SCRIPT

JOHN NEWBERY

@jfnewbery

github.com/jnewbery

WHAT THIS TALK WILL COVER

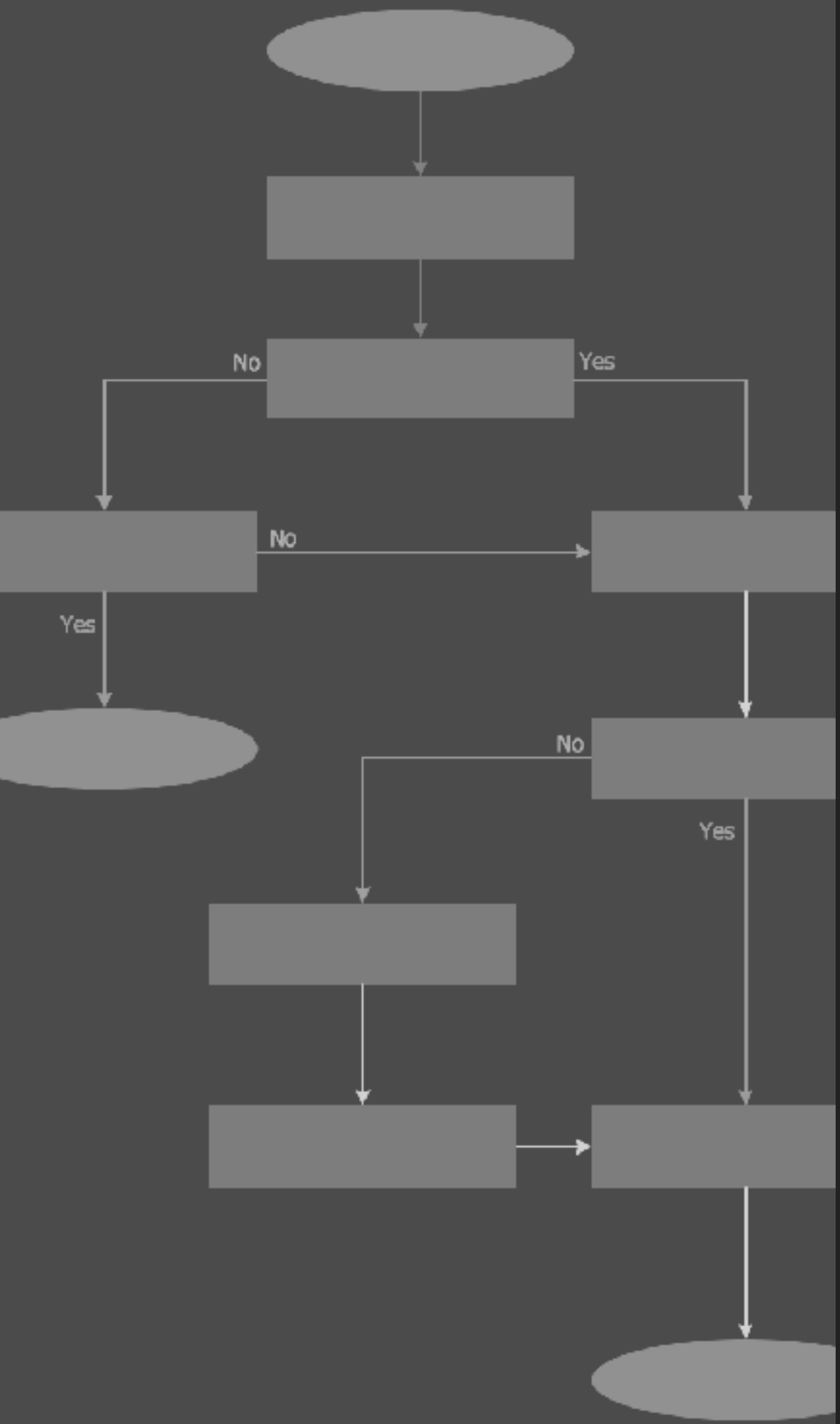
- Why we have SCRIPT
- The design philosophy of contracts on a blockchain
- A couple of SCRIPT examples

WHAT THIS TALK WON'T COVER

- A deep technical exploration of SCRIPT semantics
- An exhaustive description of common Bitcoin transactions

SCRIPT

- Why do we have SCRIPT?
- Locking and unlocking coins
- Pay-to-pubkey
- Multisig
- Computing -vs- Verifying



WHY SCRIPT?

WHY SCRIPT?

- A *chain of digital signatures* allows a digital coin to be transfered between users
- What if I want a coin to be spendable when 2-out-of-3 people sign?
- What if I want a coin to be spendable when someone knows a secret value (eg the pre-image to a hash digest)?
- What if I want a coin to be spendable after a certain time?
- ……

WHY SCRIPT?

- The Bitcoin whitepaper didn’t mention any ‘contracts’
- Satoshi added a generic scripting language that lets users to specify their own contracts
- SCRIPT may have been a late additional to the Bitcoin source code
- Early versions of SCRIPT were very buggy!

WHAT IS SCRIPT?

- A contract on Bitcoin is a **predicate**
- It takes inputs:
  - the transaction
  - additional data provided by the spender
- It returns True or False:
  - True: the transaction is valid
  - False: the transaction is invalid

WHAT IS SCRIPT?

- Contracts are implemented in Bitcoin as programs written in a language called SCRIPT
- SCRIPT is a stack-based language
- Each item in a script either:
  - pushes elements onto the stack; or
  - acts on element(s) in the stack
- At the end of execution, if the stack is non-empty and the top element is non-zero, then the script evaluates to True

LOCKING AND UNLOCKING COINS

LOCKING AND UNLOCKING

- A transaction output (txout) is *locked* with conditions under which it can be spent
- A transaction input (txin) refers to a previous txout and *unlocks* it by proving that it satisfies the txout's conditions
- The locking conditions are encoded in a **scriptPubKey**
- The unlocking proof is encoded in a **scriptSig**

EVALUATING SCIPTPUBKEY AND SCRIPTSIG

- Early versions of Bitcoin concatenated scriptSig and scriptPubKey and then ran the combined script
- This was broken - anyone could spend any coin!
- v0.3.8 of Bitcoin fixed this by running the scipts separately - first run scriptSig, leave the result on the stack, then run scriptPubKey
- (Note that scriptSig does not need to be a script - it is only used to place items on the stack)

EXAMPLE LOCKING CONDITIONS - P2PK

- The simplest scriptPubKey is called ‘Pay to pub key’ or P2PK
- The condition for spending a P2PK output is signing a message with the private key corresponding to the given public key
- The message that the spender must sign is (a part of) the transaction that spends the output

EXAMPLE LOCKING CONDITIONS - MULTISIG

- Multisig is used to require *k*-out-of-*n* parties to sign in order to spend an output
- The condition for spending a multisig output is signing a message with *k* of the private keys corresponding to the given *n* public keys
- Each signature signs the same message - (a part of) the transaction that spends the output

EXAMPLE LOCKING CONDITIONS - P2PKH

- Pay to pubkey hash (P2PKH) locks an output with the *hash digest* of a public key
- The condition for spending a P2PKH is providing:
  - a public key that hashes to the hash digest
  - a signature of a message with the private key corresponding to the given public key

EXAMPLE LOCKING CONDITIONS - P2SH

- Pay to script hash (P2SH) locks an output with the *hash digest* of any arbitrary script
- The condition for spending a P2SH is providing:
  - a SCRIPT that hashes to the hash digest
  - the data required to satisfy the locking conditions in that script

WHY P2SH?

- scriptPubKeys for P2SH are a (small) uniform size
- The sender does not need to know the spending conditions for what they're sending
- The receiver pays the fee for large or complex scripts
- A scriptPubKey can be encoded as a Bitcoin address, eg 3P14159f73E4gFr7JterCCQh9QjiTjiZrG

EXAMPLE LOCKING CONDITIONS - P2WPKH & P2WSH

- Segregated witness (BIP 141) introduced two new kinds of locking scripts:
  - Pay to witness public key hash (P2WPKH)
  - Pay to witness script hash (P2WSH)
- Key difference is that the data requiered to satisfy the conditions is carried in a separate structure called the 'witness'

PAY-TO-PUBKEY

PAY TO PUBLIC KEY

- The scriptPubKey contains the public key (33 bytes for compressed) and the OP_CHECKSIG opcode (1 byte)
- The scriptSig contains just a signature (~71 bytes)

BEFORE EXECUTION

scriptPubKey

scriptSig

stack

<PUBKEY>

OP_CHECKSIG

<SIG>

AFTER SCRIPTSIG EXECUTION

scriptPubKey

<PUBKEY>

OP_CHECKSIG

scriptSig

stack

<SIG>

SCRIPTPUBKEY EXECUTION – 1

SCRIPTPUBKEY EXECUTION - 2

AFTER SCRIPTPUBKEY EXECUTION

scriptPubKey | scriptSig | stack

MULTISIG

MULTISIG

- For a k-of-n multisig, the scriptPubKey contains:
  - the number k (1 byte)
  - all n public keys (33 bytes each for compressed)
  - the number n (1 byte)
  - the OP_CHECKMULTISIG opcode (1 byte)
- The scriptSig contains:
  - a dummy 0 byte (1 byte)
  - k signatures (~71 bytes each)

BEFORE EXECUTION

scriptPubKey	scriptSig	stack
2		
<PUBKEY 1>		
<PUBKEY 2>		
<PUBKEY 3>	0	
3	<SIG 1>	
OP_CHECKMULTISIG	<SIG 2>	

AFTER SCRIPTSIG EXECUTION

scriptPubKey

2

<PUBKEY 1>

<PUBKEY 2>

<PUBKEY 3>

3

OP_CHECKMULTISIG

scriptSig

stack

<SIG 2>

<SIG 1>

0

SCRIPTPUBKEY EXECUTION - 1

scriptPubKey

scriptSig

stack

3

<PUBKEY 3>

<PUBKEY 2>

<PUBKEY 1>

2

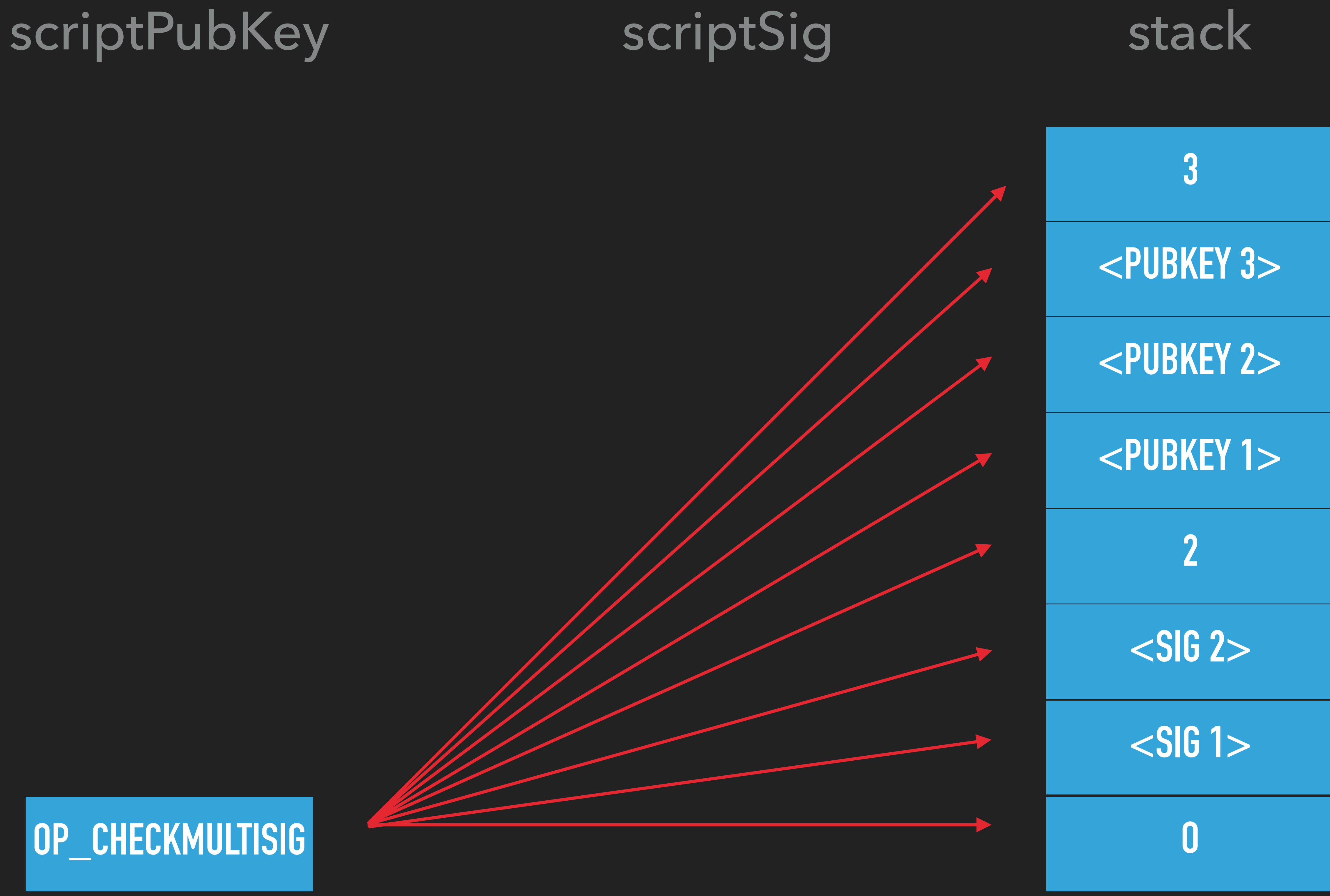
<SIG 2>

<SIG 1>

0

OP_CHECKMULTISIG

SCRIPTPUBKEY EXECUTION - 2



AFTER SCRIPTPUBKEY EXECUTION

scriptPubKey	scriptSig	stack
		1

COMPUTING
-VS-
VERIFYING

COMPUTING AND VERIFYING

- A contract is a predicate
- Bitcoin nodes are only interested in whether a contract evaluates to true, not the details of *how* it evaluates
- Bitcoin uses SCRIPT, which is interpreted and executed by every node
- Bitcoin uses *computation*, but it's really only interested in *verification*

COMPUTING AND VERIFYING (SCALING)

- Adding more computation workload to contract execution does not scale
- Verification is much easier and more scalable than computation
- At the limit, a blockchain could use zero-knowledge proofs instead of script execution
- At the margin, there are lots of technologies that can improve scalabilty by only committing minimal data to the blockchain

SCALING CONTRACTS

- Only reveal spending conditions at time of spend
  => P2SH or P2WSH
- Batch multiple payments into one on-chain commitment
  => layer 2 (eg lightning)
- Only reveal the branch of the contract that was executed
  => MAST, Taproot
- In the best case where everyone agrees, only broadcast a single (threshold) signature
  => Taproot, Graftroot
- Combine multiple signatures into a single signature
  => threshold signatures
- Embed additional conditions/committments invisibly into digital signatures
  => adaptor signatures and scriptless scripts

SCALING AND PRIVACY/FUNGIBILITY

- It’s no coincidence that these scaling techniques are also good for privacy and fungibility:
  - less data on the blockchain => better privacy
  - more uniform transactions => better fungibility

IN CONCLUSION

- A Bitcoin output can be locked with a contract
- A contract is a predicate - it takes the transaction and additional data provided by the spender and returns True or False
- Bitcoin uses SCRIPT to encode contracts and the witness data
- SCRIPT is a stack-based language that executes on all nodes
- A blockchain is for **verifying**, not for **computing**