УДК 004.056.5

РАСШИРЕНИЕ ФУНКЦИОНАЛЬНОСТИ СЛЕПЫХ АККУМУЛЯТОРОВ: КОНТЕКСТЫ

С. В. АГИЕВИЧ[1]***, М. А. КАЗЛОВСКИЙ***[2]

*[1]Научно-исследовательский институт прикладных проблем математики и информатики БГУ,
пр. Независимости, 4, 220030, г. Минск, Беларусь*
[2]Белорусский государственный университет, пр. Независимости, 4, 220030, г. Минск, Беларусь

Аннотация. Слепой аккумулятор предназначен для децентрализованной загрузки авторизованными сторонами своих личных ключей с последующей выгрузкой открытых ключей. Выгружаемый открытый ключ связан с одной из сторон, хотя и неизвестно с какой. Схема слепого аккумулятора расширяется так, чтобы открытый ключ стороны был привязан к определенному контексту и этот ключ было вычислительно трудно связать с открытыми ключами той же стороны, полученными в других контекстах. Слепые аккумуляторы с контекстами оказываются полезными в различных сценариях электронного голосования, например при переголосовании. Предлагается реализация схемы слепого аккумулятора с контекстами, и обосновывается ее безопасность.

Ключевые слова: электронное голосование; переголосование; криптографический аккумулятор; слепой аккумулятор; распознавательная задача Диффи – Хеллмана.

Благодарность. Авторы выражают признательность анонимным рецензентам за ценные отзывы, которые помогли улучшить редакционное и техническое качество статьи.

Образец цитирования:
Агиевич СВ, Казловский МА. Расширение функциональности слепых аккумуляторов: контексты. *Журнал Белорусского государственного университета. Математика. Информатика.* 2024;1:79–85 (на англ.).
EDN: KMXHJP

For citation:
Agievich SV, Kazlouski MA. Extending the functionality of blind accumulators: contexts. *Journal of the Belarusian State University. Mathematics and Informatics.* 2024;1:79–85.
EDN: KMXHJP

Авторы:
Сергей Валерьевич Агиевич – кандидат физико-математических наук; заведующий научно-исследовательской лабораторией проблем безопасности информационных технологий.
Максим Анатольевич Казловский – аспирант кафедры математического моделирования и анализа данных факультета прикладной математики и информатики. Научный руководитель – С. В. Агиевич.

Authors:
Sergey V. Agievich, PhD (physics and mathematics); head of the IT security research laboratory.
agievich@bsu.by
https://orcid.org/0000-0002-9413-8574
Maksim A. Kazlouski, postgraduate student at the department of mathematical modelling and data analysis, faculty of applied mathematics and computer science.
kazlouskima@bsu.by
https://orcid.org/0009-0004-4908-2841

EXTENDING THE FUNCTIONALITY OF BLIND ACCUMULATORS: CONTEXTS

S. V. AGIEVICH[a]***, M. A. KAZLOUSKI***[b]

[a]*Research Institute for Applied Problems of Mathematics and Informatics, Belarusian State University,*
4 Niezaliezhnasci Avenue, Minsk 220030, Belarus
[b]*Belarusian State University, 4 Niezaliezhnasci Avenue, Minsk 220030, Belarus*

Corresponding author: S. V. Agievich (agievich@bsu.by)

Abstract. Blind accumulators collect private keys of eligible entities in a decentralised manner not getting information about the keys. Once the accumulation is complete, an entity processes the resulting accumulator and derives a public key which refers to a private key previously added by this entity. We extend the blind accumulator scheme with the context functionality so that the derived key is bound to a specific context and this key is computationally hard to associate with public keys of other contexts. Blind accumulators with contexts are useful in various e-voting scenarios, for example in revoting. We provide an instantiation of the extended blind accumulator scheme and justify its security.

Keywords: e-voting; revoting; cryptographic accumulator; blind accumulator; decisional Diffie – Hellman problem.

Acknowledgements. The authors thank the anonymous referees for their valuable feedback that helped improve the editorial and technical quality of the paper.

Introduction

A blind accumulator is a cryptographic container that collects private keys and outputs (derives) the corresponding public keys. A private key is added to the accumulator in a provably correct manner while remaining secret, that is, known only to its owner. The derived public key is accompanied by a proof that it refers to some of the collected private keys while it is computationally hard to determine which one. In addition, blind accumulators are managed in the decentralised manner.

Blind accumulators are introduced in [1] as a tool for organising decentralised electronic voting (e-voting). Voters use the accumulated private keys to sign ballots, the derived public keys are used to verify signatures. The voter's public key acts as an immutable pseudonym, which can be used to prevent double voting or, conversely, allow multiple ballots to be cast with only the last one to be counted.

The last possibility, the so-called revoting, is an important feature of modern e-voting systems. The direct revoting based on blind accumulators have the following drawback: an adversary can track the change in the opinions of voters (while not violating their anonymity) by observing ballots. It is desirable to organise revoting in such a way that it is difficult to relate the original and subsequent ballots of the same voter.

This motivates us to extend the functionality of blind accumulators. In section «Contexts», we enrich interfaces of the blind accumulator algorithms by adding to some of them a parameter that describes a target context: regular voting, revoting, second round voting, etc. To ensure that voter's object in different contexts are hard to associate with each other, we introduce an additional security requirement called severance. In section «Instantiation», we propose an instantiation of the extended blind accumulator scheme. In section «Security», we justify the security of the proposed instantiation.

Contexts

Cryptographic accumulators are special encodings of tuples of objects. We write $\mathbf{a} = [S]$ to denote that an accumulator $\mathbf{a}$ encodes a tuple S. We interpret tuples as ordered multisets bringing standard set notations such as the curly braces, the membership ($\in$) and union ($\cup$) symbols.

A blind accumulator scheme introduced in [1] is a tuple of polynomial-time algorithms `BAcc = (Init, Add, PrvAdd, VfyAdd, Der, PrvDer, VfyDer)` that are defined as follows.

1. The probabilistic algorithm `Init`: $1^l \mapsto \mathbf{a}_0$ takes a security level $l \in \mathbb{N}$ (in the unary form) and outputs an initial accumulator $\mathbf{a}_0 = [\varnothing]$.

We assume that $\mathbf{a}_0$ implicitly refers to l and public parameters (such as a description of an elliptic curve) and that these parameters implicitly define a set of private keys `SKeys` and a set of public keys `PKeys`.

2. The deterministic algorithm `Add`: $(\mathbf{a}, sk) \mapsto \mathbf{a}'$ takes an accumulator $\mathbf{a} = [S]$ and a private key sk, and outputs an updated accumulator $\mathbf{a}' = [S \cup \{sk\}]$.

We assume that **a** is an output of either `Init` or some previous call to `Add`. This ensures the consistency of **a**, i. e. that it is constructed as

$$\mathbf{a} \leftarrow \texttt{Add}\Big(\ldots\texttt{Add}\Big(\texttt{Add}\big(\texttt{Add}(\mathbf{a}_0,\ sk_1),\ sk_2\big),\ \ldots\Big),\ sk_n\Big),\ \mathbf{a}_0 \leftarrow \texttt{Init}\big(1^l\big),\ sk_i \in \texttt{SKeys},$$

and, therefore, is an incrementally built encoding $[S]$ of the multiset $S = \{sk_1,\ sk_2,\ \ldots,\ sk_n\}$.

We also assume that the public parameters referenced in the initial accumulator $\mathbf{a}_0$ are passed to all accumulators incrementally built from it.

3. The probabilistic algorithm `PrvAdd`: $(\mathbf{a},\ \mathbf{a}',\ sk) \mapsto \alpha$ takes accumulators **a**, **a′** and a private key *sk*, and generates a proof α that $\mathbf{a}' = \texttt{Add}(\mathbf{a},\ sk)$.

4. The deterministic algorithm `VfyAdd`: $(\mathbf{a},\ \mathbf{a}',\ \alpha) \mapsto b$ takes accumulators **a**, **a′** and a proof α that $\mathbf{a}' = \texttt{Add}(\mathbf{a},\ sk)$ for some private key *sk*. The algorithm verifies the proof and outputs either $b = 1$ for acceptance or $b = 0$ for rejection.

5. The deterministic algorithm `Der`: $(\mathbf{a},\ sk) \mapsto pk \,|\, \perp$ takes an accumulator **a** and a private key *sk*, and either derives a public key *pk* or outputs the error symbol $\perp$.

We require that for a consistent $\mathbf{a} = [S]$, $\texttt{Der}(\mathbf{a},\ sk) = \perp$ if and only if $sk \notin S$.

6. The probabilistic algorithm `PrvDer`: $(\mathbf{a},\ pk,\ sk) \mapsto \delta$ takes an accumulator **a**, a private key *sk* and a public key *pk*, and generates a proof δ that $pk \leftarrow \texttt{Der}(\mathbf{a},\ sk)$.

7. The deterministic algorithm `VfyDer`: $(\mathbf{a},\ pk,\ \delta) \mapsto b$ takes an accumulator **a**, a public key *pk* and a proof δ that $pk = \texttt{Der}(\mathbf{a},\ sk)$ for some private key *sk*. The algorithm verifies the proof and outputs either $b = 1$ for acceptance or $b = 0$ for rejection.

Further details on the algorithms and additional requirements are presented in [1].

To support contexts, we extend the interfaces of the last three algorithms. We describe a context with a non-empty binary word (string) *c* and use it as an additional input parameter of `Der`, `PrvDer` and `VfyDer`. Denote the resulting extension of `BAcc` by `BAcc1`.

The algorithm `Der` of `BAcc1` takes a triple $(\mathbf{a},\ sk,\ c)$ and outputs a public key *pk* bound to the context *c*. We require that if $sk \xleftarrow{\$} \texttt{SKeys}$, $sk \in S$ and $\mathbf{a} = [S]$, then $pk \leftarrow \texttt{Der}(\mathbf{a},\ sk)$ has a fixed distribution *D* over `PKeys` regardless of *S* and *c*.

Hereinafter we write $r_1,\ r_2,\ \ldots \xleftarrow{L} R$ to denote that $r_1, r_2, \ldots$ are chosen independently at random from a set *R* according to a probability distribution *L* and denote by \$ the uniform distribution.

The paper [1] defines four security requirements for blind accumulators, namely, consistency, soundness, blindness, and unlinkability. To extend these notions to `BAcc1`, we modify the last three requirements as follows:

- in the definition of soundness, the algorithms $\mathcal{A}$ and $\mathcal{E}$ take the additional input *c* that is transferred to `VfyDer` and `Der`, respectively;
- in the definition of blindness, the algorithms $\mathcal{S}_2$ takes the additional input *c* that is transferred to `Der` and `PrvDer`;
- in the definition of unlinkability, the game *G* takes the additional input *c* that is repeated when calling `Der`.

The consistency, soundness, blindness, and unlinkability do not guarantee that the public keys derived in different contexts are hard to associate with each other. To ensure such guarantees, we introduce an additional requirement called *severance*.

Consider an algorithm $\mathcal{A}$ that takes a consistent accumulator $\mathbf{a} = [S]$ of security level *l*, different context strings *c*, *c′* and public keys *pk*, *pk′*. The first public key is derived from a using $sk \in S$ in the context *c*. The algorithm $\mathcal{A}$ guesses if the second public key is also derived from a using *sk* but in the context *c′*. The algorithm returns 1 (true) or 0 (false). We allow all elements of *S* except *sk* to be predefined and thus known to $\mathcal{A}$. Therefore, it is enough to consider only the simplest case $S = \{sk\}$.

Definition 1. A scheme `BAcc1` provides severance if for any different context strings *c*, *c′* and for any probabilistic polynomial-time algorithm $\mathcal{A}$ described above it holds that

$$\mathbf{Adv}(\mathcal{A}) = \left| \mathbf{P}\left\{ \mathcal{A}(\mathbf{a},\ c,\ c',\ pk,\ pk') = 1 : \begin{matrix} \mathbf{a}_0 \leftarrow \texttt{Init}\big(\mathbf{1}^l\big),\ sk \xleftarrow{\$} \texttt{SKeys},\ \mathbf{a} \leftarrow \texttt{Add}(\mathbf{a}_0,\ sk) \\ pk \leftarrow \texttt{Der}(\mathbf{a},\ sk,\ c),\ pk' \leftarrow \texttt{Der}(\mathbf{a},\ sk,\ c') \end{matrix} \right\} - \right.$$

$$-\mathbf{P}\left\{\mathcal{A}(\mathbf{a}, c, c', pk, pk') = 1 : \begin{matrix} \mathbf{a}_0 \leftarrow \texttt{Init}\left(1^l\right),\ sk \xleftarrow{\$} \texttt{SKeys},\ \mathbf{a} \leftarrow \texttt{Add}(\mathbf{a}_0, sk) \\ pk \leftarrow \texttt{Der}(\mathbf{a}, sk, c),\ pk' \xleftarrow{D} \texttt{PKeys} \end{matrix}\right\}\Bigg|$$

is negligible in l.

Instantiation

In [1], an instantiation of the `BAcc` scheme, called `BAcc-DH`, is proposed. We modify `BAcc-DH` to support the `BAcc1` functionality. The resulting instantiation is called `BAcc1-DH`.

In `BAcc1-DH`, a cyclic group $\mathbb{G}_q$ of large prime order q is used. We write this group additively and denote by $\mathbb{G}_q^*$ the set of nonzero elements of $\mathbb{G}_q$. We also use the ring $\mathbb{Z}_q$ of residues of integers modulo q and the set $\mathbb{Z}_q^*$ of nonzero (invertible) residues. The group $\mathbb{G}_q$ is constructed in the algorithm `BAcc1-DH.Init`. An input security level l determines the bit length of q. Once $\mathbb{G}_q$ is constructed, the set of private keys `SKeys` and the set of public keys `PKeys` are defined as $\mathbb{Z}_q^*$ and $\mathbb{G}_q^*$, respectively. We use hash functions H and H_1 that map to these sets.

The initial accumulator $\mathbf{a}_0$ and all subsequent accumulators are words in the alphabet $\mathbb{G}_q^*$. The set of non-empty words in an alphabet Σ is denoted by Σ^+. An empty word is denoted by $\bot$. The notation $\left(\mathbb{G}_q^*\right)^+$ is shortened to $\mathbb{G}_q^{*+}$. For a word $\mathbf{w}$, let $|\mathbf{w}|$ be its length and $\texttt{most}(\mathbf{w})$ be the word obtained from $\mathbf{w}$ by dropping its last symbol. For $V \in \mathbb{G}_q^*$ and $u \in \mathbb{Z}_q^*$, let uV denote the u-multiple of V. A word that obtained from $\mathbf{w} \in \mathbb{G}_q^{*+}$ by replacing each symbol by its u-multiple is denoted as $u\mathbf{w}$. Two words in $\mathbb{G}_q^*$ can be added symbol-wise to obtain a word in $\mathbb{G}_q$.

In `BAcc1-DH`, the algorithms `Add`, `PrvAdd` and `VfyAdd` are the same as in `BAcc-DH`. The remaining algorithms are updated, the corrections are highlighted in frames in the listings below.

Algorithm `BAcc1-DH.Init`

```
Input: $1^l$ (security level).
Output: $\mathbf{a}_0 \in \mathbb{G}_q^{*+}$ (initial accumulator).
Steps:
1. Construct a group $\mathbb{G}_q$ of prime order $q$ such that $C_1 2^l < q < C_2 2^l$, where $C_1$, $C_2$ are some constants.
2. $\boxed{\text{Construct hash functions } H : \{0, 1\}^* \to \mathbb{Z}_q,\ H_1 : \{0, 1\}^+ \cup \{\bot\} \to \mathbb{G}_q^*.}$
3. $\boxed{G \leftarrow H_1(\bot).}$
4. $\mathbf{a}_0 \leftarrow G$.
5. Return $\mathbf{a}_0$.
```

The descriptions of $\mathbb{G}_q$ and G can be interpreted as additional outputs of the algorithm. We allow H to process arbitrary input data assuming they are pre-encoded into a binary word.

Algorithm `BAcc1-DH.Der`

```
Input: $\mathbf{a} \in \mathbb{G}_q^{*+}$ (accumulator), $u \in \mathbb{Z}_q^*$ (private key), $\boxed{c \in \{0, 1\}^+ \text{ (context)}}$.
Output: $V \in \mathbb{G}_q^*$ (public key).
Steps:
1. Parse $\mathbf{a} = G_0 G_1 \ldots G_n$.
2. Find $i \in \{1, 2, \ldots, n\}$ such that $uG_i = G_0$. If such $i$ does not exist, return $\bot$.
3. $\boxed{C \leftarrow H_1(c).}$
4. $\boxed{\text{Return } uC.}$
```

Algorithm `BAcc1-DH.PrvDer`

```
Input: $\mathbf{a} \in \mathbb{G}_q^{*+}$ (accumulator), $u \in \mathbb{Z}_q^*$ (private key), $V \in \mathbb{G}_q^*$ (public key), $\boxed{c \in \{0, 1\}^+ \text{ (context)}}$.
Output: $\delta \in \mathbb{Z}_q^+ \times \mathbb{Z}_q^+$ (proof).
```

Steps:

```
1. Parse $\mathbf{a} = G_0 G_1 \ldots G_n$.
2. Find $i \in \{1, 2, \ldots, n\}$ such that $uG_i = G_0$. If such $i$ does not exist, return $(0, 0)$.
3. $\boxed{C \leftarrow H_1(c).}$
4. For $j = 1, 2, \ldots, n$, $j \neq i$:
   a) $h_j, s_j \xleftarrow{\$} \mathbb{Z}_q$;
   b) $\boxed{\mathbf{r}_j \leftarrow s_j(G_j G_0) + h_j(CV).}$
5. $k_i \xleftarrow{\$} \mathbb{Z}_q$.
6. $\mathbf{r}_i \leftarrow k_i(G_i G_0)$.
7. $h_i \leftarrow \left(H(\mathbf{a}, \mathbf{r}_1\mathbf{r}_2\ldots\mathbf{r}_n, V) - \sum_{j \neq i} h_j\right) \bmod q$.
8. $s_i \leftarrow (k_i - uh_i) \bmod q$.
9. $\delta \leftarrow (h_1 h_2 \ldots h_n, s_1 s_2 \ldots s_n)$.
10. Return $\delta$.
```

Algorithm `BAcc1-DH.VfyDer`

Input: $\mathbf{a} \in \mathbb{G}_q^{*+}$ (accumulator), $V \in \mathbb{G}_q^*$ (public key), $\delta \in \mathbb{Z}_q^+ \times \mathbb{Z}_q^+$ (proof), $\boxed{c \in \{0, 1\}^+ \text{ (context)}}$.
Output: 1 (accept) or 0 (reject).
Steps:

```
1. Parse $\delta \leftarrow (\mathbf{h}, \mathbf{s})$. If $|\mathbf{h}| \neq |\mathbf{s}|$ or $|\mathbf{a}| \neq |\mathbf{h}| + 1$, return 0.
2. Parse $\mathbf{a} = G_0 G_1 \ldots G_n$, $\mathbf{h} = h_1 h_2 \ldots h_n$ and $\mathbf{s} = s_1 s_2 \ldots s_n$.
3. $\boxed{C \leftarrow H_1(c).}$
4. For $j = 1, 2, \ldots, n$:
   a) $\boxed{\mathbf{r}_j \leftarrow s_j(G_j G_0) + h_j(CV).}$
5. If $H(\mathbf{a}, \mathbf{r}_1\mathbf{r}_2\ldots\mathbf{r}_n, V) \not\equiv h_1 + h_2 + \ldots + h_n \pmod q$, return 0.
6. Return 1.
```

Security

In this section, we justify the security of `BAcc1-DH` examining five security requirements stated in [1] and section «Contexts».

The security definitions in [1] allow runtime environments to be managed. We use this to replace the hash functions H and H_1 with random oracles [2] and permit these oracles to be programmed. Technically, this is achieved by manipulating the random tape of the algorithm `BAcc1-DH.Init` which constructs H and H_1. The random oracle responds to a fresh input μ with a random output h and repeats a previous output when an input is repeated. Programming the oracle consists in assigning a given random output h to a given input μ. Conflicts can potentially occur when programming, namely, the input μ may already be associated with an output $h' \neq h$. Fortunately, we avoid conflicts.

To justify the unlinkability and severance, we use the well-known DDH (decisional Diffie – Hellman) problem [3]. This problem is specified with respect to a cyclic group $\mathbb{G}_q$ with a generator G and consists in deciding for a given tuple (G, uG, vG, wG), $u, v, w \in \mathbb{Z}_q^*$, if $w \equiv uv \pmod q$. The algorithm $\mathcal{B}$ that solves DDH guesses if this is indeed the case and outputs either 1 (true) or 0 (false).

Definition 2. Let $\mathcal{G}$ be an algorithm that constructs a cyclic group $\mathbb{G}_q$ and its generator G given an input 1^l. The DDH problem is hard with respect to $\mathcal{G}$ if for any polynomial-time algorithm $\mathcal{B}$ operating on $\mathbb{G}_q$ and G constructed by calling $\mathcal{G}(1^l)$ it holds that the advantage

$$\mathbf{Adv}(\mathcal{B}) = \left| \mathbf{P}\left\{ \mathcal{B}(G, uG, vG, uvG) = 1 : u, v \xleftarrow{\$} \mathbb{Z}_q^* \right\} - \mathbf{P}\left\{ \mathcal{B}(G, uG, vG, wG) = 1 : u, v, w \xleftarrow{\$} \mathbb{Z}_q^* \right\} \right|$$

is negligible in l. The probabilities here are over a random tape of $\mathcal{B}$ and $\mathcal{G}$ and over a random choice of u, v and w.

Theorem. *The* `BAcc1-DH` *instantiation of the* `BAcc1` *scheme satisfies the requirements of consistency, soundness, blindness, unlinkability, and severance in the programmable random oracle model provided that DDH is hard with respect to* `BAcc1-DH.Init`.

Proof. Let us examine security requirements each time switching to the scope of the corresponding security definition. For full details of the first four security definitions we refer the reader to [1].

Consistency. Let $\mathcal{E}$ control a random tape of the algorithm $\mathcal{A}$ and be able to restart (rewind) the algorithm with the tape repeating. This is possible since $\mathcal{E}$ is allowed to manage the runtime environment of $\mathcal{A}$. Let $\mathcal{A}$ return a proof $(\mathbf{r}, s)$ with $s = (k - hu) \bmod q$ on the first run. On the second run, the random tape is repeated and, therefore, the word $\mathbf{r}$ as well as the input $(\mathbf{a}, \mathbf{a}', \mathbf{r})$ to the oracle H are also repeated. The oracle is programmed to return a fresh random output h' on this input. Since h' differs from the first output h with probability $\frac{q-1}{q}$, after $\frac{q}{q-1} = 1 + O\left(\frac{1}{2^l}\right)$ restarts on average $\mathcal{E}$ gets $h' \neq h$ and the corresponding $s' = (k - h'u) \bmod q$. After that $\mathcal{E}$ determines

$$u = (s - s')(h' - h)^{-1} \bmod q.$$

We use here the standard arguments for Σ-protocols [4; 5].

Soundness. It is justified similarly to the consistency. A private key u is determined by two different outputs of H on the same input $(\mathbf{a}, \mathbf{r}_1 \mathbf{r}_2 \ldots \mathbf{r}_n, V)$.

Blindness. The algorithm $\mathcal{S}_1$ generates $h, s \xleftarrow{\$} \mathbb{Z}_q$, constructs $\mathbf{r} \leftarrow s\mathbf{a} + h\, \mathtt{most}(\mathbf{a}')$ and programs H, that is, assigns the output h to the input $(\mathbf{a}, \mathbf{a}', \mathbf{r})$. The algorithm $\mathcal{S}_1$ returns a pair $(\mathbf{r}, s)$ as a proof α. This proof is accepted by `BAcc-DH.VfyAdd` and is statistically indistinguishable from the standard proof generated by `BAcc-DH.PrvAdd` provided that H is a random oracle.

The algorithm $\mathcal{S}_2$ is constructed similarly.

Unlinkability. Let us construct an algorithm $\mathcal{B}$ that solves an instance (G, uG, vG, wG) of DDH by playing the game $G(1^l, n, m, c)$ for the role of $\mathcal{V}$.

The algorithm $\mathcal{B}$ acts as follows.

1. Programs the runtime environment when calling `BAcc1-DH.Init` in step 1 of the game:
   - uses $\mathbb{G}_q$ from the instance of DDH;
   - assigns $H_1(\perp) = G$ and $H_1(c) = vG$.
2. Generates $j \xleftarrow{\$} \{1, 2, \ldots, m\}$.
3. Processes `BAcc1-DH.Add` and `BAcc1-DH.PrvAdd` calls made by $\mathcal{A}$ and determines used private keys. To do this, $\mathcal{B}$ restarts $\mathcal{A}$ several times and extracts private keys from the provided proofs acting as the algorithm $\mathcal{E}$ that justifies the consistency. It takes $m + O\left(\frac{m}{2^l}\right)$ restarts on average to determine all the keys.
4. Makes its own calls to `BAcc1-DH.Add` (the order of calls is determined by $\mathcal{A}$) numbered $1, \ldots, j - 1, j + 1, \ldots, m$ using keys $u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_m \xleftarrow{\$} \mathbb{Z}_q^*$ generated by itself. The calls are accompanied by proofs constructed using `BAcc1-DH.PrvAdd`.
5. Makes the j call to `BAcc1-DH.Add` in a non-standard way embedding the private key u hidden in the instance (G, uG, vG, wG). To do this, performs transitions $G_i \mapsto uG_i$, using the knowledge of $d_i = \log_G G_i$ and determining uG_i as $d_i(uG)$. The discrete logarithms d_i are indeed known to $\mathcal{B}$, since they are products of its own private keys and $\mathcal{A}$ private keys extracted from the proofs.
6. Accompanies the j call to `BAcc1-DH.Add` with the a proof of consistency indistinguishable from the real one and obtained by programming the oracle H. Here $\mathcal{B}$ acts as the algorithm $\mathcal{S}_1$ that justifies the blindness. Note that the inputs of H when constructing proofs of consistency at different steps of accumulator management are certainly different since the length of the accumulators as words increases. Therefore, there are no conflicts when programming.
7. Processes the final accumulator $\mathbf{a} = G_0 G_1 \ldots G_n$ and generates public keys. The public keys V_i, $i \neq j$, are constructed using `BAcc1-DH.Der` as $u_i H_1(c) = u_i vG$. The public key V_j is constructed by the instance of DDH as wG. This is the correct public key with $w = uv \bmod q$ and a random public key with a random w. Let b be the indicator of the correctness of V_j. The bit b is unknown to $\mathcal{B}$ and is not used by it (unlike $\mathcal{V}$).
8. Passes $\mathcal{A}$ the public keys $(V_1, V_2, \ldots, V_m)$, waits the guess $\hat{b}$ and outputs it as its own guess to $\mathrm{DDH}(G, uG, vG, wG)$.

The algorithm $\mathcal{B}$ requires $m + O\left(\frac{m}{2^l}\right)$ restarts of $\mathcal{A}$ on average and additional time polynomial in l. Thus, if $\mathcal{A}$ is polynomial, then $\mathcal{B}$ is expected polynomial. At the same time,

$$\mathbf{Adv}(\mathcal{B}) = \left|\mathbf{P}\{\mathcal{B} = 1 | b = 1\} - \mathbf{P}\{\mathcal{B} = 1 | b = 0\}\right| = \left|\mathbf{P}\{\hat{b} = 1 | b = 1\} - \mathbf{P}\{\hat{b} = 1 | b = 0\}\right| = \mathbf{Adv}(\mathcal{A}).$$

This means that if DDH is hard, i. e. $\mathbf{Adv}(\mathcal{A})$ is negligible, then $\mathbf{Adv}(\mathcal{B})$ is also negligible and the unlinkability is ensured.

Severance. Let us construct an algorithm $\mathcal{B}$ that solves an instance (P, uP, vP, wP) of DDH using an algorithm $\mathcal{A}$ from definition 1. The algorithm $\mathcal{A}$ takes an accumulator $\mathbf{a}$ of capacity 1, different context strings $c, c' \in \{0, 1\}^+$ and public keys V, V'.

The algorithm $\mathcal{B}$ acts as follows.

1. Generates $r \xleftarrow{\$} \mathbb{Z}_q^*$ and calculates $G \leftarrow rP$.

2. Simulates the call $\mathbf{a}_0 \leftarrow$ `BAcc1-DH.Init`(1^l) using $\mathbb{G}_q$ from the instance of DDH and assigning $H_1(\perp) = G$ so that $\mathbf{a}_0 = G$. Additionally assigns $H_1(c) = P$, $H_1(c') = vP$.

3. Simulates the call $\mathbf{a} \leftarrow$ `BAcc1-DH.Add`$(\mathbf{a}_0, u)$ assigning $\mathbf{a} = G'G$, where $G' = uG = r(uP)$. Accompanies $\mathbf{a}$ with a proof of consistency indistinguishable from the real one and obtained by programming the oracle H. Here $\mathcal{B}$ acts as the algorithm $\mathcal{S}_1$ that justifies the blindness.

4. Using the instance of DDH, constructs public keys $V = uP = uH_1(c)$ and $V' = wP$. Note that V is the correct public key derived from $\mathbf{a}$ using u in the context c. If $w = uv \bmod q$, then $V' = u(vP) = uH_1(c')$ is the correct public key derived from $\mathbf{a}$ using u in the context c'. If w is random, then V' is a random public key. Let b be the indicator of the correctness of V'. The bit b is unknown to $\mathcal{B}$ and has to be guessed by it.

5. Passes $\mathcal{A}$ the tuple $(\mathbf{a}, c, c', V, V')$, waits the guess $\hat{b}$ and outputs it as its own guess to DDH(P, uP, vP, wP).

If $\mathcal{A}$ is polynomial, then $\mathcal{B}$ is also polynomial. At the same time, repeating the computations above, $\mathbf{Adv}(\mathcal{B}) = \mathbf{Adv}(\mathcal{A})$. This means that if DDH is hard, i. e. $\mathbf{Adv}(\mathcal{B})$ is negligible, then $\mathbf{Adv}(\mathcal{A})$ is also negligible and the severance is ensured.

References

1. Agievich S. Blind accumulators for e-voting. In: Nemoga K, Ploszek R, Zajac P, editors. *Proceedings of Central European conference on cryptology – CECC'22; 2022 June 26–29; Smolenice, Slovakia.* Bratislava: Mathematical Institute of the Slovak Academy of Sciences; 2022. p. 15–18.

2. Bellare M, Rogaway P. Random oracles are practical: a paradigm for designing efficient protocols. In: Denning DE, Pyle R, Ganesan R, Sandhu RS, Ashby V, editors. *CCS'93. Proceedings of the 1st ACM conference on computer and communications security; 1993 November 3–5; Fairfax, USA*. New York: Association for Computing Machinery; 1993. p. 62–73. DOI: 10.1145/168588.168596.

3. Boneh D. The decision Diffie – Hellman problem. In: Buhler JP, editor. *Algorithmic number theory. Proceedings of the Third International symposium, ANTS-III; 1998 June 21–25; Portland, USA.* Berlin: Springer; 1998. p. 48–63 (Goos G, Hartmanis J, van Leeuwen J, editors. Lecture notes in computer science; volume 1423). DOI: 10.1007/BFb0054851.

4. Cramer RJF. *Modular design of secure yet practical cryptographic protocols* [dissertation on the Internet]. Amsterdam: Universiteit van Amsterdam; 1997 [cited 2023 December 1]. 187 p. Available from: https://ir.cwi.nl/pub/21438/21438A.pdf.

5. Damgård I. *On Σ-protocols* [Internet]. Aarhus: University of Aarhus; 2002 [cited 2023 December 1]. 22 p. Available from: https://cs.au.dk/~ivan/Sigma.pdf.

Received 08.12.2023 / revised 13.03.2024 / accepted 13.03.2024.