On the Revocation of U-Prove Tokens

Christian Paquin, Microsoft Research

September 2nd 2014

U-Prove tokens provide many security and privacy benefits over conventional credential technologies such as X.509 certificates. Like any long-lived credentials, there might be a need to revoke issued U-Prove tokens before they expire. Achieving this might seem counterintuitive: how can you revoke an identity when users are anonymous or pseudonymous? This paper explores various revocation mechanisms compatible with the U-Prove technology, to help system designers select the best one for their applications. All of these mechanisms can be implemented today with the core and extensions U-Prove C# SDKs.

Introduction

U-Prove tokens provide many security and privacy benefits over conventional credential technologies such as X.509 certificates.[1] They allow users to minimally disclose whatever is necessary to satisfy a relying party's access policy, without leaking further information. In many scenarios, users can remain pseudonymous or even anonymous, while enjoying attribute-based access to resources and services.

Like any long-lived credentials, there might be a need to revoke issued U-Prove tokens before they expire. Indeed, a user might want to invalidate compromised or stolen tokens, or might need to be prevented from using them while they are still valid (because, say, she lost her access rights to certain resources, or acted maliciously). Achieving this might seem counterintuitive: how can you revoke an identity when users are anonymous or pseudonymous? This paper explores various revocation mechanisms compatible with the U-Prove technology, to help system designers select the best one for their applications.

Most mechanisms are facilitated by an entity called the *revocation authority* (RA). The revocation authority typically maintains and distributes the revocation list; how users get added to the revocation list is mechanism/application specific. As with the other U-Prove protocol participants, this is simply a role that can be taken by other participants. As an example, the revocation authority might be the issuer, or a separate entity shared by many issuers. Different issuers can share the revocation infrastructure by encoding the same user identifier into U-Prove tokens. This can be achieved either by disclosing the certified identifier to secondary issuers at issuance time, or by hiding it.[2]

The resources section at the end of the paper provides links useful to the reader, including to the U-Prove Technology Overview, core and extensions specification, and SDKs.

Issuer-driven revocation

The first model consists of revoking user identifiers encoded into tokens by its issuer. In these mechanisms, a user can convince verifiers at presentation time that her token is still valid by proving that her (undisclosed) identifier does not appear on the revocation list managed by the revocation authority.

A U-Prove revocation list must be distributed to users and verifiers by the revocation authority. Just like a X.509 Certificate Revocation List (CRL),[1] the authenticity and origin of the list must be attested to prevent attackers from changing it.[3] This can be achieved in different ways, and applications must decide which one suits their needs best. As an example, the list could be obtained from a secure endpoint identified in the token, or it could be signed and distributed offline by the revocation authority. List format and refresh granularity is left to the system designer, as this is application specific.

Multiple negation proofs

Using the inequality extension[4] it is possible to prove that an attribute is not equal to a target value, known or unknown to the verifier. Using this feature, we can build a powerful revocation mechanism allowing a

[1] Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. http://tools.ietf.org/html/rfc5280.

[2] Indeed, attributes can be carried over from a token to another without disclosing their values to the issuer. See the U-Prove extensions paper for details.

[3] A revoked user could try to remove her identifier from the list to gain access to a resource, or an attacker could add a user's identifier to the list to prevent her from accessing the resource.

[4] http://research.microsoft.com/apps/pubs/?id=219673.

user to prove that her unique identifier encoded in the token does not appear on the revocation list without disclosing it. Assuming that the revocation list contains the identifiers of all revoked users $(r_1, r_2, \dots, r_n)$, then a non-revocation proof consists of proving that $(x \neq r_1, x \neq r_2, \dots, x \neq r_n)$ where x is the value of the identifier attribute. Figure 1 illustrates the following steps:

1) The issuer issues a token to the user, encoding her unique user identifier *uid*.
2) The revocation authority periodically updates and distributes the revocation list. (Alternatively, the user and verifier can download the latest list from a secure endpoint.) Values can be added to the list by the issuer or a set of issuers if they share user identifiers.
3) When presenting a token to a verifier, the user creates a proof that the undisclosed value *uid* does not appear on the current revocation list.

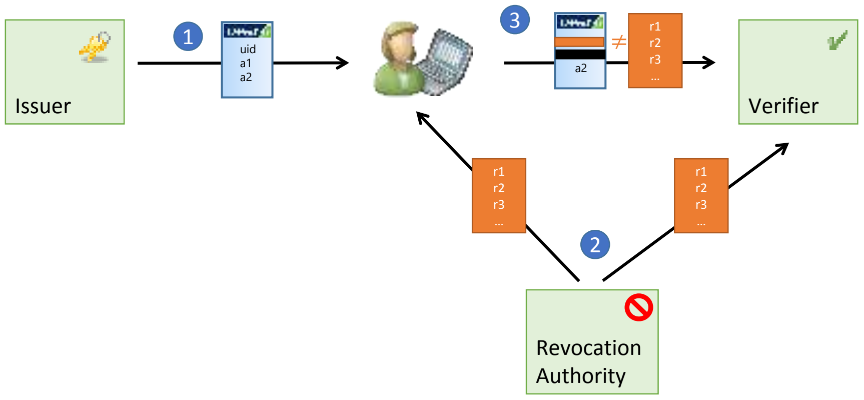

Figure 1: multiple negations

This technique can be implemented using the inequality proof type in the U-Prove extensions SDK, as illustrated in its `InequalityRevocationSample` sample.

The revocation proof's computation time and size is linear to the size of the revocation list, and therefore might be problematic for very large lists. Clever approaches have been proposed resulting in schemes that have square root[5] and logarithmic[6] complexities in the size of the revocation list, and to batch the validation of the proof.[7] We present next a mechanism designed to deal with very large revocation lists.

Accumulator

Another revocation approach makes use cryptographic *accumulators* in which revoked values are accumulated into a single aggregate value calculated by the revocation authority. The benefit of accumulator-based schemes is that they allow users to create non-revocation proof in constant size and time, once some accumulator pre-computations are made. These pre-computed values must be updated

[5] *A practical system for globally revoking the unlinkable pseudonyms of unknown users.* Stefan Brands, Liesje Demuynck, and Bart De Decker. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW472.pdf.
[6] *Zero-knowledge Argument for Polynomial Evaluation with Application to Blacklists*. Stephanie Bayer and Jens Groth. http://www0.cs.ucl.ac.uk/staff/J.Groth/PolynomialZK.pdf.
[7] Using, for example, techniques from Bellare, Garray, and Rabin's "Fast batch verification for modular exponentiation and digital signatures". http://cseweb.ucsd.edu/~mihir/papers/batch.pdf.

when the revocation list changes, which makes the scheme interesting if this occurs at well-defined intervals.

Most dynamic accumulator schemes use a type of algebraic construction called bilinear pairings.[8] Although pairings are popular in recent cryptographic research, they are seldom used in practice due to their maturity level and implementation complexity.

It is possible to build an accumulator scheme that uses a conventional finite field or elliptic curve construction (like the ones used in DSA and ECDSA, and in U-Prove) if we accept to involve the revocation authority during token verification. In this setting, the verifier validates the presented U-Prove tokens, but leaves the non-revocation proof validation to the revocation authority.[9]

Here are the details of the scheme illustrated in Figure 2:

1) The revocation authority generates its public parameters and secret key, and makes the public parameters available to users.
2) The user authenticates to the issuer and obtains U-Prove tokens encoding her unique identifier *uid*.
3) The revocation authority periodically updates the revocation list, and the user obtains non-revocation witnesses from the revocation authority.
4) The user presents a U-Prove token to the verifier, including a non-revocation proof (computed using the non-revocation witnesses). The verifier validates the presentation proof.
5) The verifier sends the non-revocation proof to the revocation authority that verifies that the undisclosed *uid* does not appear on the current revocation list.

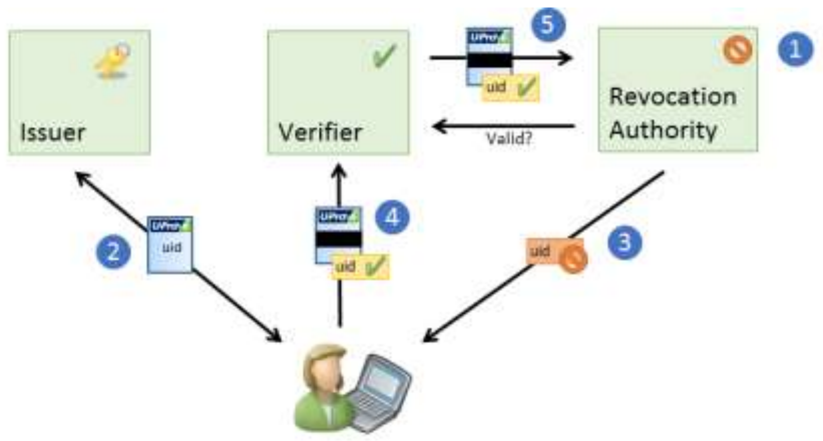

Figure 2: accumulator-based revocation list

[8] For an accumulator schemed designed to work with U-Prove, see Acar, Chow, and Nguyen's "Accumulators and U-Prove Revocation"; http://fc13.ifca.ai/proc/5-3.pdf.
[9] Much like an Online Certification Status Protocol (OCSP) call when using X.509 certificates. For more details, see RFC 6960: X.509 Internet Public Key Infrastructure - Online Certificate Status Protocol. http://tools.ietf.org/html/rfc6960.

This mechanism is available in the U-Prove extensions SDK as illustrated in the `AccumulatorRevocation-Sample` sample, and is documented in details in the U-Prove Designated-Verifier Accumulator Revocation Extension specification.[10]

This approach is significantly more efficient than the multiple not-proofs presented in the previous section. Indeed, the proof generation time is constant regardless of the size of the revocation list, compared to multiple negations approach which grows linearly with the size of the revocation list. As an example, Figure 3 shows the proof generation time for revocation lists containing from 10 to 1,000,000 items using the C# SDK. We observed the same results for the proof verification time. The revocation authority pays a cost to generate the accumulator and compute the user's witnesses on demand, but this is negligible even for very large revocation lists.

This method is the most efficient for the user. An alternative approach involves having the user calculating her own witnesses based on the list updates published by the revocation authority, but this would involve extra computation and storage costs for the user, and is not currently supported by the SDK.

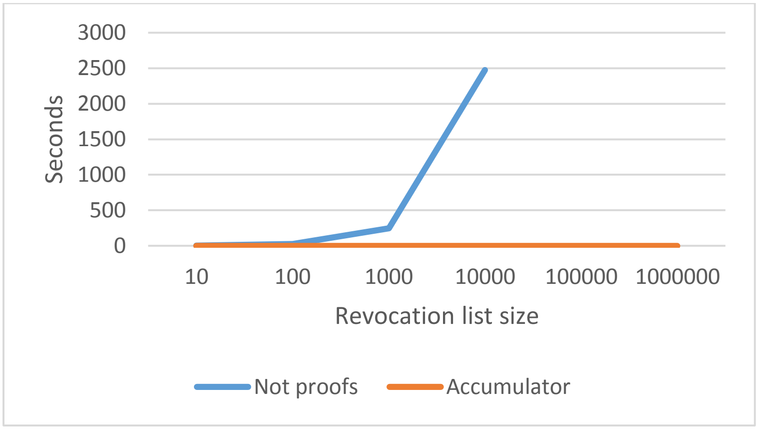

Figure 3: Revocation proof generation time for not-proofs vs. accumulator. Test performed on a HP Z820 machine with an Intel Xeon E5-2620 @ 2.00 GHz processor and 16GB of RAM. Execution of the Not-proofs program was stopped after the 10,000 list size since this approach is clearly not scalable past a few hundred items.

Device-driven revocation

In many scenarios, electronic identities are protected by some trusted device, such as a smartcard. Indeed, many countries, companies, and banks issue electronic authentication cards to their citizens, employees, and customers, respectively. Moreover, with the mobile revolution, many identity technologies made their way to the devices we carry in our pockets. For example, more and more credentials can be presented directly from mobile phones using cryptographic protocols.

In such settings, it makes sense to revoke the device itself, rather than the credential(s) it carries. Device-protected U-Prove tokens[11] provide an interesting way to enforce two-factor security, while minimizing the computations performed by the device. One revocation approach is for the revocation authority to

[10] http://research.microsoft.com/apps/pubs/?id=219671

[11] See the Technology Overview for more details about device-protected tokens.

distribute a list of revoked device identifiers, and let the trusted device enforce the revocation. This is illustrated in Figure 4:

1) The user obtains a device-protected token. The user's device contains a unique device identifier *uid*. This *uid* is never revealed to verifiers, otherwise this would contravene the privacy properties of the U-Prove technology.
2) At presentation time, the user retrieves the current device revocation list (either directly from the verifier, or from a common revocation authority).
3) The user passes the device revocation list to the device during the presentation protocol.[12] If the device sees its own identifier on the list, it refuses to collaborate with the user to present the token. If not, then presentation continues as usual.
4) The user sends the presentation proof (including the device response) to the verifier, which is convinced that the device, and therefore the user, is not revoked.

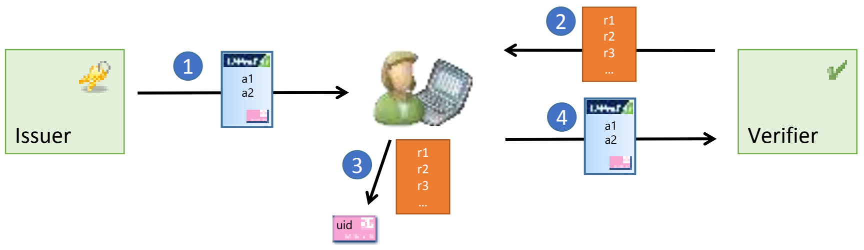

Figure 4: device-based revocation

Relying on a software-only second factor enhances authentication strength, but it is preferable to bind cryptographic keys to a hardware component. One interesting choice is the use of a Trusted Platform Module (TPM) chip. Fortunately, the TPM 2.0 specification[13] allows us to implement a U-Prove Device[14] therefore making it suitable to provide hardware protection of U-Prove tokens. While the TPM could not check the revocation list directly, this could be done by trusted firmware on the device.

This approach can be implemented using the C# SDK; see the `DeviceSample` sample for an example on how to use device-protected tokens.

Verifier-driven revocation

In this section, we explore revocation mechanisms driven by verifiers, allowing them to further control access to their resources, in addition to the revocation mechanisms offered by a central revocation authority.

[12] The list can be encoded in the device message. It cannot be modified by the user in transit, otherwise the resulting device response would be invalid and therefore the presentation would fail.
[13] https://www.trustedcomputinggroup.org/resources/tpm_library_specification.
[14] The TPM 2.0 **TPM2_Commit** and **TPM2_Sign** commands allow us to implement the Device's commitment and response computations, respectively.

These mechanisms are useful to prevent repeat visits from users who acted maliciously or contravened to the verifier’s terms of service. A group of verifiers can share their revocation information, allowing them to federate their revocation infrastructure.

Imagine, as an example, a set of e-book providers accepting university eID cards to provide free access to students. The university would manage its own (issuer-driven) revocation infrastructure, so an e-book provider would be convinced that visiting users are currently registered students with valid credentials. But on top of that, the group of e-book providers would want to make sure that students do not share online the e-books they obtained. To this end, e-book providers would request a contextual identifier when a user present a token. This identifier, which does not identify the user, would be embedded in the requested e-book. If, later, this identifier is found in an online sharing website, then it could be added to the providers’ revocation list, preventing the user from further accessing their collection. This “local” violation would not necessarily violate the school’s policy, so it wouldn’t be sufficient to revoke the student’s eID card (the user would still be a student at the school!), but these mechanism would protect the providers from further abuses from the user by locally revoking her tokens.

Using token identifiers

The simplest mechanism to invalidate a particular token is to revoke its token identifier. As explained in the U-Prove Technology Overview, the token identifier is a unique value calculated by hashing the token’s public key and issuer signature. This unique and unpredictable value is always the same when the token is presented, regardless of which attributes (if any) are presented to the verifier. Therefore, if a user only has one token, this technique can effectively be used to revoke users at the verifier(s), as illustrated in Figure 5:

1) The user obtains a token.
2) The user presents her token to the verifier, and misbehaves, prompting the verifier to add her token identifier to its local revocation list.
3) The next time the user visits the verifier presenting the same token, verification fails because her token identifier appears on the revocation list.

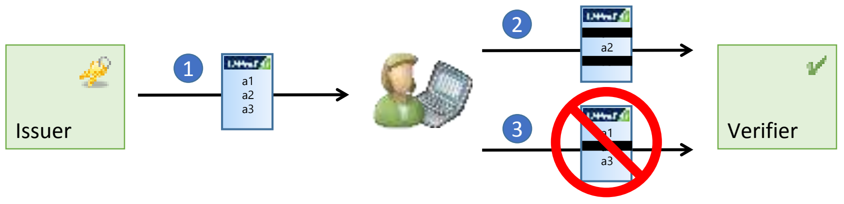

Figure 5: token identifier revocation

This mechanism can also be used by users to self-revoke their tokens. If a user’s tokens get compromised, then she can publish to a central revocation list the token identifiers of the affected tokens, and obtain new ones from the issuer. The verifiers, checking that revocation list, would therefore reject the compromised tokens even if they are used anonymously.

This approach does not work if a user has more than one token that could be presented to a verifier. In this case, the following approach could be of used instead.

Using scope-exclusive pseudonyms

When presenting a token, a user can present a scope-exclusive pseudonym derived from a unique attribute value and a verifier-provided context string. The resulting pseudonym will persist across sessions, even if different tokens are presented, as long as they encode the same identifiers. It is easy to see that this can be used for revocation purposes, as illustrated in Figure 6:

1) The user obtains a batch of tokens, each encoding the same attributes.
2) The user presents a token and a scope-exclusive pseudonym to the verifier. The user misbehaves, prompting the verifier to add her pseudonym to its local revocation list.
3) The next time the user visits presenting another token and the same scope-exclusive pseudonym, verification fails because her pseudonym appears on the revocation list.

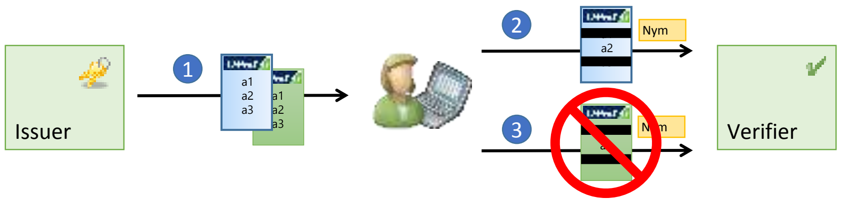

Figure 6: scope-exclusive pseudonyms as revocation identifier

Usage of the scope-exclusive pseudonym is illustrated in the `SoftwareOnlySample` sample of the C# SDK.

Other approaches

Different applications will prefer different revocation mechanisms. We discuss in this section alternative approaches of interest.

Short validity period

The simplest approach is to not support revocation, but instead limit the validity period of issued tokens, and force (non-revoked) users to periodically obtain new tokens. This is a common approach in many authentication protocols such as Kerberos, OAuth, and SAML; where credentials are obtained on demand and can be cached for a short amount of time (for example, a few hours). This approach is easily implemented and incur no extra cost during token presentation time, but limits the long-term usage of the tokens and imposes additional token issuance cost to both the issuer and the user.

Care must be taken however when encoding a validity period in a token, to avoid leaking information that could identify the user. Indeed, a fine-grained expiration time could be used as a correlation handle between the issuer and the verifier. Therefore, it is recommended to use neutral dates and times (for example, rounded values identifying a day, week, or month), or using an interval proof[15] to convince the verifier that the undisclosed expiration date is “larger than” the current time. The latter feature is available in the U-Prove Extensions SDK, as illustrated in the `RangeProofSample` sample.

Hybrid approach

One very interesting hybrid approach is to create a non-revocation proof service, akin to X.509’s Online Certificate Status Protocol (OSCP).[9] In OSCP, a verifier sends an identifier of the X.509 certificate to the

[15] http://research.microsoft.com/apps/pubs/?id=219674.

OSCP responder which in turn returns the certificate's validity to the verifier. This relieves the verifier from dealing with the revocation infrastructure. As a drawback, the responder learns that a specific user (identified by her certificate identifier) has visited a specific service provider.

This approach does not translate directly into the U-Prove world because, by design, user identifiers encoded into tokens are not disclosed to verifiers. What can be done, however, is to allow the user to contact the revocation authority beforehand, proving that she is not revoked, and obtaining the equivalent of a "valid" OSCP response to be presented to the verifier along with her U-Prove token. Assuming that the token contains a user identifier, as explained in the issuer-driven revocation section, then the user can use any of the aforementioned mechanisms to convince the revocation authority that her identifier/token is not revoked. The simplest case is to disclose her identifier and have the revocation authority issue a short-lived "not-revoked" token encoding the same user identifier. The span of the validity period range is application specific, but it should be neutral enough in order to protect the user (see discussion in the previous section).[16] Then, when presenting a U-Prove token to a verifier, a user would also present a non-revoked token, proving that the user's undisclosed identity has recently been validated by the revocation authority. This is illustrated in Figure 7:

1) The issuer issues a token to the user, encoding her *uid.*
2) The user periodically obtains a non-revoked token from the Revocation Authority, also encoding her *uid*. The user must present a valid U-Prove token, proving that she is not revoked (either by disclosing her *uid*, or by using one of the revocation mechanisms described in this paper).[17]
3) When presenting her token, the user also presents her most recent non-revoked token, proving that the *uid* attributes match in both, without disclosing them.

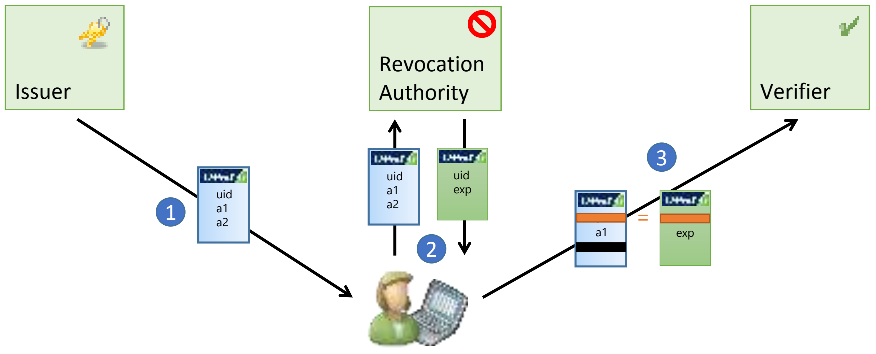

Figure 7: hybrid approach

To achieve this, an implementer must use the equality proof available in the U-Prove extensions SDK and as illustrated in the `CollaborativeIssuanceAndEqualitySample` sample, and documented in details in the U-Prove Equality Proof Extension specification.[18]

[16] One option would be to have all non-revoked token expire at midnight of the current day. Therefore, all such tokens would have the same expiration date, preventing information leakage about the user.
[17] The user should use different U-Prove tokens in step 2 and 3, to preserve unlinkability between the revocation authority and the verifiers.
[18] http://research.microsoft.com/apps/pubs/?id=219672.

Whitelisting

An alternate method to maintaining a list of revoked values is to maintain a list of currently valid values, often called a whitelist. The goal is then for the user to prove that her identifier is on the whitelist, without disclosing which value it is.

When the number of allowed identifiers is not too prohibitive, the set membership extension, available in the U-Prove extensions SDK and documented in the U-Prove Set Membership Proof Extension specification,[19] can be used to implement this mechanism.[20]

Resources

The following resources are useful to learn more about the features and techniques described in this document:

- U-Prove Technology Overview: http://research.microsoft.com/apps/pubs/default.aspx?id=166980
- U-Prove Extensions: http://research.microsoft.com/apps/pubs/default.aspx?id=226360
- U-Prove C# SDK: https://uprovecsharp.codeplex.com
- U-Prove Extensions C# SDK: http://bit.ly/uproveextensions

To learn more about the U-Prove technology, visit http://www.microsoft.com/u-prove.

[19] http://research.microsoft.com/apps/pubs/?id=219675.

[20] One proof per list element must be calculated, which affect both the proof size and computation time. Cryptographic accumulators, presented earlier in this paper, can also be used to implement whitelisting approaches for large lists.