Dex: Deterministic Predicate Expressions for Smarter Signatures

©2016 Peter Todd, with contributions from Christopher Allen.
Commissioned by Christopher Allen of Alacrity Software to further work on Smart Signatures.

While substantial effort has gone into PKI, WoT, "blockchain", and other schemes to better map human identities meaningful identities to public keys, very little thought has gone into the keys themselves. Bitcoin's scripting system is a rare exception, showing how we can also have a more flexible and useful notion of a cryptographic identity, with flexible validation rules set in advance with evaluatable code. For identity systems separating the tasks of mapping human-relevant identity to cryptographic identity, and managing the cryptographic identity itself, has the potential to simplify both layers, and enable cryptographic identity innovation by removing the need to update all signature validators for every signing-side innovation.

> Predicates: that which is affirmed or denied

Dex is a deterministic expression standard that builds on initial exploration and requirements definition work done by the Smart Signatures group at the Nov 2015 Rebooting The Web of Trust conference. This allows cryptographic identities, and even protocol definitions, to be specified in terms of predicate expressions that evaluate to either true or false. Dex is deterministic, so for a specific environment and signature an expression will always evaluate to the same result; Dex aims to be usable in consensus-critical environments and draws on the author's experience working on the consensus-critical Bitcoin protocol and Bitcoin Core codebase.

Requirements

Before we go directly into how expressions are evaluated, it's important to remember *why* they're evaluated: we're trying to prove what the expression evaluates to efficiently. In short, Alice is trying to prove to Bob that expression <foo> returns <bar>. Bob wants to be able to evaluate that proof efficiently with as little bandwidth, memory, and CPU as possible; he does not care how much work Alice had to do to create that proof. This is completely unlike standard programming languages, which usually have efficiency of computation as a goal. Thus our requirements are, in order of importance:

1. Determinism --- Expression evaluation must be 100% deterministic across multiple platforms.

2. Efficient Validation --- Determining the result of an expression, evaluated against the specified arguments, must be cheap.

3. Bounded --- Attackers must not be able to create expressions that exceed application-appropriate memory and CPU limits; enforcement of these limits must itself be deterministic.

4. Understandable --- Humans examining an expression - potentially created by an adversary! - should have a reasonable chance at understanding what the expression is doing. This implies that raw expressions be relatively high level.

Mereklized S-expressions

We follow the Lisp tradition and represent both expressions and data with merkelized (hashed) S-expressions (msexprs) defined recursively from atoms such as numbers, strings, and symbols, and cons cells. Being hashed, all msexprs have a unique digest with each unique msexpr mapping to a unique digest. As values may be repreated, msexprs are directed acyclic graphs.

It's often the case that not all of a particular expression is needed to prove a particular result. For example, suppose you wish to determine what the following expression evaluates to:

```
(or (or (or true true)
        (or true true))
    (or (or true true)
        (or true true)))
```

Evaluation of ``or'' short-circuits, so we don't actually need the entire expression to know it evaluates to true. This allows Alice to prove to Bob that the above returned true, even if she only gives him the following:

```
(or (or (or true true)
        #<digest1>)
    #<digest2>)
```

We call this "pruning" and will use it extensively.

Evaluation

Atoms such as numbers, booleans, symbols, etc. are self-evaluating, and evaluate to themselves regardless of their argument.

How a cons cells is evaluated depends on the car (first item) in the cell. If the car is a symbol for a built-in function, such as the arithmetic operators or hashing functions, the cdr (second item) is used as the argument to the operator. For example, here's an expression that adds two numbers:

```
(+ 1 1) => 2
```

The apply operator has the important role of performing argument substitution. It takes two expressions as arguments, and substitutes all instances of the ``arg'' symbol in the first expression with the second expression. Here's it in use:

```
(apply (+ 1 arg) 1) => (+ 1 1)
```

While apply is rarely needed directly, it's used implicityly when evaluating cons cells whose car is an expression: the cdr of the cell is applied to the car, and the result is evaluated, acting as a function call.

Trivial Message Signing Protocol

Let's make use of expression evaluation to define a simple protocol for verifying that messages are authentic. We'll define

a message as "authentic" if an identity predicate expression evaluates to true; basically we're using a predicate expression to define a cryptographic identity, rather than using a pubkey directly.

We'll need a built-in function for hashing messages:

```
(sha256 <data>) => bytes
```

As well as a function for validating signatures:

```
(sig_valid <pubkey> <sig> <hash>) => boolean
```

A message is considered valid if the persistant identity expression returns true, when evaluated against the message and the signature expression:

```
(persist_id <msg> <sig_msexpr>) => boolean
```

The simplest persistant identity expression is to just just a single keypair to validate signatures:

```
(sig_valid <pubkey> (cdr argm) (sha256 (car argm)))
```

So how does that work? Let's look at how the expression evaluates in the case of a valid message. For the above expression, this means that the signature expression has been set to a valid signature. First the special symbol arg evaluate to the argument expression given to the validation expression:

```
(sig_valid <pubkey> (cdr '(<msg> <sig>)) (sha256 (car '(<msg> <sig>))))
```

Next the car's and cdr's evaluate, extracting the specific arguments needed:

```
(sig_valid <pubkey> <sig> (sha256 <msg>))
```

The message is hashed:

```
(sig_valid <pubkey> <sig> <SHA256(msg)>)
```

And finally `sig_valid` and its arguments evaluates to true:

```
True
```

Multisig

With just a single key our identity expression has a single point of failure. Let's change it to require 2-of-2 signatures, allowing those two keys to be placed on separate, isolated, machines for greater security. Our signature expression will now be `(<sig1> <sig2>)`, and our validation expression:

```
(and (sig_valid <pubkey1> (car (cdr argm)) (sha256 (car argm)))
     (sig_valid <pubkey2> (cdr (cdr argm)) (sha256 (car argm))))
```

For the expression to be true, both signatures must be valid. While at first glace it would appear that a number of calculations are duplicated, this doesn't need to be true: expression evaluation is strictly deterministic, so results can be

cached and reused; expressions are directed acyclic graphs, not trees.

Temporary Delegation

What if we want to temporarily delegate message signing authority? We can do this by signing a sub-expression. First we'll introduce a new primitive that evaluates to the digest of the specified expression:

```
(digest <expr>) => bytes
```

Second we'll introduce the `eval` operator, so we can evaluate expressions:

```
(eval <expr>) => <evaluated expr>
```

We need to make the time available to the identity expression, however if we make the current time available, we'd no longer have a deterministic signature. So instead we change our validity protocol to expect signatures to contain a *claimed* time, which is verified to be less than the actual time separately:

```
(persist_id <msg> (claimed_time <sig_expr>)) => boolean
```

Finally our signature now includes a sub-expression and a binding signature signing that sub-expression: `((<binding_sig> <sub_expr>) <sub_sig>)`. Our identity expression now validates that binding signature and evaluates the sub-expression against the sub-signature:

```
(and (sig_valid <master_pubkey> (car (car (cdr argm))) (SHA256 (cdr (car (cdr argm)))))
     (eval ((cdr (car (cdr argm)))) (cons argm (cdr (cdr argm)))))
```

That expression is rather hard to read with all the car's and cdr's used on argm directly; here's it with symbolic names instead:

```
(and (sig_valid <master_pubkey> <binding_sig> (SHA256 <delegated_expr>)
     (eval <delegated_expr> (cons argm <sig_expr>))
```

Finally, the delegated expression:

```
(and (sig_valid <delegated_pubkey> (SHA256 (cons claimed_time msg)))
     (< claimed_time delegation_expiry_time))
```

State and non-mathematical proof

Our predicate evaluation examples have been almost entirely stateless, with the one exception of the (indirect) use of the current time via the claimed time. While this is very mathematically pure, it's not sufficient for many real-world usecases like timestamping, revocation, key rotation, etc. Let's look at why.

; so what's intersting here, is that we have a repudiatable/non-repudiatable dichotomy here: anything that relies on timestamps can be done in a non-repudiatable model, while anything without timestamps is forced into a repudiatable model where validity can change after the fact

Timestamping --- Delegation Revisited

Our previous delegation example has an serious drawback: signatures made by the delegated authority are invalid after the expiry time is reached. We would like those signatures to remain valid, while simultaneously ensuring that after the

temporary delegation expires new signatures can't be created. We can solve this problem with timestamping.

Suppose Trudy runs a timestamping service, Trudy's Trusty Timestamps. The predicate expression for that service, , returns true when provided with a valid msexpr of the form ( ) where is the signature returned for the service when is timestamped.

This let's us remove the current time from the validity protocol:

```
(persist_id (<msg> <sig_msexpr>)) => boolean
```

Instead we're going to rely just on the timestamp, giving us a signature of the form:

```
(<binding_sig> <delegated_msexpr> <timestamp_time> <timestamp_sig> <sub_sig>)
```

Now our delegated expression can verify that the timestamp was valid, proving that the signature was created prior to when the delegation expired:

```
(and (sig_valid <delegated_pubkey> (SHA256 <msg>))
     (< timestamp_time delegation_expiry_time)
     (<ttt-predicate> <timestamp_time> (digest sub_sig) timestamp_sig))
```

Timestamp Latency

A problem with making temporary delegation rely on timestamps is the necessity of getting a timestamp for every bit of data signed. We have three basic options for our timestamps:

1. Individually signed --- A trusted third party signs each timestamped digest individually. While the timestamps are small, this has poor scalability and poor security due to large numbers of trusted timestamping servers required, each containing critical key material.

2. Signed merkle tree --- Multiple digests are collected in a merkle tree, with the tip of that tree timestamped by a trusted third party. Guardtime implements this scheme in a multi-level system with about 1 second latency per timestamp. Highly efficient, although at the cost of log2(n) sized timestamps.

3. Decentralized blockchain --- Similar to the signed merkle tree, but with a decentralized blockchain as the root of trust. Has the advantage of no trusted third party, at the cost of much higher latencies: minutes to as much as one or even two hours.

Anti-replay/Lazy-Commitments --- Key Rotation

If we want to rotate keys, making the previous key permantly unusable, we face a similar problem to the one we faced in time limited delegation: with math alone we can't prevent the old key from creating new signatures. With timestamping however we can timestamp our delegation signature and redelegate at regular intervals, prior to an expiry time.

What we want is a way of ensuring that some digital action can only happen once, an anti-replay device. A well-known example of this in action is Bitcoin: transaction outputs can be spent exactly once. Another way of looking at this is to think in terms of a commitment: where a commitment binds you to some future thing, like revealing a message with a particular digest, a lazy-commitment binds you to something you haven't chosen yet.

We can implement lazy-commitments in a variety of ways, such as a trusted oracle that promises to only sign once, or by taking advantage of the fact that a Bitcoin transaction output can only be spent once. Either way we can model the finalized commitment as a special form of signature:

```
(lazy_commit_valid <sig> <digest>) => bool
```

We can use the lazy-commitment in a very similar fashion to our delegation example above:

```
(and (and (sig_valid <master_pubkey> <binding_sig> (SHA256 <delegated_expr>))
          (lazy_commit_valid <lazy_sig> (digest <delegated_expr>)))
     (eval <delegated_expr> (cons argm <sig_expr>)))
```

If the delegated expression itself has a lazy commitment the process can be repeated indefinitely in a chain.

Revocation

With revocation we want to be able to make a key unusable, permanently preventing valid looking signatures from being created in the future. If we have key rotation we can of course do this by simply rotating to an invalid key (finalizing a lazy-commitment over an expression that always evaluates to false).

However an attacker can do this too, before we get a chance to revoke; what if we want revocation even under adversarial circumstances? OpenPGP supports this with special signed revocation packets, that are distributed on a best-effort basis; both attacker and defender can create a revocation message. We can model this as a function that returns true if a piece of data with a particular digest is known to exist; here's this in use:

```
(and <identiy_expr>
     (not (data_published <digest>)))
```

A practical implementation could be a service that acts as a publication platform for revocation messages, with each message actually being a signature that makes an expression return true; public blockchains like Bitcoin can do this. Regardless of what publication platform is used, our goal is to ensure that a revocation message - once published - will reliably get to all potential verifiers.