Smart Signatures

A White Paper from Rebooting the Web of Trust

by Christopher Allen, Greg Maxwell, Peter Todd, Ryan Shea, Pieter Wuille, Joseph Bonneau, Joseph Poon, and Tyler Close

Abstract

Traditional cryptographic signature systems are based on strictly-defined authentication and authorization mechanisms that assume a single private key can be used to produce a given signature and that a single public key can be used to verify it. Given the evident limitations of this design, we propose an alternative with more powerful capabilities, based on the ability to explicitly outline and fully program conditions for verification. These conditions would then be used to determine when a signature or set of signatures can be considered valid.

Our inspiration for this authorization system is the bitcoin scripting language, where the authorization to spend funds is explicitly defined within a script, rather than being implicitly defined through the reference to an authorized public key. The largest benefit of explicit specification of authorization conditions is that the system is fully extensible, so new operations can be defined at any time, with the only limitation being the set of operations that the authorization interpreters understand.

1. Background

1.1 Traditional Message Signing and Verification

In traditional message signing systems, one generates two mathematically-linked keys, a public key and a private key, where the public key can be derived from the private key, but the reverse derivation cannot be performed.

To produce a signature for a given message, a signature generation function takes as inputs a private key and the message to be signed. A reader can verify the signature with a known public key, which is linked to that unknown private key. This is done by a signature verification function that uses the public key, the signature, and the signed message as inputs.

Traditional signature systems are very powerful on their own, as one can verify a signature without being able to produce it. However, these longstanding cryptographic systems are also limited in scope.

1.2 Bitcoin Scripting

Bitcoin contains a fairly advanced authorization system. Every transaction in bitcoin has a set of recipients, where each of the recipients is actually a script that outlines the conditions under which the coins can be spent at a future date. These scripts are the equivalent of challenges. Anyone who can meet the conditions outlined by the script is granted access to spend the funds.

These scripts can be very powerful and support various levels of complexity:

1. Scripts can require signatures that correspond to a given hash of an unknown public key. This is a traditional signature, referred to in bitcoin as a "pay-to-pubkey-hash" script.
2. Scripts can require a set of signatures to be provided that correspond to K out of N specified public keys. This is known as a multi-signature script.
3. Scripts can keep their actual redemption conditions secret and simply provide a hash of those conditions. At a later date, a redeemer can reveal the redemption conditions along with the signatures that allow it to meet those conditions — simultaneously providing the conditions and meeting them. If the conditions are met *and* the hash of the conditions matches the hash provided in the script, authorization is granted.

The following examples shows a standard bitcoin script:

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

Here, authorization would be granted if a user could provide a smart signature that contains a standard signature and a public key that together could be used as input for a bitcoin script-compatible verification function and that together produce a "true" output.

2. Proposal

2.1 Bitcoin's Scripting Outside of the Blockchain

Bitcoin's smart authorization mechanism can be used in other contexts. A wide variety of resources, from API endpoints to assets on a blockchain, can be protected by an authorization system where: (i) each resource links to script-encoded conditions for access; and (ii) each authorization request links to information intended to meet those conditions. Such a system would be a *smart signature* system.

2.2 Smart Verification

A smart signature system could begin with the standard bitcoin script referenced above:

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

This script would be embedded inside a smart certificate that is used outside of the blockchain context. When provided with a signature, certificate validation code would verify the authenticity of the script, as follows:

1. The user's signature is pushed onto the stack, producing `[SIG1]`
2. The user's public key is pushed onto the stack, producing `[SIG1][PUB1]`
3. The user's public key is duplicated (`OP_DUP`) on the stack, producing `[SIG1][PUB1][PUB2]`
4. The top public key on the stack is replaced by a hash of itself (`OP_HASH160`), producing `[SIG1][PUB1][PUBHASH1]`
5. The smart certificate's public key hash is pushed onto the stack, producing `[SIG1][PUB1][PUBHASH1][PUBHASH2]`
6. The public key hashes from the user and the smart certificate are checked for equality and then popped off the stack (`OP_EQUALVERIFY`), producing `[SIG1][PUB1]`
7. The user's signature is checked against the user's public key (`OP_CHECKSIG`), and if it is valid, both items are popped off the stack and replaced by "true", producing `[true]`
8. The certificate is validated

This process demonstrates the simplest sort of smart certificate, which is valid if its signature is valid. Basically it's the same as a self-signed certificate, except the rules for its validity are *inside* the certificate.

More complex scripts could replicate CA-style infrastructures, web-of-trust approaches, or use multisig scripts to create certain kinds of smart contracts. Examples of some of these results are included in the use cases, below.

2.3 Implications

Embedding the script inside the certificate ensures that the same method is used to evaluate it on all devices. Refactoring certificate policies into scripts that are executed and specifying a standard-tested virtual machine that executes those scripts may help avoid many of the common errors in certificate policy code in various apps and services.

The script inside a certificate can be inspected and evaluated. Like bitcoin today, some standard scripts may emerge that are trusted at a higher level than arbitrary written scripts.

3. Use Cases

At minimum smart signatures should support existing trust models, including self-signed certificates, CA-style certificates, and PGP-style key validation. In addition, they need to demonstrate flexibility for uses cases such as:

3.1 Short-term Delegation

- Bob has a key for website authentication, but is going on vacation. He wants website sysadmin Alice to be able to sign on his behalf while he's gone. Alice should be able to get into Bob's servers for a week, but things should revert to normal when Bob returns.
- Carol has a very secure key for signing her email. She wants to give her phone the temporary ability to sign emails with that key for the next week.

3.2 Limited Delegation

- Dan, Dana, Erin, Frances, and Frank are in charge of the development branch of a software project. Project leader Alice wants to give them a key that lets them release development versions of the project without allowing them to release stable versions.

3.3 Unbundling Delegation

- Carol uses a revocation service. She wants to give them the ability to revoke her key or her previously signed certificates without letting them authenticate keys.

3.4 Complex Delegation

- CFO Augustus wants to delegate authority to a department to issue a set amount of new bonds, but no more than the set amount, and with an interest rate within certain specifications.

- When Dan, Dana, Erin, Frances, and Frank release a development release, 3-of-5 of them need to sign the release. Further, Dan has a hardware token and wants to sign with 2-of-2 keys, where one key is stored on his hardware token and one is stored on his desktop computer.

4. Implementation Status

At this point, self-validating certificates only exist as a rough "on the napkin" proposal: neither a specification nor a proof-of-concept has been created. The next step may be to create a proof-of-concept prototype before focusing too much on a detailed specification.

4.1 Implementation Concepts

Some possible implementation concepts:

- **Schnorr Signatures & Script2 –** Many of the more complex signature delegations that we wish to offer (in particular Tree Signatures) require implementation of Schnorr Signatures, which bitcoin's script language does not currently offer. Blockstream offers a new library to support Schnorr under curve secp256k1 and is working on Script2, an upgrade of bitcoin's script, to support it.
- **Merkelized Abstract Syntax Tree (MAST)** - MAST is a cryptographically committed version of an abstract syntax tree that naturally maps to Lisp-like syntax and semantics. MAST-based Tree Signatures allow for multisig scripts with sophisticated AND/OR operations, which can be supported using bitcoin's script language by adding a single op-code: `OP_CAT`.
- **Pruning** - Unexecuted branches of the MAST can be pruned away and replaced by hashes. The verifier knows that the code was executed correctly while code not executed isn't relevant. This improves privacy.
- **Delegation via signed code** - To delegate control to another party, sign a secondary MAST that implements additional checks, such as a secondary public key and have the master MAST execute that code if the signature passes.
- **Upgrades** - We can add an operation that looks like a `no-op`. By definition this means "There are rules here that I don't know how to enforce"; systems have to treat that as a failure. Then we can add meaning to that operand in an upgrade. Once a supermajority is updated to enforces the rules, everyone sees the new op and can validate the script.

5. Open Questions

- **Static Context & Run Context** - In order to perform a number of use cases, the virtual machine may need to provide a context to a script and some way to parse that context. In the above example the `[SIG1][PUB1]` was provided to the script; in bitcoin, this would come from the output script of the previous transaction, but smart signatures don't have a parallel construct. Other examples abound: a certificate authority-like script needs to know which domain a child script is approving access to (an internal static context), which domain is actually being accessed (an external static context), and even the content of a web page itself (a run context), and then it must compare these values. What context is required? What additional script operands may be needed to parse and evaluate context given the limitations of a single Forth-like stack?

- **Asynchronous Oracles** - A number of use cases may require connecting to an third-party oracle to evaluate certain conditions such as a proof-of-existence as of a certain date, proof-of-uniqueness, specific financial information, or revocation status. It could be that these oracles are pre-fetched and added to the script's context before execution. If so, how does a script tell the virtual machine what to pre-fetch? It could be that certain oracles are distributed on DHT or blockchain and thus are always static. In any case, any asynchrony has security implications including the possibility of denial of service. How do we minimize these risks?
- **Revocation** - There was some discussion by the team about how to do revocation. Revocation is not as simple as a failed authentication; revoking a key is distinct from all other signature operations as it implies a desire for proof of non-revocation. It could be that revocation scripts need to be separate and distinct from a validation script. Possibly a proof-of-publication oracle can support this by providing a signature attesting to the fact that a specific revocation does not exist as of some timestamp. Finally, we could only use short-lived keys and not rely on revocation at all.
- **Hierarchical Deterministic Keys** - A number of use cases (in particular short-lived keys) may require scripts to evaluate validity of HD keys. Can we do this securely in script? What additional operands may be required?
- **Choice of Language** - The team has focused on a variant of bitcoin's Forth-like scripting language. The advantage here is that we can leverage the security reviews and understanding of a widely deployed existing system. However, at some point if we add enough features to the language that are unique to smart signatures, the benefits of being derived from bitcoin's script become less valuable. Other language approaches may offer superior features without compromising security, or offer superior tools. This team agrees that a simplified language should be a requirement, but where to draw the line is unclear.

Additional Credits

Lead Paper Editors: *Christopher Allen, Ryan Shea*

About Rebooting the Web of Trust

This paper was produced as part of the ***Rebooting the Web of Trust*** *design workshop. On November 3rd and 4th 2015, over 40 tech visionaries came together in San Francisco, California to talk about the future of decentralized trust on the internet with the goal of writing 3-5 white papers and specs. This is one of them.*

Workshop Sponsors: *Respect Network, PricewaterhouseCoopers, Open Identity Exchange, and Alacrity Software*

Workshop Producer: *Christopher Allen*

Workshop Facilitators: *Christopher Allen and Brian Weller with graphic facilitation by Sonia Sawhney and additional paper editorial & layout by Shannon Appelcline*

What's Next?

The design workshop and this paper are just starting points for Rebooting the Web of Trust. If you have any comments, thoughts, or expansions on this paper, please post them to our GitHub issues page: http://bit.ly/weboftrust-issues. We are also planning for more gatherings on this topic in the near future, with the object being to have something notable ready for release on the 25th anniversary of PGP, in July 2016. If you'd like to be involved or would like to help sponsor these events, email:

ChristopherA@LifeWithAlacrity.com