Contracts in Bitcoin

John Newbery, Chaincode Labs
April 8, 2020

Contracts and “Smart” Contracts

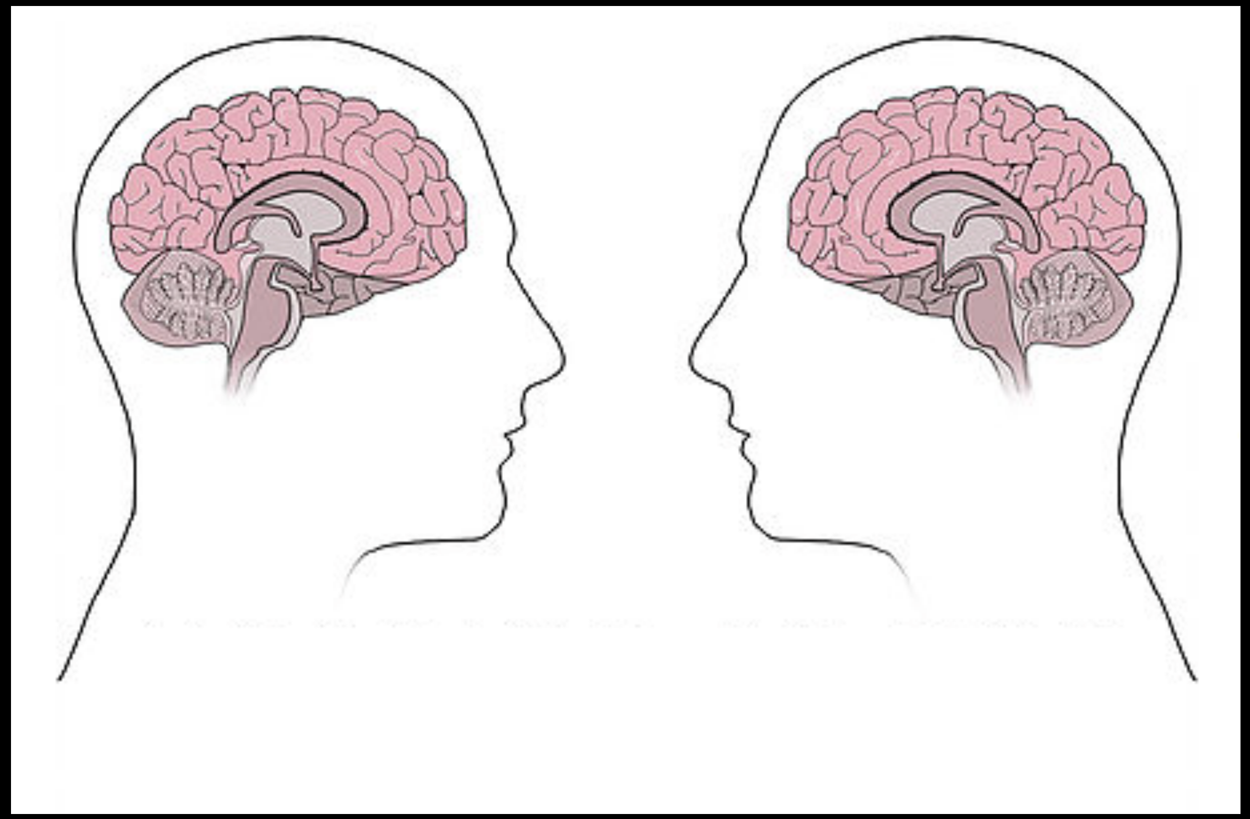
Script in Bitcoin

Contracts on a Public Blockchain

The History of Script in Bitcoin

The Future of Contracts in Bitcoin

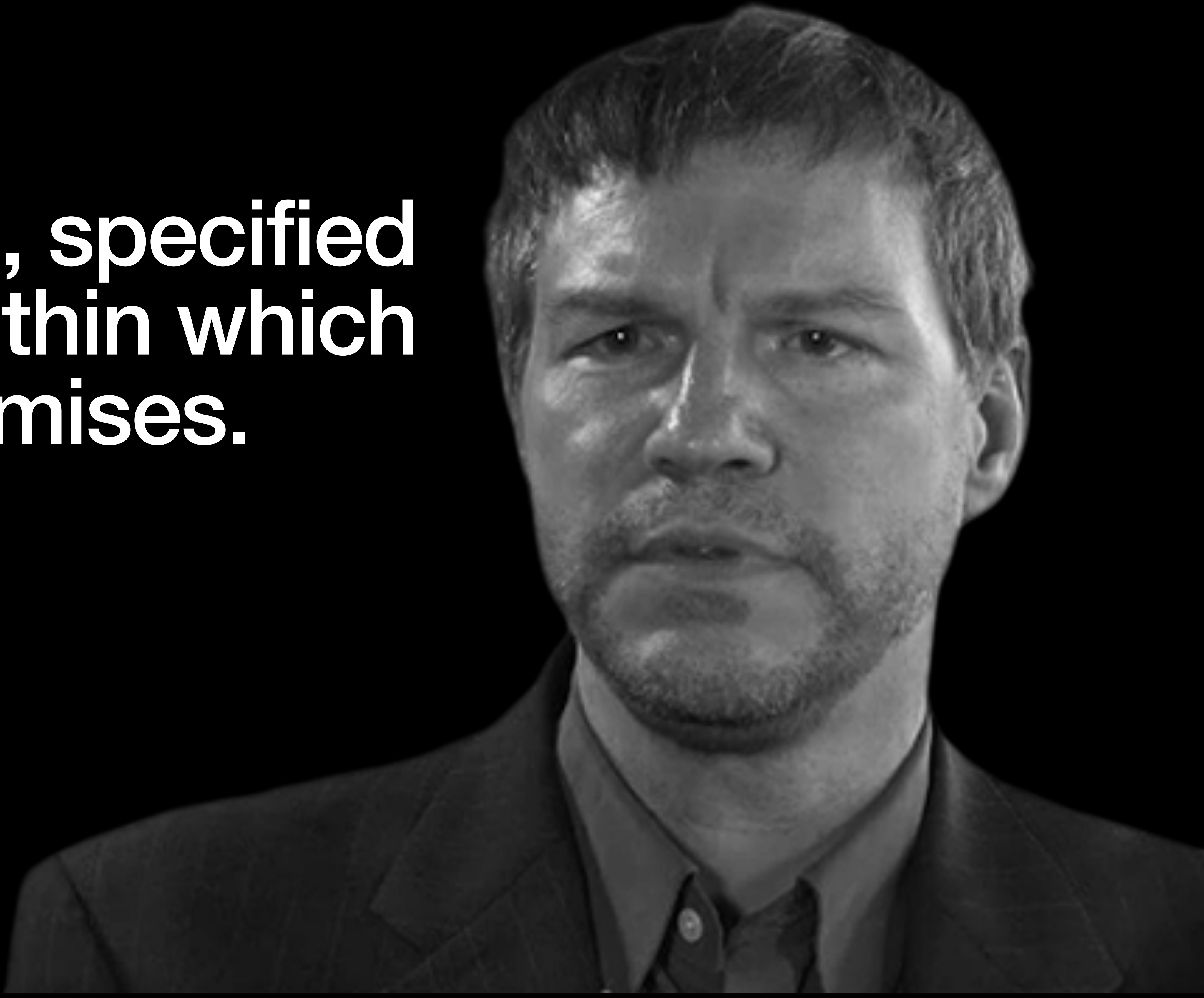
Contracts and “Smart” Contracts


Meeting of the minds

Binding

New institutions, and new ways to formalize the relationships that make up these institutions, are now made possible by the digital revolution. I call these new contracts "smart", because they are far more functional than their inanimate paper-based ancestors.

A smart contract is a set of promises, specified in digital form, including protocols within which the parties perform on these promises.



Smart Contracts: Building Blocks for Digital Markets (1996) by Nick Szabo

Objectives of contract design

Observability

Verifiability

Privacy

Enforceability

Script in Bitcoin

script 0/0

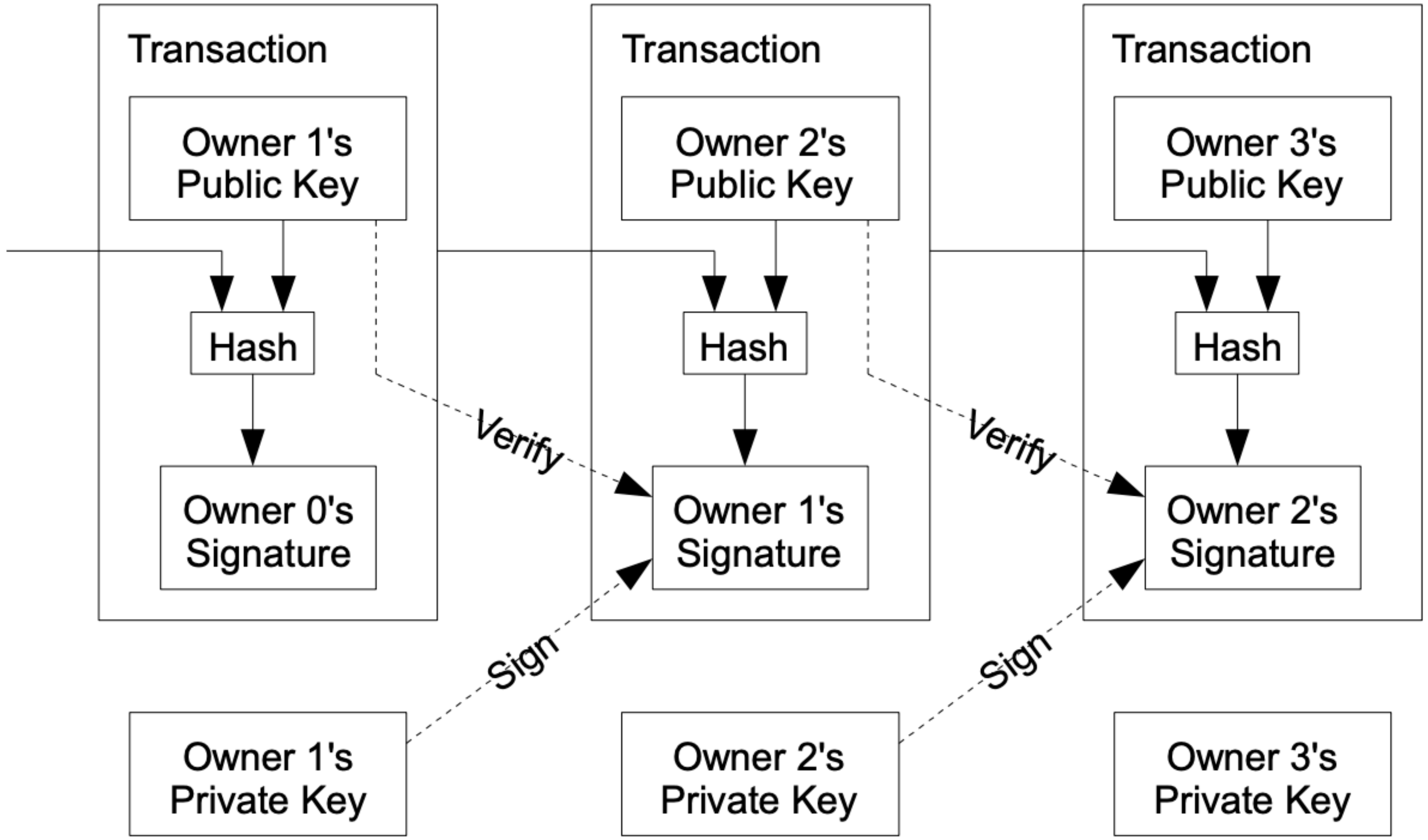
Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

Abstract. A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort

2. Transactions

We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.



```cpp
//
// Script is a stack machine (like Forth) that evaluates a predicate
// returning a bool indicating valid or not.  There are no loops.
//
#define stacktop(i)  (stack.at(stack.size()+(i)))
#define altstacktop(i)  (altstack.at(altstack.size()+(i)))

bool EvalScript(const CScript& script, const CTransaction& txTo, unsigned int nIn, int nHashType,
                vector<vector<unsigned char> >* pvStackRet)
{
    CAutoBN_CTX pctx;
    CScript::const_iterator pc = script.begin();
    CScript::const_iterator pend = script.end();
    CScript::const_iterator pbegincodehash = script.begin();
    vector<bool> vfExec;
    vector<valtype> stack;
    vector<valtype> altstack;
    if (pvStackRet)
        pvStackRet->clear();


    while (pc < pend)
    {
        bool fExec = !count(vfExec.begin(), vfExec.end(), false);

        //
        // Read instruction
        //
        opcodetype opcode;
        valtype vchPushValue;
        if (!script.GetOp(pc, opcode, vchPushValue))
            return false;

        if (fExec && opcode <= OP_PUSHDATA4)
            stack.push_back(vchPushValue);
        else if (fExec || (OP_IF <= opcode && opcode <= OP_ENDIF))
        switch (opcode)
        {
            //
            // Push value
            //
            case OP_1NEGATE:
            case OP_1:
            case OP_2:
            case OP_3:
            case OP_4:
            case OP_5:
            case OP_6:
            case OP_7:
```

```cpp

bool VerifySignature(const CTransaction& txFrom, const CTransaction& txTo, unsigned int nIn, int nHashType)
{
    assert(nIn < txTo.vin.size());
    const CTxIn& txin = txTo.vin[nIn];
    if (txin.prevout.n >= txFrom.vout.size())
        return false;
    const CTxOut& txout = txFrom.vout[txin.prevout.n];

    if (txin.prevout.hash != txFrom.GetHash())
        return false;

    return EvalScript(txin.scriptSig + CScript(OP_CODESEPARATOR) + txout.scriptPubKey, txTo, nIn, nHashType);
}
```

Once version 0.1 was released, the core design was set in stone for the rest of its lifetime. Because of that, I wanted to design it to support every possible transaction type I could think of.

The solution was script. The nodes only need to understand the transaction to the extent of evaluating whether the sender's conditions are met.

The script is actually a predicate. It's just an equation that evaluates to true or false. Predicate is a long and unfamiliar word so I called it script.

https://bitcointalk.org/index.php?topic=195.msg1611#msg1611 (2010) by Satoshi Nakamoto

Contracts on a Public Blockchain

The Good News

🔍 Observability

✓ Verifiability

🔓 Enforceability

The Bad News

Enforceability

Fungibility

Privacy

Scalability

Expressiveness

Contracts executed by explicitly published code are really only using the blockchain for one thing - to get an immutable ordering of what order the transactions happen in. All that they really care about is that their transaction is not reversed and not double spent.

All of the extra details of the contract execution can be done by things that are not blockchains.

Using chains for what they're good for (2017) by Andrew Poelstra

Post's Theorem

- Turing-complete languages define *Computably Enumerable* predicates
- In the 1930s, Emil Post defined an *Arithmetic Hierarchy*
    - Δ0 predicates have no *unbound quantifiers* (eg $\forall x < z \ \exists y < z$ s.t. $x + y = z$)
    - Σ1 predicates are those which have an *unbound quantifier* (eg $\exists x \in N$ s.t. $\forall y > z$, $x < y$)
- Post's Theorem states that computably enumerable predicates are identical to Σ1 predicates
- Validating a Σ1 predicate can *always* be reduced to validating a Δ0 predicate with a *witness*
- Evaluating a Δ0 predicate *always* terminates

In Σ1 thinking, I'm going to write a program and everyone on the blockchain is going to execute this program and everyone will do the same thing because they're running in the same environment.

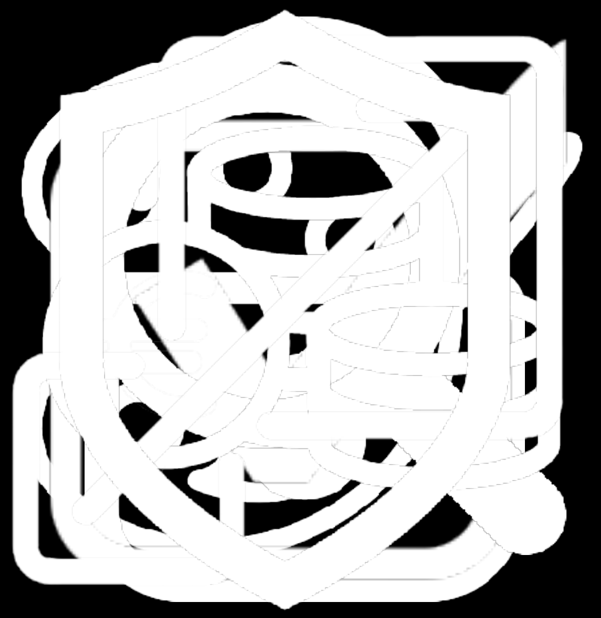
With Δ0 thinking, I'm going to run the program myself on my computer and generate some witness data and I'm going to have everyone only validate that witness data instead of running the entire program.

It's a change of attitude [...] that can be a lot more efficient, a lot more private and save a lot of time.

Post's Theorem and Blockchain Languages: A Short Course in the Theory of Computation (2017)
by Russell O'Connor

Computation

Verification



[illegible]

[illegible]

[illegible]

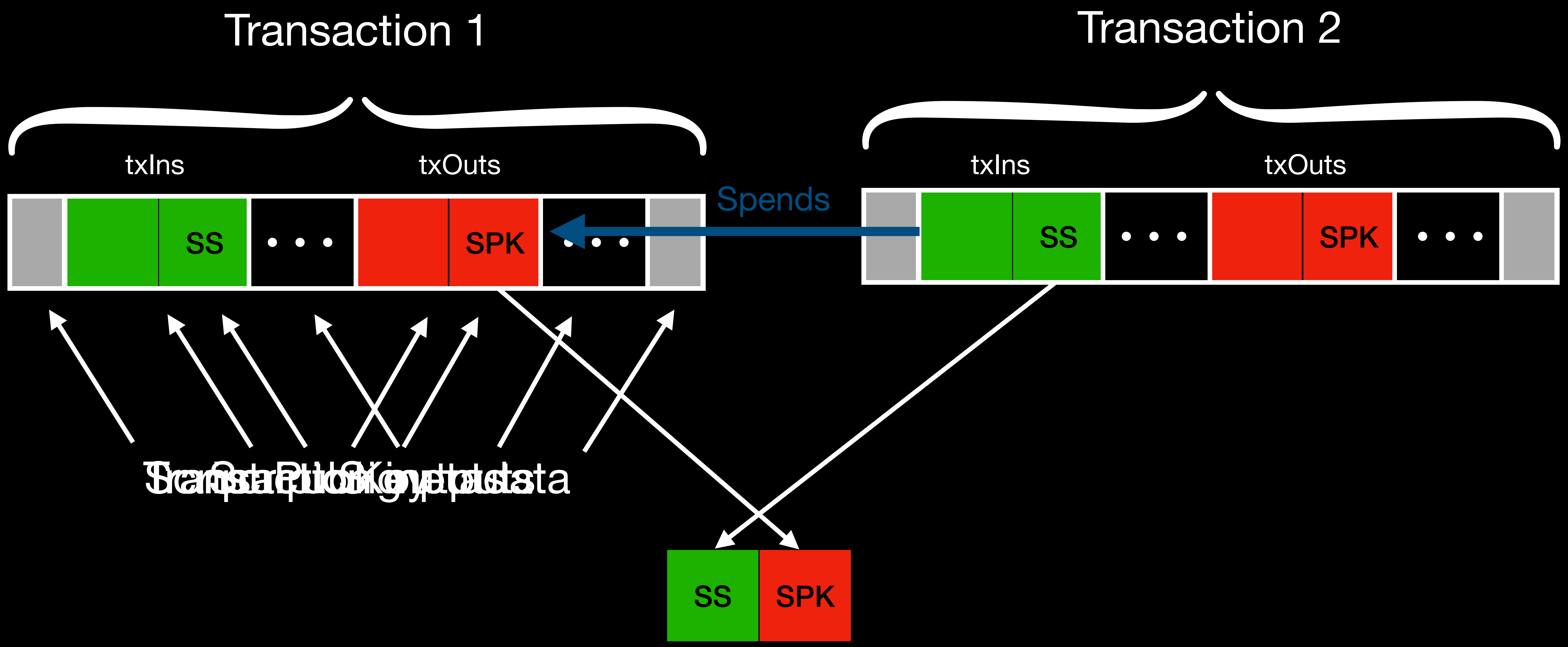
Is this mental model similar to conventional programming? No. But smart contracts aren't conventional programming, and blockchain isn't a conventional computing environment (how often does history change out from under most of your programs?).

These design elements make for a clean, simple system with predicable interactions. To the extent that they make some things "harder" they do so mostly by exposing their latent complexity that might otherwise be ignored.

https://www.reddit.com/r/Bitcoin/comments/4pdx6k/comparison_between_bitcoin_and_ethereums/d4k6n7d/ (2016) by Greg Maxwell

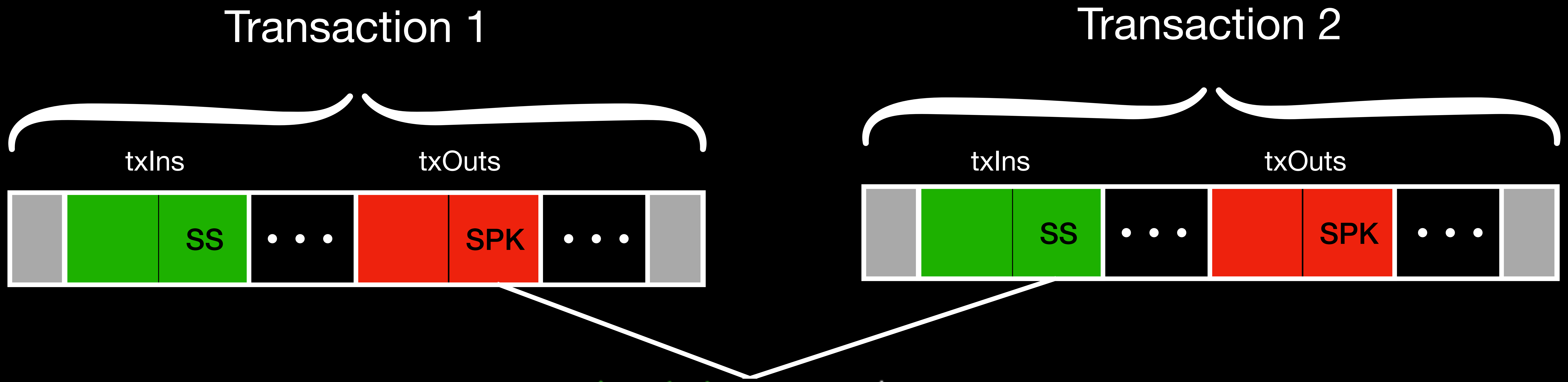
The History of Script in Bitcoin

2009 - V0.1



```
return EvalScript(txin.scriptSig + CScript(OP_CODESEPARATOR) + txout.scriptPubKey, txTo, nIn, nHashType);
```

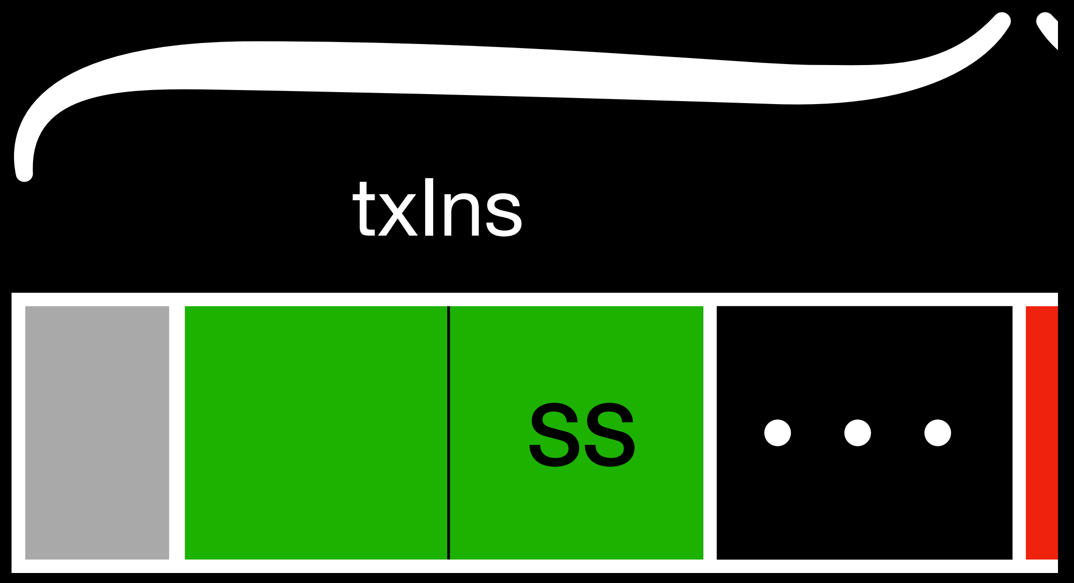
2010 - Fix bugs!



```
vector<vector<unsigned char> > stack;
if (!EvalScript(stack, scriptSig, txTo, nIn, nHashType))
    return false;
if (!EvalScript(stack, scriptPubKey, txTo, nIn, nHashType))
    return false;
if (stack.empty())
    return false;
return CastToBool(stack.back());
```

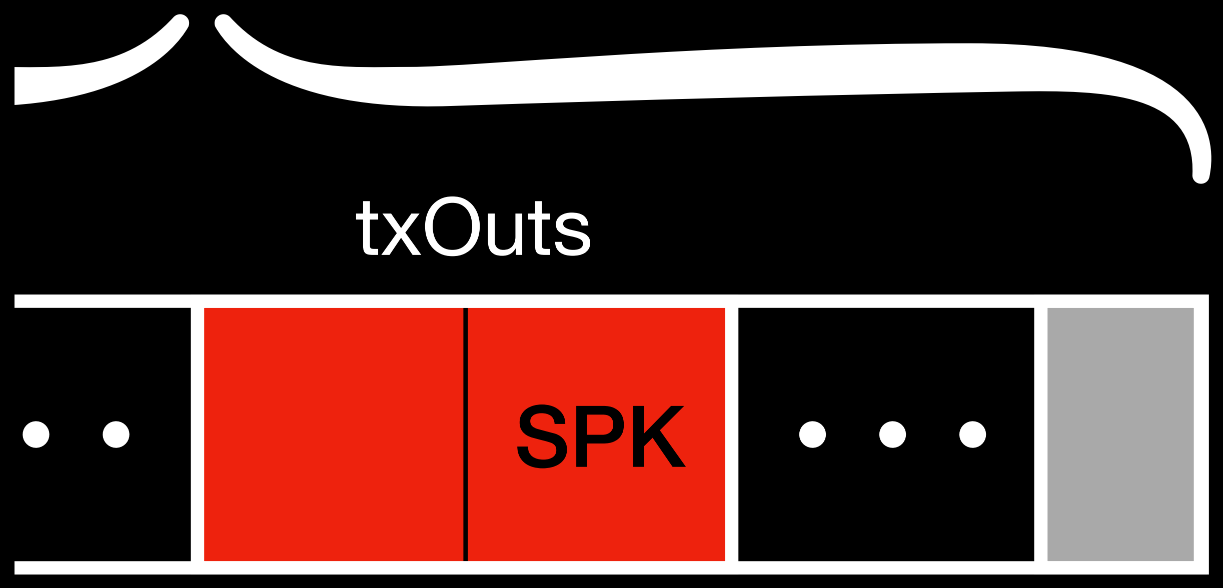
2012 - Pay To Script Hash

Transac



```
vector<vector<unsigned char> > stack, stackCopy;
if (!EvalScript(stack, scriptSig, txTo, nIn, nHashType))
    return false;
if (fValidatePayToScriptHash)
    stackCopy = stack;
if (!EvalScript(stack, scriptPubKey, txTo, nIn, nHashType))
    return false;
if (stack.empty())
    return false;

if (CastToBool(stack.back()) == false)
    return false;

// Additional validation for spend-to-script-hash transactions:
if (fValidatePayToScriptHash && scriptPubKey.IsPayToScriptHash())
{
    if (!scriptSig.IsPushOnly()) // scriptSig must be literals-only
        return false;            // or validation fails

    const valtype& pubKeySerialized = stackCopy.back();
    CScript pubKey2(pubKeySerialized.begin(), pubKeySerialized.end());
    popstack(stackCopy);

    if (!EvalScript(stackCopy, pubKey2, txTo, nIn, nHashType))
        return false;
    if (stackCopy.empty())
        return false;
    return CastToBool(stackCopy.back());
}

return true;
```

nsaction 2



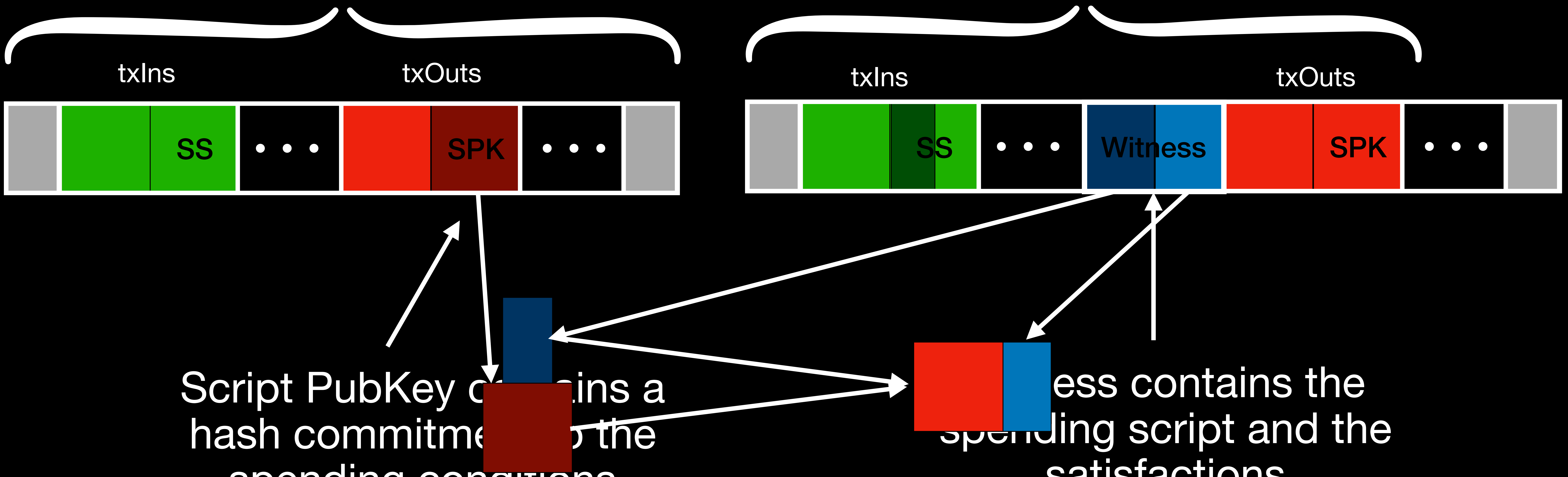
Script Pu
hash cor
spendi
C

ig contains the
script and the
isfactions
nput
s the

2016 - Segregated Witness

Transaction 1

Transaction 2



Script PubKey c ains a
hash commitme o the
spending conditions

ess contains the
sp ding script and the
satisfactions

Check that spending
script matches
commitment

Check that witness
satisfies the script

The Future of Contracts in Bitcoin

- Schnorr signatures and MuSig
- MAST - script trees
- Taproot and Graftroot
- Adaptor signatures

Questions?