(i) We claim that the mapping from $\mathcal{C}$ to $\hat{\mathcal{C}}$ is an injective function such that the client is not able to build $\hat{\mathcal{C}}$ from a password $\texttt{pwd}' \neq \texttt{pwd}$. The elements in $\hat{\mathcal{C}}$ have the form $\hat{c}_i = (H_2(c_i))^d$ where d is the server's secret RSA key. If the attacker is able to generate $\hat{c}_i = (H_2(c_i))^d$ from $c_i \notin \mathcal{C}$, we can use it to build a successful attacker on the one-more RSA assumption.

(ii) We claim that the password verifier $\texttt{ver} = (H_3(r_{v+1}^e \prod_{i=1}^v u_i), e, N, d, \boldsymbol{u})$ is uniquely identified by $(\texttt{pwd}, e, N, d, \boldsymbol{u})$. In this case we assume the client chose a policy compliant password $\texttt{pwd}$ and performed the protocol honestly. The claim is easy to see since $H_3(r_{v+1}^e \prod_{i=1}^v u_i) = H_3(f_{H_1(\texttt{pwd})}(v+1)^e \cdot \prod_{i=1}^v (H_2(c_i) \cdot f_{H_1(\texttt{pwd})}(i)))$. We can in particular either find collisions in H_1 or H_3, or distinguish between f_k and a random function. □