# Single-Use Delegatable Signatures Based on Smart Contracts

Stephan Krenn
stephan.krenn@ait.ac.at
AIT Austrian Institute of Technology
Vienna, Austria

Thomas Lorünser
thomas.loruenser@ait.ac.at
AIT Austrian Institute of Technology
Vienna, Austria

## ABSTRACT

Delegation of cryptographic signing rights has found many application in the literature and the real world. However, despite very advanced functionalities and specific use cases, existing solutions share the natural limitation that the number of usages of these signing rights cannot be efficiently limited, but users can at most be disincentivized to abuse their rights.

In this paper, we suggest a solution to this problem based on blockchains. We let a user define a smart contract defining delegated signing rights, which needs to be triggered to successfully sign a message. By leveraging the immutability of the blockchain, our construction can now guarantee that a user-defined threshold of signature invocations cannot be exceeded, thereby circumventing the need for dedicated hardware or similar assistance in existing constructions for one-time programs.

We discuss different constructions supporting different features, and provide concrete implementations in the Solidity language of the Ethereum blockchain, proving the real-world efficiency and feasibility of our construction.

## KEYWORDS

Delegatable signatures, one-time programs, smart contracts

**ACM Reference Format:**
Stephan Krenn and Thomas Lorünser. 2021. Single-Use Delegatable Signatures Based on Smart Contracts. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–20, 2021, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3465481.3469192

## 1 INTRODUCTION

Digital signatures are the central cryptographic primitive to provide strong and provable authenticity and integrity guarantees. Over the last decades, numerous advanced, so-called malleable, signature schemes have been introduced, which allow the holder of the secret key to delegate certain signature rights to a delegate. Examples for such signature schemes include proxy signatures [8], poly-based signatures [3], functional signatures [9], blank signatures [31], redactable signatures [29], sanitizable signatures [11], or protean signatures [24]; for a detailed overview, we refer to Bilzhause et al. [6].


*ARES 2021, August 17–20, 2021, Vienna, Austria*

ACM ISBN 978-1-4503-9051-4/21/08…$15.00
https://doi.org/10.1145/3465481.3469192

Despite significant differences in terms of functionality and addressed use cases, all these schemes share the natural limitation that delegated rights are not limited in the number of usages of these delegated rights. However, this might be a very desirable property in many applications, e.g., for blank cheques.

*Related work.* In the following we give an overview of diverse existing approaches that aim at enforcing a single usage of cryptographic keys, which may also be adapted for the case of delegated signature rights.

One-time programs (OTPs), introduced by Goldwasser et al. [20], are computer programs that can only be executed a single time, and self-destruct afterwards. However, it is easy to see that such programs cannot be fully software-based, as it is always possible to copy and re-execute a piece of software. Consequently, constructions for one-time programs found in the literature require tamper-proof hardware, e.g., [17, 20, 23] or are based on trusted execution environments [32].

While it is known that perfect, information theoretically secure one-time programs are not even possible in the quantum setting [10], Roehsner et al. [27] proposed a probabilistic quantum-based solution for OTPs. In a nutshell, the idea is to encode the program onto quantum states in a way that it needs to be measured for evaluation, in which case the system changes its state and can no longer be reused. Furthermore, by the no-cloning theorem, the quantum program cannot be copied before execution. Specifically for the case of delegated signing rights, Roehsner et al. [27] showed how to make the failure probability arbitrarily small. However, even though the resulting signature is classical, their construction is not (yet) of practical interest as it requires to transfer quantum states between the user and the delegate, and to maintain this quantum state until measurement.

A complementary approach to prevent double-spending of cryptographic tokens (e.g., signature keys, e-cash, etc.) is to guarantee that some secret key is revealed if two linked activities are performed, e.g., [1, 4, 7, 13, 25]. While this approach disincentivizes users to use cryptographic tokens multiple times, its practical usability is sometimes limited. Depending on the application scenario it might be difficult to actually detect such a double usage. Furthermore, in this case, also potential physical consequences of issued signatures need to be annulled, causing substantial overhead. Specifically in the context of e-cash, online double-spending prevention avoids this problem by constantly contacting the issuing bank to detect whether a specific coin has already been spent [15]. This approach is well suited for centralized systems, but does not scale for scenarios where, e.g., single signature rights are to be delegated without the existence of a central party.

Finally, recently, Goyal and Goyal [22] proposed a generic solution for OTPs leveraging proof-of-stake based blockchains, without requiring trusted setup or random oracles.

*Our contribution.* In this paper, we propose a simple yet elegant solution which allows one to enforce a single use of a cryptographic token. Our solution is based on smart contracts, and intuitively works as follows: when delegating signing rights, the owner of a secret key publishes a smart contract, which may later be triggered by the delegate holding a secret key. By executing the smart contract, its state changes such that it cannot be invoked any more. Furthermore, the state contains a blinded hash of the signed message, such that the verifier can check that the received signature was indeed the first signature issued by this delegate.

We propose multiple variants for our mechanism for different scenarios, e.g., depending on whether a dedicated delegate is required or whether transparency is important.

Finally, we provide concrete implementations of our schemes using Ethereum's Solidity specificaton language for smart contracts, and demonstrate the real-world efficiency and cheapness of our solution.

## 2 PRELIMINARIES

We will next introduce the notation and required background that will be used in the remainder of this document.

### 2.1 Notation

We denote the main security parameter by $\lambda$. For a finite set $\mathcal{S}$, we write $s \xleftarrow{s} \mathcal{S}$ to denote that $s$ was sampled uniformly at random in $\mathcal{S}$. Similarly, we write $a \xleftarrow{s} \mathsf{A}(b)$ to denote that $a$ is assigned the outputs of a potentially randomized algorithm A on input $b$. All algorithms discussed throughout this paper are probabilistic polynomial time.

### 2.2 Cryptographic Background

We briefly recap the notation and security properties for digital signatures and zero-knowledge proofs, as well as our modeling of blockchains, but omit details due to space limitations and refer to the original literature.

*Digital signatures.* A digital signature scheme consists of four algorithms (SParGen, SKGen, SSign, SVerify):

- $pp_\Sigma \xleftarrow{s} \mathsf{SParGen}(1^\lambda)$**.** On input the security parameter, this algorithm outputs the public parameters $pp_\Sigma$, which are assumed to be implicit input to all further algorithms.
- $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{s} \mathsf{SKGen}(pp_\Sigma)$**.** On input the public parameters, this algorithm outputs a secret signing key sk and a corresponding verification key pk.
- $\sigma \xleftarrow{s} \mathsf{SSign}(\mathsf{sk}, m)$**.** On input a secret signing key and a message, this algorithm outputs a signature $\sigma$.
- $b \leftarrow \mathsf{SVerify}(\mathsf{pk}, m, \sigma)$**.** On input a verification key, a message, and a signature, this algorithm outputs a bit indicating whether to accept or to reject the signature.

Informally, a signature scheme is correct if every honestly generated signature also passes the verification algorithm. Furthermore, the scheme is said to be existentially unforgeable under chosen-message attacks (EUF-CMA), if no adversary can produce a valid on a new signature, even after having seen arbitrarily many signatures on messages of his choice. For a formal discussion we refer to the literature [21].

*Zero-knowledge proofs.* Non-interactive zero-knowledge proof of knowledge consists of three algorithms (ZKSetup, ZKProof, ZKVerify):

- $crs_\Pi \xleftarrow{s} \mathsf{ZKSetup}(1^\lambda)$**.** On input the security parameter, this algorithm outputs the common reference string $crs_\Pi$ for the proof system, which is assumed to be implicit input to all further algorithms.
- $\pi \xleftarrow{s} \mathsf{ZKProof}(y, w)$**.** On input a statement $y$ and a corresponding $w$ such that $(y, w) \in R$, this algorithm outptus a non-interactive zero-knowledge proof of knowledge for $w$.
- $b \leftarrow \mathsf{ZKVerify}(y, \pi)$**.** On input a statement $y$ and a proof $\pi$, this algorithm outputs a bit indicating whether to accept or to reject the proof.

Intuitively, such a proof system needs to be correct, i.e., every honestly generated proof should also pass the verification algorithm. The zero-knowledge property requires that no adversary can infer any information about $w$ only knowing $crs_\Pi$, $y$ and $\pi$; this is modeled through a simulator knowing a trapdoor to a (simulated) $crs_\Pi$ which, given as input a statement, generates simulated proofs that are indistinguishable from honestly generated ones. Finally, extractability requires that an adversary not knowing a valid witness is incapable of generated a valid proof for a given $y$; again, this is modeled through the existence of an algorithm, which, knowing a trapdoor to a (simulated) $crs_\Pi$ can efficiently extract a valid $w$ for every accepting proof $\pi$ for a given statement. For formal definitions and further discussion, we refer, e.g., to [14].

For readability, we will use the notation introduced by Camenisch and Stadler [12] to denote zero-knowledge proofs. That is, we will write:

$$\mathsf{NIZK}\left[(x_1, x_2) : Y_1 = x_1 G \quad \wedge \quad Y_2 = x_2 H\right](m)$$

to denote a non-interactive zero-knowledge proof of knowledge of $x_1, x_2$ such that the relation on the right hand side is satisfied. All protocols used in this paper can efficiently be instantiated in the random-oracle model using $\Sigma$-protocols [16, 28] and the Fiat-Shamir heuristic [19], which will also be used to bind a proof to a given $m$.

Following the observations of Bernhard et al [5], we assume that all relevant context–and in particular the statement to be proven–is used when computing the challenge in the Fiat-Shamir transform, even though we do not make this explicit to not disguise the notation.

*Blockchains.* In this paper, we consider a blockchain as a permissionless, public bulletin board with two natural properties. Namely, we require immutability, meaning that information written to the blockchain can not be altered or deleted, and we assume that adversarial forks can efficiently be distinguished from the actual block chain state. For detailed discussions, we refer, e.g., to Goyal and Goyal [22].

## 3 DEFINITIONS

The following sections first introduce the syntax and notation for single-use delegatable signatures, and then summarize the security requirements posed to such schemes.

### 3.1 Syntax

A single-use delegatable signature scheme consists of the following set of algorithms:

- $pp \xleftarrow{s} \mathsf{ParGen}(1^\lambda)$. On input the security parameter, this algorithm outputs public parameters $pp$.
- $(\mathsf{usk}, \mathsf{upk}) \xleftarrow{s} \mathcal{U}.\mathsf{KGen}(pp)$. On input the public parameters, this algorithm outputs a secret key usk and a corresponding public key upk for a user.
- $(\mathsf{dsk}, \mathsf{dpk}) \xleftarrow{s} \mathcal{D}.\mathsf{KGen}(pp)$. On input the public parameters, this algorithm outputs a secret key dsk and a corresponding public key dpk for a delegate.
- $(\mathsf{osk}, sc) \xleftarrow{s} \mathsf{Delegate}(\mathsf{usk}, \mathsf{dpk}, aux)$. On input a user's secret key, a delegate's public key, and some auxiliary information $aux$ this algorithm outputs a delegated one-time signature key osk, as well as a value $sc$ (which in our case will be a smart contract published in a blockchain).
- $(\sigma, tr) \xleftarrow{s} \mathcal{D}.\mathsf{Sign}(\mathsf{dsk}, \mathsf{osk}, \mathsf{upk}, m)$. This algorithm allows a delegate holding a one-time key to compute a signature $\sigma$ on a message $m$. Furthermore, the algorithm outputs an auxiliary value $tr$ (which in our case will be trigger for the smart contract).
- $(\sigma, tr) \xleftarrow{s} \mathcal{U}.\mathsf{Sign}(\mathsf{usk}, \mathsf{osk}, \mathsf{dpk}, m)$. This algorithm allows the user to compute a signature on a message as well as an auxiliary value $tr$.
- $sc' \xleftarrow{s} \mathsf{BCUpdate}(sc, tr, aux)$. Knowing $tr$, this algorithm updates the state $sc$ (which in our case will be a modification of the state of smart contract).
- $b \leftarrow \mathsf{Verify}(\mathsf{upk}, \mathsf{dsk}, m, \sigma, sc, aux)$. This message outputs a bit indicating whether to accept or to reject a signature for a given message depending also on $sc$ and $aux$ (which in our case will be the state of the blockchain).

### 3.2 Security Requirements

In the following we informally discuss the security requirements expected from a single-use delegatable signature scheme. A full formalization of these requirements is left for future work.

*Completeness.* Completeness requires that, if all parties behave honestly, signatures will always verify correctly.

*Unforgeability.* Strong unforgeability requires that an adversary neither knowing the user's nor the delegate's secret key and the one-time key can generate a valid signature on its own. In the case that an adversary can generate new signatures on messages previously signed by the user or the delegate, we say that the scheme satisfies weak unforgeability.

*Transparency.* For malleable signatures, transparency typically requires that an outsider not knowing any secret keys can decide whether a valid signature has been generated by the user or by the delegate. In the context of our work we additionally require that also the originator of $tr$ cannot be determined, in order to also guarantee transparency during the BCUpdate process.

*Onetimeness.* Onetimeness requires that a delegated signature key osk can only be used to sign a single message, either by the user or by the delegate. Any further attempt to re-use osk, even by a legitimate user, will result in an invalid signature. As for unforgeability, we distinguish between weak and strong onetimeness, depending on whether multiple valid signatures for the same message can be generated or not.

## 4 CONSTRUCTIONS

In the following we present constructions of single-use delegatable signatures. We first present a very basic scheme where delegated rights can be forwarded to third party (yet only consumed once). We then present a scheme with a designated verifier, and subsequently discuss possible extensions to achieve accountability, $n$-time signatures, and more.

### 4.1 A Basic Scheme

The idea of our basic scheme is that the user puts a signed commitment by means of a smart contract into a blockchain, and gives the opening of the commitment to a delegate. To sign, the delegate provides a zero-knowledge proof of knowledge of the opening to the blockchain network, which verifies the proof and locks the smart contract by storing the hash of the signed message.

Somewhat surprisingly, for the basic construction, the delegate does not need to own any local secret key, i.e., dsk can be set to $\perp$; furthermore, when signing a message using the delegated signing key, the delegate does not need to generate an actual signature (i.e., $\sigma = \perp$): because of the soundness of the NIZK, already the fact that $\mathsf{h}(m)$ is stored in the smart contract suffices to convince the verifier that a legitimate entity (i.e., the signer or the delegate) have triggered the signing process.

In the following presentation, let (SKGen, SSign, SVerify) be a EUF-CMA secure signature scheme.

- $\mathsf{ParGen}(1^\lambda)$ outputs $pp = (1^\lambda, \mathcal{G}, G, q)$, where $\mathcal{G} = \langle G \rangle$ is a cyclic group of prime order $q$, such that the discrete logarithm problem is hard in $\mathcal{G}$.
- $\mathcal{U}.\mathsf{KGen}(pp)$ outputs a key pair $(\mathsf{usk}, \mathsf{upk}) \xleftarrow{s} \mathsf{SKGen}(1^\lambda)$.
- $\mathcal{D}.\mathsf{KGen}(pp)$ outputs $(\mathsf{dsk}, \mathsf{dpk}) = (\perp, \perp)$.
- $\mathsf{Delegate}(\mathsf{usk}, \mathsf{dpk}, aux)$ samples $\mathsf{osk} \xleftarrow{s} \mathbb{Z}_q$. The algorithm furthermore computes $Y = \mathsf{osk} \cdot G$ and $\tau = \mathsf{SSign}(\mathsf{usk}, (Y, aux))$, where $aux$ is a unique identifier of the block into which the smart contract will be inserted in the blockchain. Finally, the algorithm defines $sc$ as a stateful smart contract for the following functionality:
  - The contract fixes $pp$, $Y$, $\tau$, and upk, and initializes its internal state as $st = \varepsilon$.
  - Being called on input $\pi$ and h, the contract first checks whether $st = \varepsilon$ and aborts if this is not the case.
  - It then checks whether $\pi$ is a valid NIZK of osk such that $Y = \mathsf{osk} \cdot G$. If this is the case, it sets $st = \mathsf{h}$.
- $\mathcal{D}.\mathsf{Sign}(\mathsf{dsk}, \mathsf{osk}, \mathsf{upk}, m)$ computes $\mathsf{h} = \mathsf{h}(m)$ and
$$\pi \leftarrow \mathsf{NIZK}[(\mathsf{osk}) : Y = \mathsf{osk} \cdot G](\mathsf{h}) .$$
It outputs $\sigma = \perp$ and $tr = (\pi, \mathsf{h})$.
- $\mathcal{U}.\mathsf{Sign}(\mathsf{usk}, \mathsf{osk}, \mathsf{dpk}, m)$ computes $\mathsf{h} = \mathsf{h}(m)$ and
$$\pi \leftarrow \mathsf{NIZK}[(\mathsf{osk}) : Y = \mathsf{osk} \cdot G](\mathsf{h}) .$$
It outputs $\sigma = \perp$ and $tr = (\pi, \mathsf{h})$.

- BCUpdate($sc, tr$) checks that $tr = (\pi, \mathsf{h})$ is as defined in the $sc$ and ouputs $sc$ with the potentially updated $st$.
- Verify(upk, dsk, $m, \sigma, sc$) outputs 1 if and only if $sc$ contains a valid signature for upk and if $st = \mathsf{h}(m)$.

*4.1.1 Security Considerations.* Correctness of the scheme follows immediately by inspection.

Regarding unforgeability, one can see that the smart contract is bound to the specific block on the blockchain by the inclusion of the block identifier in the user's signature $\tau$. Therefore, any attempt to forge a signature would need to either forge a signature of the underlying EUF-CMA signature scheme, or leverage the given instance of the smart contract $sc$. Now, by the soundness properties of the deployed NIZK system, it follows that knowledge of osk is required to generate a valid $tr$ to activate the execution of $sc$.

From the immutability properties and the usual soundness assumptions of the block chain (honest majority, etc.), it furthermore follows that the smart contract can only be executed once, and thus strong onetimeness follows.

Finally, the distributions of signatures generated by the user and the delegate are identical, and thus transparency follows immediately.

Formal proofs, together with formal security definitions, are planned for future work.

*4.1.2 Discussion.* The above construction does not define a dedicated delegate. That is, the delegate could further delegate the signing rights by simply forwarding osk to a third party, without having to reveal any sensitive private key material. While this may be desirable in certain situations, we will discuss in the following constructions where the delegate is defined by the user and forwarding of signing rights is prohibited. However, it is worth noting that this delegation does not contradict our unforgeability definition, as the $\mathsf{dsk} = \bot$ would be known to a third party, and therefore the requirement that dsk and osk need to be known to generate a signature would be satisfied. Furthermore, we note that also onetimeness is not affected by forwarding osk, as the blockchain network would still only accept the first invocation of $sc$.

We also note that somewhat surprisingly it is not necessary to attach an actual signature to the signed message, as the pure fact that a hash value is stored in the smart contract's state is sufficient to prove the authenticity of the message. However, certain situations might require that for privacy reasons this hash value does not enable an attacker to reconstruct the signed message. This can be addressed by simply replacing $\mathsf{h} = \mathsf{h}(m)$ by $\mathsf{h} = \mathsf{h}(r, m)$ for $r \leftarrow \{0,1\}^{2\lambda}$, and defining $\sigma = r$ as the actual signature. By doing so, it is guaranteed that h statistically hides any information about $m$.

Finally, the value of $\mathsf{osk} = x$ does not need to be kept secret any longer once $sc$ has been updated and the signing right has been consumed. However, the NIZK cannot simply be replaced by sending $x$ in the plain, as malicious nodes could otherwise modify h without being detected before the transaction has been sufficiently distributed within the network. It is thus important to bind the knowledge of $x$ to the value of h.

## 4.2 Adding a Designated Delegate

We next present an extension of the basic scheme which allows for a dedicated designate. To achieve this, it becomes necessary for delegates to have sensitive private keys.

For a basic scheme, the user would now simply sign the delegate's public key dpk instead of $Y$ in the Delegate algorithm. In order to trigger the smart contract, the delegate would then no longer compute a NIZK for the discrete logarithm of $Y$, but for dsk corresponding to dpk. In order to also give the user the option to trigger the smart contract herself, also a NIZK for usk corresponding to upk would be accepted. It would now be guaranteed that only the holder of dsk or of usk could sign the message, and thus the delegate could no longer forward signing rights as this would require to reveal dsk.

However, while this construction is complete, unforgeable, and onetime, it does not achieve transparency, as the statement proven by the NIZK would reveal whether it was generated by the user or the delegate. In order to achieve symmetry, the NIZK thus shows that one either knows the user's or the delegate's secret key.

More precisely, the full construction for a designated-delegate signature scheme is given by the following algorithms:

- $\mathsf{ParGen}(1^\lambda)$ outputs $pp = (1^\lambda, \mathcal{G}, G, q)$, where $\mathcal{G} = \langle G \rangle$ is a cyclic group of prime order $q$, such that the discrete logarithm problem is hard in $\mathcal{G}$.
- $\mathcal{U}.\mathsf{KGen}(pp)$ computes a key pair $(\mathsf{usk}', \mathsf{upk}') \xleftarrow{s} \mathsf{SKGen}(1^\lambda)$. Furthermore, it chooses $\mathsf{usk}'' \xleftarrow{s} \mathbb{Z}_q$ and sets $\mathsf{upk}'' \leftarrow \mathsf{usk}'' \cdot G$. Finally, it outputs
$$(\mathsf{usk}, \mathsf{upk}) = ((\mathsf{usk}', \mathsf{usk}''), (\mathsf{upk}', \mathsf{upk}'')).$$
- $\mathcal{D}.\mathsf{KGen}(pp)$ chooses $\mathsf{dsk} \xleftarrow{s} \mathbb{Z}_q$ and sets $\mathsf{dpk} \leftarrow \mathsf{dsk}G$.
- $\mathsf{Delegate}(\mathsf{usk}, \mathsf{dpk}, aux)$ sets $\mathsf{osk} = \bot$. It then parses $\mathsf{usk} = (\mathsf{usk}', \mathsf{usk}'')$ and computes $\tau = \mathsf{SSign}(\mathsf{usk}', (\mathsf{dpk}, aux))$. Here $aux$ is a unique identifier of the block into which the smart contract will be inserted in the blockchain. Finally, the algorithm defines $sc$ as a stateful smart contract for the following functionality:
  - The contract fixes $pp$, dpk, $\tau$, and $\mathsf{upk} = (\mathsf{upk}', \mathsf{upk}'')$, and initializes its state $st = \varepsilon$.
  - Being called on input $\pi$ and h, the contract first checks whether $st = \varepsilon$ and aborts if this is not the case.
  - It then checks whether $\pi$ is a valid NIZK for $\mathsf{usk}''$ corresponding to $\mathsf{upk}''$ or for dsk corresponding to dpk. If this is the case, it sets $st = \mathsf{h}$.
- $\mathcal{D}.\mathsf{Sign}(\mathsf{dsk}, \mathsf{osk}, \mathsf{upk}, m)$ computes $\mathsf{h} = \mathsf{h}(m)$ and
$$\pi \leftarrow \mathsf{NIZK}[(\mathsf{dsk}, \mathsf{usk}'') : \mathsf{dpk} = \mathsf{dsk} \cdot G \quad \vee \quad \mathsf{upk}'' = \mathsf{usk}'' \cdot G](\mathsf{h}),$$
thereby using dsk as the witness and proving the first literal of the clause. It outputs $\sigma = \bot$ and $tr = (\pi, \mathsf{h})$.
- $\mathcal{U}.\mathsf{Sign}(\mathsf{usk}, \mathsf{osk}, \mathsf{dpk}, m)$ computes $\mathsf{h} = \mathsf{h}(m)$ and
$$\pi \leftarrow \mathsf{NIZK}[(\mathsf{dsk}, \mathsf{usk}'') : \mathsf{dpk} = \mathsf{dsk} \cdot G \quad \vee \quad \mathsf{upk}'' = \mathsf{usk}'' \cdot G](\mathsf{h}),$$
thereby using $\mathsf{usk}''$ as the witness and proving the second literal of the clause. It outputs $\sigma = \bot$ and $tr = (\pi, \mathsf{h})$.

- BCUpdate$(sc, tr)$ checks that $tr = (\pi, \mathsf{h})$ is as defined in the $sc$ and ouputs $sc$ with the potentially updated $st$.
- Verify$(\mathsf{upk}, \mathsf{dsk}, m, \sigma, sc)$ outputs 1 if and only if $sc$ contains a valid signature for $\mathsf{upk}'$ and if $st = \mathsf{h}(m)$.

Note that the NIZKs computed by the delegate and the user are indistinguishable due to the zero-knowledge property.

### 4.3 Further Extensions

Our basic constructions can be extended in various directions, depending on the specific needs and requirements of the use case.

*Accountability.* The constructions presented above do not offer any possibility to identify the originator of a specific signature, as both the user as well as the delegate could equally trigger the smart contract. Accountability enables a predefined third party acting as a judge to identify the signer, see, e.g., Beck et al [2]. One way to achieve this in our protocols would be to let the signer encrypt its public key, and later prove that the public key contained in the ciphertext is the key for which the corresponding secret key is known. That is, the NIZK would be changed to the following:

$$\pi \leftarrow \mathsf{NIZK}[(\mathsf{dsk}, \mathsf{usk}'', r) : (\mathsf{dpk} = \mathsf{dsk} \cdot G \wedge c = \mathsf{Enc}(\mathsf{dpk}; r)) \quad \vee \quad \left(\mathsf{upk}'' = \mathsf{usk}'' \cdot G \wedge c = \mathsf{Enc}(\mathsf{upk}''; r)\right)](\mathsf{h}),$$

where $c$ is an encryption of the public key with randomness $r$. Here, one can think of the encryption scheme as the ElGamal crypto system [18].

For this proof, the delegate would use dsk and $r$ in order to prove the first statement, while the user could use $\mathsf{usk}''$ and $r$ to prove the latter statement. It is important to note that the NIZK implicitly also proves that the *same* value for which the discrete logarithm is known, is also encrypted within the ciphertext $c$, and by the soundness property it is thus infeasible to encrypt a different value in order to escape accountability.

*Immutability.* For instance in the scenario of sanitizable or blank signatures, the user may wish to fix certain parts of the message a delegate can sign. To achieve this, the user commits to the restrictions (e.g., in form of a message template, or as a circuit which outputs 1 if and only if the signed message was valid) as part of the smart contract, and hands over the opening of the commitment to the delegate. Now, depending on the privacy requirements–whether or not the restrictions may be known to the verifier–the delegate either forwards the opening to the verifier as part of $\sigma$, or computes a NIZK proving that the (known) signed message is indeed valid with respect to the (secret) restrictions; note however that the latter may be computationally expensive depending on the valid modifications.

*Multiple delegates and n-time signatures.* Our constructions can directly be extended to multiple delegates, by letting the user defining a list of public keys that are allowed to act on behalf of him. Also, $n$-times signatures can be obtained by storing a list of up to $n$ hash values before denying further execution of the smart contract. Here, though causing some computational overhead and thus increasing the costs of the smart contract, the user could hide the upper bound $n$ from the public by only signing a commitment on it, and the delegate could prove that the number of preceding invocations is smaller than the number hidden in the commitment.

```
pragma solidity ^0.4.14;
pragma experimental ABIEncoderV2;
import "./altbn128.sol";

// One time delegatable signatures
contract Otds
{
    // contract state
    address public owner;
    uint256 state;
    Curve.G1Point comm;

    // constructor initializing state and delegation
    constructor(Curve.G1Point _comm) public {
        owner = msg.sender;
        state = 0;
        comm = _comm;
    }

    // function called by delegatee to sign once
    function OtdsSign(uint256 c, uint256 r, uint256
         hmessage)
        public
    {
        require(state == 0, "Already signed.");
        Curve.G1Point memory tp = Curve.g1add(
                Curve.g1mul(Curve.P1(), r % Curve.N()),
                Curve.g1mul(comm, c % Curve.N()));
        uint256 cp =  uint256(
                keccak256(abi.encodePacked(
                comm.X, comm.Y, tp.X, tp.Y, hmessage)));
        require(c == cp, "Invalid proof.");
        state = hmessage;
    }
}
```

**Listing 1: Basic scheme as defined in Section 4.1**

## 5 EVALUATION

In the following we provide implementations of the schemes specified above in the Solidity language for smart contracts on the Ethereum blockchain. Our implementation partially leverages existing elliptic curve implementations in Solidity [26, 30], and was implemented using the Remix Suite for Solidity smart contracts, which was also used to compute the cost estimates. The resulting code is given in Listings 1 and 2.

In contrast to the abstract specification of our schemes it is not necessary to let the user sign the smart contract in the concrete implementation, as in Ethereum every transaction is anyways signed, and the smart contract thus points back to its sender. If, however, a binding to an existing public key outside of Ethereum is important, including a signature as in the construction would be a straightforward modification.

Note that locally executed algorithms (i.e., for signing and verification) are not depicted here due to space limitations, and as they do not need to be included in the contract.

Ethereum distinguishes two types of costs related to smart contracts. On the one hand, *transaction costs* are based on the costs for sending a smart contract to the blockchain, and depends on fixed costs for transactions and smart contracts, as well as the size of the smart contract to be deployed. On the other hand, *execution costs* are based on the actual computations which need to be performed

```
pragma solidity ^0.4.14;
pragma experimental ABIEncoderV2;
import "./altbn128.sol";

// One time delegatable signatures
contract OtdsOR
{
    // contract state and public parameters
    address public owner;
    uint256 public state;
    Curve.G1Point g1;
    Curve.G1Point g2;
    Curve.G1Point y1;
    Curve.G1Point y2;

    constructor(Curve.G1Point _g1, Curve.G1Point _g2,
        Curve.G1Point _y1, Curve.G1Point _y2)
        public
    {
        // init state and public parameters
        owner = msg.sender;
        state = 0;
        g1 = _g1;
        g2 = _g2;
        y1 = _y1;
        y2 = _y2;
    }

    function OtdsSign(uint256 c1, uint256 c2,
        uint256 r1, uint256 r2, uint256 hmessage)
        public
    {
        require(state == 0, "Already signed.");
        // proof is (c1, c2, r1, r2)
        Curve.G1Point memory t1p = Curve.g1add(
                Curve.g1mul(y1, c1 % Curve.N()),
                Curve.g1mul(g1, r1 % Curve.N()));
        Curve.G1Point memory t2p = Curve.g1add(
                Curve.g1mul(y2, c2 % Curve.N()),
                Curve.g1mul(g2, r2 % Curve.N()));
        uint256 cp = uint256(keccak256(abi.encodePacked(
                g1.X, g1.Y, y1.X, y1.Y,
                g2.X, g2.Y, y2.X, y2.Y,
                t1p.X, t1p.Y, t2p.X, t2p.Y, hmessage)
                )) % Curve.N();
        require(addmod(c1, c2, Curve.N()) == cp,
                "ERROR: Invalid proof.");
        // accept signature
        state = hmessage;
    }
}
```

**Listing 2: Advanced scheme as defined in Section 4.2**

as the result of a transaction. This *gas* is caluclated in *gwei*, where 1ETH = $10^9$gwei; simple transactions require $21k$ gas, whereas complex transactions can easily exceed $1M$ gas. Therefore, for contracts to be practical they have not only to be implementable, but also with reasonable cost for the users.

Table 1 shows the transaction costs and execution costs for the contracts presented above. For these contracts, the user needs to pay the transaction costs, while the delegate would need to pay for the execution costs.

At the time of writing this paper, 1ETH ≈ 2′500USD,[1] resulting in about 0.25c/kGas. Thus, for instance, the transaction costs for a basic signature are about 16c, while the execution costs are about 10c, which can be considered practical for many sensitive applications compared to the costs caused by potential abuse of delegated rights. It can be seen that the main costs are due when initializing the smart contract. These costs could easily be amortized by modifying the contract in a way that it can be called by many users and delegates, instead of using one contract per delegation.

[1] https://coinmarketcap.com/currencies/ethereum/

| | Transaction costs in kGas | Execution costs in kGas |
|---|---|---|
| Plain EC multiplication | 30 | 8 |
| Basic scheme constructor | 500 | 34 |
| Basic scheme signature | 67 | 41 |
| Advanced scheme constructor | 843 | 62 |
| Advanced scheme signature | 95 | 69 |

**Table 1: Overview of costs measured (rounded to kGas).**

## 6 CONCLUSION

In this paper we presented an alternative approach to enforcing the limited use of delegated cryptographic rights. Instead of relying on special-purpose hardware or aiming at disincentivizing delegates to abuse their rights, our approach leverages smart contracts to upper bound the number of invocations of delegated rights. We provided concrete implementations of the corresponding smart contracts for the Ethereum blockchain, proving the real-world applicabiltiy of our schemes.

Future work will aim at extending the approach to additional applications beyond basic signature schemes.

## ACKNOWLEDGMENTS

The projects leading to this work have received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830929 ("CyberSec4Europe"), from the SESAR Joint Undertaking under grant agreement No 890456 ("SlotMachine"), and from the Austrian Research Promotion Agency ("FlexProd").

## REFERENCES

[1] Foteini Baldimtsi, Melissa Chase, Georg Fuchsbauer, and Markulf Kohlweiss. 2015. Anonymous Transferable E-Cash. 101–124. https://doi.org/10.1007/978-3-662-46447-2_5

[2] Michael Till Beck, Jan Camenisch, David Derler, Stephan Krenn, Henrich C. Pöhls, Kai Samelin, and Daniel Slamanig. 2017. Practical Strongly Invisible and Strongly Accountable Sanitizable Signatures. 437–452.

[3] Mihir Bellare and Georg Fuchsbauer. 2014. Policy-Based Signatures. 520–537. https://doi.org/10.1007/978-3-642-54631-0_30

[4] Mihir Bellare, Bertram Poettering, and Douglas Stebila. 2017. Deterring Certificate Subversion: Efficient Double-Authentication-Preventing Signatures. 121–151. https://doi.org/10.1007/978-3-662-54388-7_5

[5] David Bernhard, Olivier Pereira, and Bogdan Warinschi. 2012. How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios. 626–643. https://doi.org/10.1007/978-3-642-34961-4_38

[6] Arne Bilzhause, Henrich C. Pöhls, and Kai Samelin. 2017. Position Paper: The Past, Present, and Future of Sanitizable and Redactable Signatures. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*. ACM, 87:1–87:9. https://doi.org/10.1145/3098954.3104058

[7] Jan Bobolz, Fabian Eidens, Stephan Krenn, Daniel Slamanig, and Christoph Striecks. 2020. Privacy-Preserving Incentive Systems with Highly Efficient Point-Collection. 319–333. https://doi.org/10.1145/3320269.3384769

[8] Alexandra Boldyreva, Adriana Palacio, and Bogdan Warinschi. 2012. Secure Proxy Signature Schemes for Delegation of Signing Rights. 25, 1 (Jan. 2012), 57–115. https://doi.org/10.1007/s00145-010-9082-x

[9] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. 2014. Functional Signatures and Pseudorandom Functions. 501–519. https://doi.org/10.1007/978-3-642-54631-0_29

[10] Anne Broadbent, Gus Gutoski, and Douglas Stebila. 2013. Quantum One-Time Programs - (Extended Abstract). 344–360. https://doi.org/10.1007/978-3-642-40084-1_20

[11] Jan Camenisch, David Derler, Stephan Krenn, Henrich C. Pöhls, Kai Samelin, and Daniel Slamanig. 2017. Chameleon-Hashes with Ephemeral Trapdoors - And Applications to Invisible Sanitizable Signatures. 152–182. https://doi.org/10.1007/978-3-662-54388-7_6

[12] Jan Camenisch and Markus Stadler. 1997. Efficient Group Signature Schemes for Large Groups (Extended Abstract). 410–424. https://doi.org/10.1007/BFb0052252

[13] Dario Catalano, Georg Fuchsbauer, and Azam Soleimanian. 2020. Double-Authentication-Preventing Signatures in the Standard Model. 338–358. https://doi.org/10.1007/978-3-030-57990-6_17

[14] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. 2014. Malleable Signatures: New Definitions and Delegatable Anonymous Credentials. 199–213. https://doi.org/10.1109/CSF.2014.22

[15] David Chaum. 1983. Blind Signature System. 153.

[16] Ronald Cramer. 1997. *Modular Design of Secure yet Practical Cryptographic Protocols.* Ph.D. Dissertation. CWI Amsterdam, The Netherlands.

[17] Konrad Durnoga, Stefan Dziembowski, Tomasz Kazana, and Michal Zajac. 2013. One-Time Programs with Limited Memory. In *Information Security and Cryptology - 9th International Conference, Inscrypt 2013, Guangzhou, China, November 27-30, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8567)*, Dongdai Lin, Shouhuai Xu, and Moti Yung (Eds.). Springer, 377–394.

[18] Taher ElGamal. 1985. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. 31 (1985), 469–472.

[19] Amos Fiat and Adi Shamir. 1987. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. 186–194. https://doi.org/10.1007/3-540-47721-7_12

[20] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2008. One-Time Programs. 39–56. https://doi.org/10.1007/978-3-540-85174-5_3

[21] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. 1988. A Digital Signature Scheme Secure Against Adaptive Chosen-message Attacks. 17, 2 (April 1988), 281–308.

[22] Rishab Goyal and Vipul Goyal. 2017. Overcoming Cryptographic Impossibility Results Using Blockchains. 529–561. https://doi.org/10.1007/978-3-319-70500-2_18

[23] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. 2010. Founding Cryptography on Tamper-Proof Hardware Tokens. 308–326. https://doi.org/10.1007/978-3-642-11799-2_19

[24] Stephan Krenn, Henrich C. Pöhls, Kai Samelin, and Daniel Slamanig. 2018. Protean Signature Schemes. 256–276. https://doi.org/10.1007/978-3-030-00434-7_13

[25] Bertram Poettering and Douglas Stebila. 2014. Double-Authentication-Preventing Signatures. 436–453. https://doi.org/10.1007/978-3-319-11203-9_25

[26] Christian Reitwiessner. 2017. zkSNARKs test code. https://gist.github.com/chriseth. last accessed on March 23, 2021.

[27] Marie-Christine Roehsner, Joshua A. Kettlewell, Tiago B. Batalhão, Joseph F. Fitzsimons, and Philip Walther. 2018. Quantum advantage for probabilistic one-time programs. *Nature Communications* 9 (2018).

[28] Claus-Peter Schnorr. 1990. Efficient Identification and Signatures for Smart Cards. 239–252. https://doi.org/10.1007/0-387-34805-0_22

[29] Ron Steinfeld, Laurence Bull, and Yuliang Zheng. 2002. Content Extraction Signatures. 285–304.

[30] Kendrick Tan. 2019. Heiswap Dapp. https://github.com/kendricktan/. last accessed on March 23, 2021.

[31] Yujue Wang, HweeHwa Pang, and Robert H. Deng. 2018. Verifiably encrypted cascade-instantiable blank signatures to secure progressive decision management. *Int. J. Inf. Sec.* 17, 3 (2018), 347–363. https://doi.org/10.1007/s10207-017-0372-2

[32] Lianying Zhao, Joseph I. Choi, Didem Demirag, Kevin R. B. Butler, Mohammad Mannan, Erman Ayday, and Jeremy Clark. 2019. One-Time Programs Made Practical. 646–666. https://doi.org/10.1007/978-3-030-32101-7_37