Models for Bitcoin smart contracts

Massimo Bartoletti

University of Cagliari

Polimi Fintech Journey 2018

joint work with...

Nicola Atzei
Tiziana Cimoli
Stefano Lande
(@ UniCA)

Roberto Zunino (@ UniTN)

Smart contracts

> A **smart contract** is a computerized protocol that executes the terms of a contract [...] to minimize the need for trusted intermediaries and transaction costs.
>
> *Nick Szabo*, 1994

Smart contract platforms

- Ethereum
- Hyperledger
- Rootstock, Tezos, ... (?)

Smart contracts

> A **smart contract** is a computerized protocol that executes the terms of a contract [...] to minimize the need for trusted intermediaries and transaction costs.
>
> *Nick Szabo*, 1994

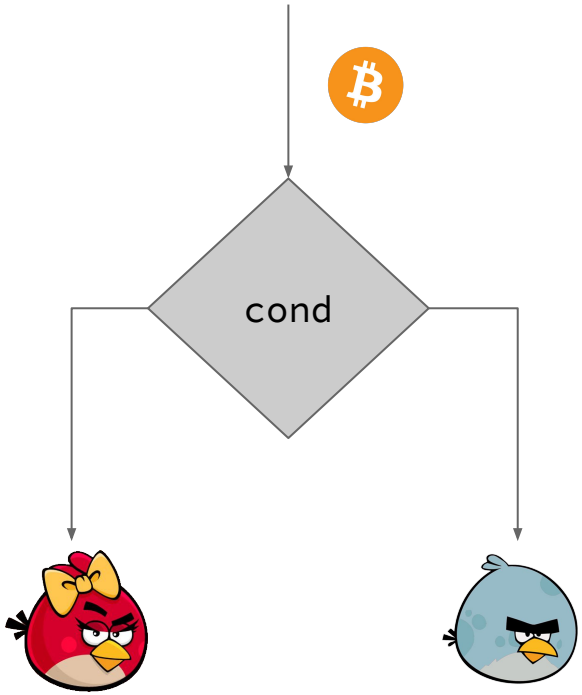
Smart contract platforms

Why smart contracts on Bitcoin?

- exchange BTC (rather than ETH)
- underlying guarantees (*theorems!*) on the security of the blockchain
  [Garay,Kiayias,Leonardos, EUROCRYPT'15] [...]
- bugs in Solidity code are difficult to spot (*recall the DAO and the Parity attacks!*)
- narrower attack surface (txs *vs.* EVM/Solidity)

Bitcoin smart contracts

Bitcoin contracts are cryptographic protocols to transfer BTC, executed respecting the consensus protocol of the blockchain.



Problems

- Bitcoin was not designed for smart contracts
  - ...still, many use cases exist
  - unclear expressiveness (off-chain protocols!)

- Low level scripting language
  - poorly documented
  - standards to adhere (minimal push, P2SH, ...)
  - subtleties require to inspect Bitcoin Core source code

- **no formal specification**

  ⇒ no automatic verification

Example 4: Using external state

Scripts are, by design, pure functions. They cannot poll external servers or import any state that may change as it would allow an attacker to outrun the block chain. What's more, the scripting language is extremely limited in what it can do. Fortunately, we can make transactions connected to the world in other ways.

Consider the example of an old man who wishes to give an inheritance to his grandson, either on the grandson's 18th birthday or when the man dies, whichever comes first.

To solve this, the man first sends the amount of the inheritance to himself so there is a single output of the right amount. Then he creates a transaction with a lock time of the grandson's 18th birthday that pays the coins to another key owned by the grandson, signs it, and gives it to him - but does not broadcast it. This takes care of the 18th birthday condition. If the date passes, the grandson broadcasts the transaction and claims the coins. He could do it before then, but it doesn't let him get the coins any earlier, and some nodes may choose to drop transactions in the memory pool with lock times far in the future.

The death condition is harder. As Bitcoin nodes cannot measure arbitrary conditions, we must rely on an *oracle*. An oracle is a server that has a keypair, and signs transactions on request when a user-provided expression evaluates to true.

Here is an example. The man creates a transaction spending his output, and sets the output to:

```
<hash> OP_DROP 2 <sons pubkey> <oracle pubkey> CHECKMULTISIG
```

This is the oracle script. It has an unusual form - it pushes data to the stack then immediately deletes it again. The pubkey is published on the oracle's website and is well-known. The hash is set to be the hash of the user-provided expression stating that he has died, written in a form the oracle knows how to evaluate. For example, it could be the hash of the string:

```
if (has_died('john smith', born_on=1950/01/02)) return (10.0, 1JxgRXEHBi86zYzHN2U4KMyRCg4LvwNUrp);
```

This little language is hypothetical, it'd be defined by the oracle and could be anything. The return value is an output: an amount of value and an address owned by the grandson.

Once more, the man creates this transaction but gives it directly to his grandson instead of broadcasting it. He also provides the expression that is hashed into the transaction and the name of the oracle that can unlock it.

It is used in the following algorithm:

1. The oracle accepts a measurement request. The request contains the user-provided expression, a copy of the output script, and a partially complete transaction provided by the user. Everything in this transaction is finished except for the scriptSig, which contains just one signature (the grandson's) - not enough to unlock the output.
2. The oracle checks the user-provided expression hashes to the value in the provided output script. If it doesn't, it returns an error.
3. The oracle evaluates the expression. If the result is not the destination address of the output, it returns an error.
4. Otherwise the oracle signs the transaction and returns the signature to the user. Note that when signing a Bitcoin transaction, the input script is set to the connected output script. The reason is that when OP_CHECKSIG runs, the script containing the opcode is put in the input being evaluated, _not_ the script containing the signature itself. The oracle has never seen the full output it is being asked to sign, but it doesn't have to. It knows the output script, its own public key, and the hash of the user-provided expression, which is everything it needs to check the output script and finish the transaction.
5. The user accepts the new signature, inserts it into the scriptSig and broadcasts the transaction.

Bitcoin contracts in literature

Pre-condition:

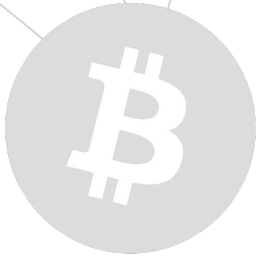
1) The key pair of C is $\widetilde{C}$ and the key pair of each P_i is $\widetilde{P}_i$.
2) The Ledger contains n unredeemed transactions $U_1^{\mathsf{C}}, \ldots, U_n^{\mathsf{C}}$, which can be redeemed with key $\widetilde{C}$, each having value d ฿.

The $\mathsf{CS.Commit}(\mathsf{C}, d, t, s)$ **phase**

3) The Committer C computes $h = H(s)$. He sends to the Ledger the transactions $Commit_1, \ldots, Commit_n$. This obviously means that he reveals h, as it is a part of each $Commit_i$.
4) If within time $\max_{\mathsf{Ledger}}$ some of the $Commit_i$ transactions does not appear on the Ledger, or if they look incorrect (e.g. they differ in the h value) then the parties abort.
5) The Committer C creates the bodies of the transactions $PayDeposit_1, \ldots, PayDeposit_n$, signs them and for all i sends the signed body $[PayDeposit_i]$ to P_i. If an appropriate transaction does not arrive to P_i, then he halts.

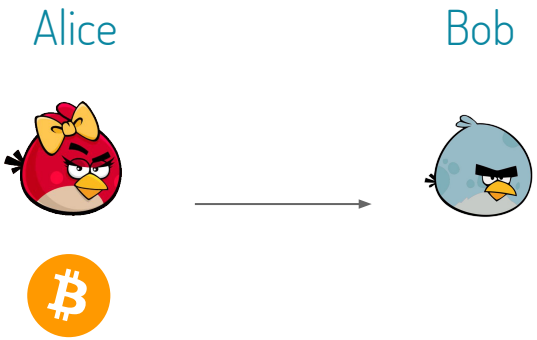
The $\mathsf{CS.Open}(\mathsf{C}, d, t, s)$ **phase**

6) The Committer C sends to the Ledger the transactions $Open_1, \ldots, Open_n$, what reveals the secret s.
7) If within time t the transaction $Open_i$ does not appear on the Ledger then $\widetilde{P}_i$ signs and sends the transaction $PayDeposit_i$ to the Ledger and earns d ฿.

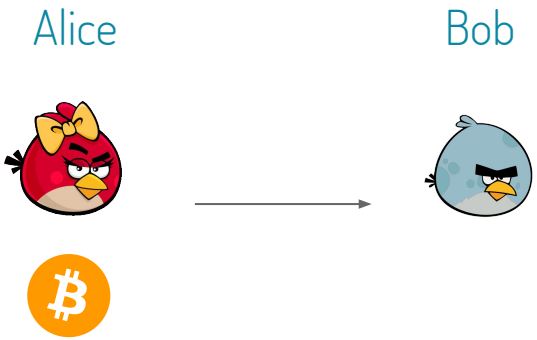


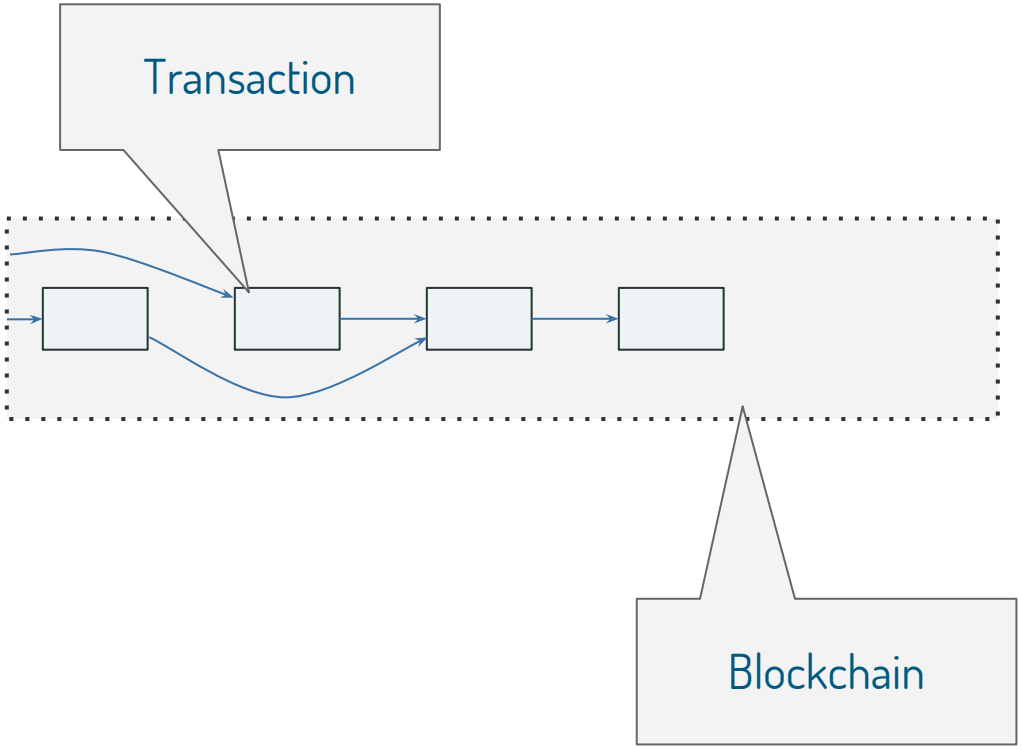
Bitcoin transactions

Bitcoin in a nutshell

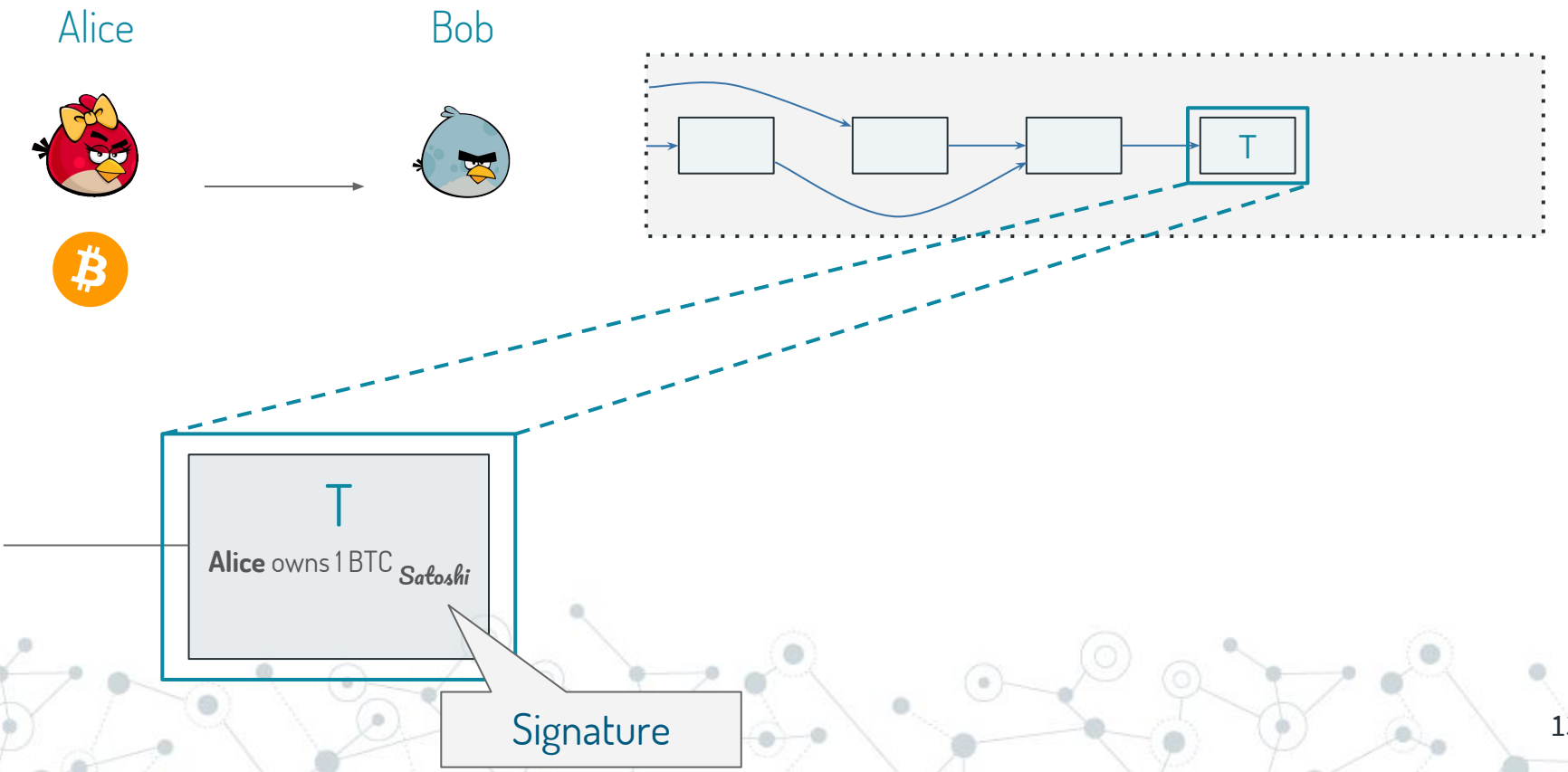


Bitcoin in a nutshell

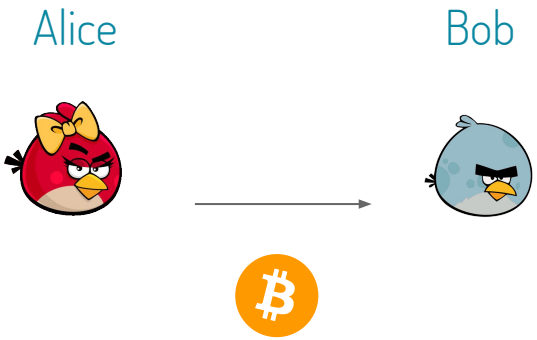


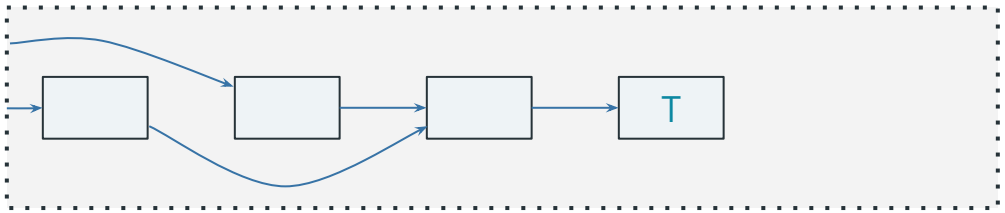


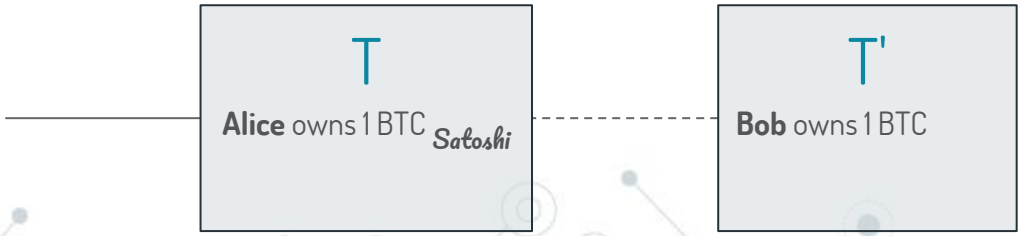
Bitcoin in a nutshell



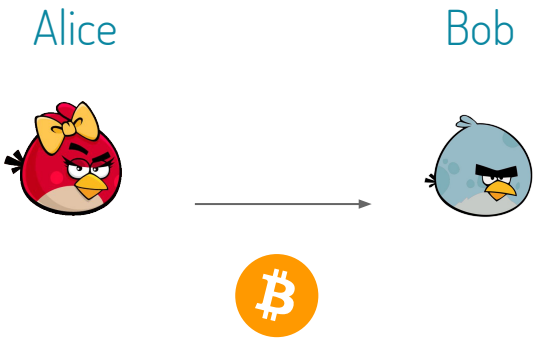
Bitcoin in a nutshell

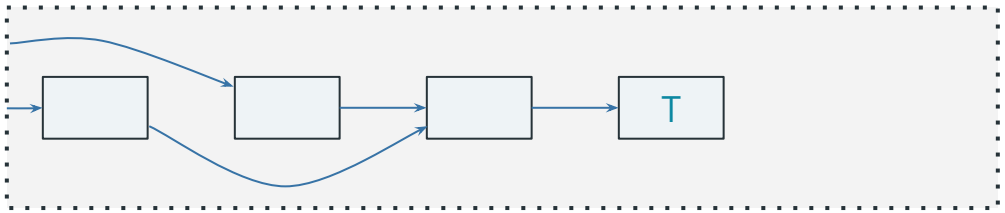


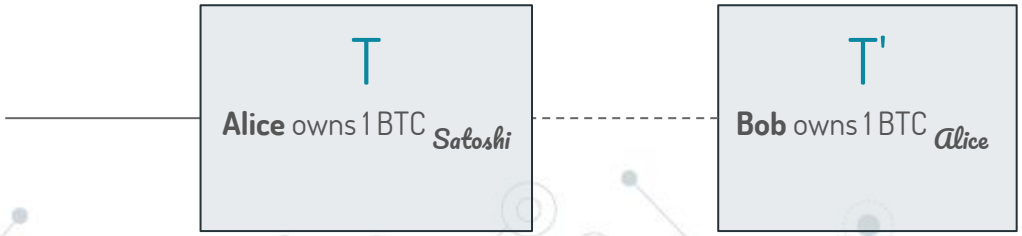




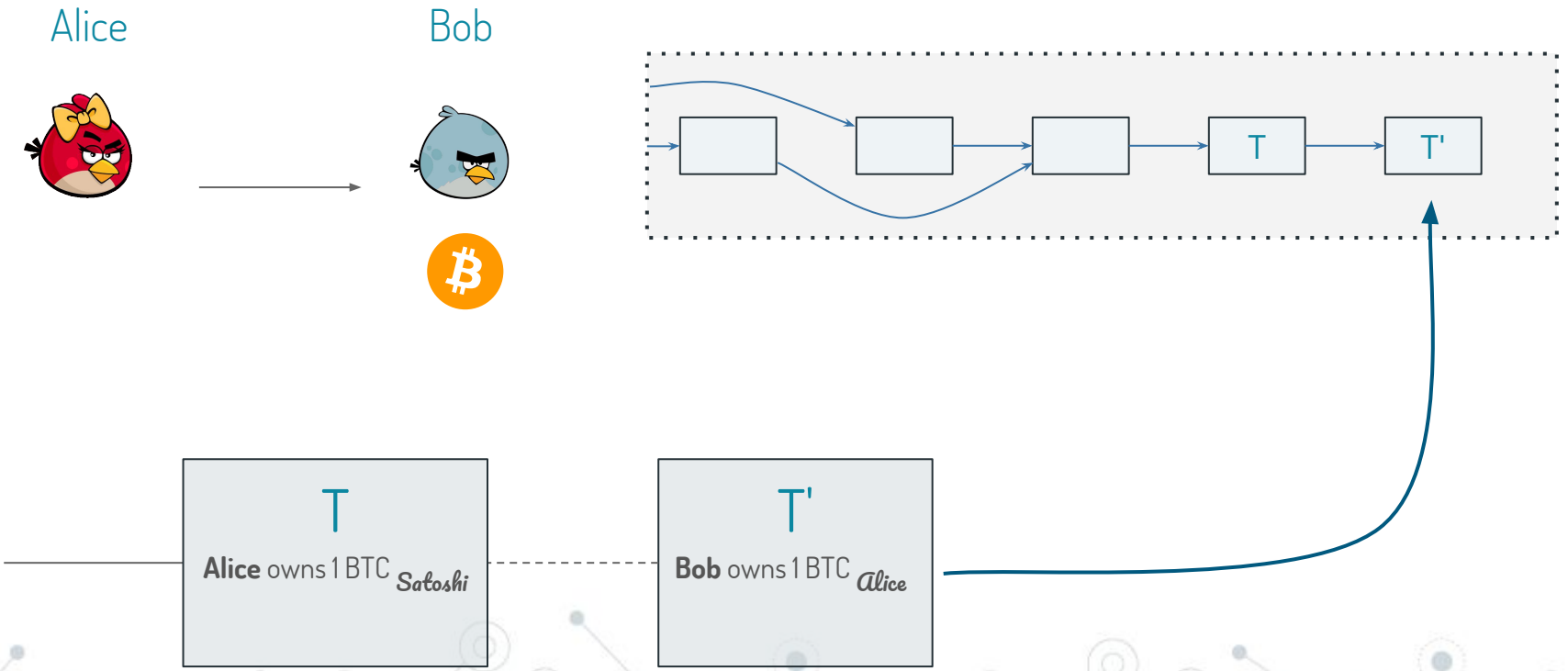
Bitcoin in a nutshell



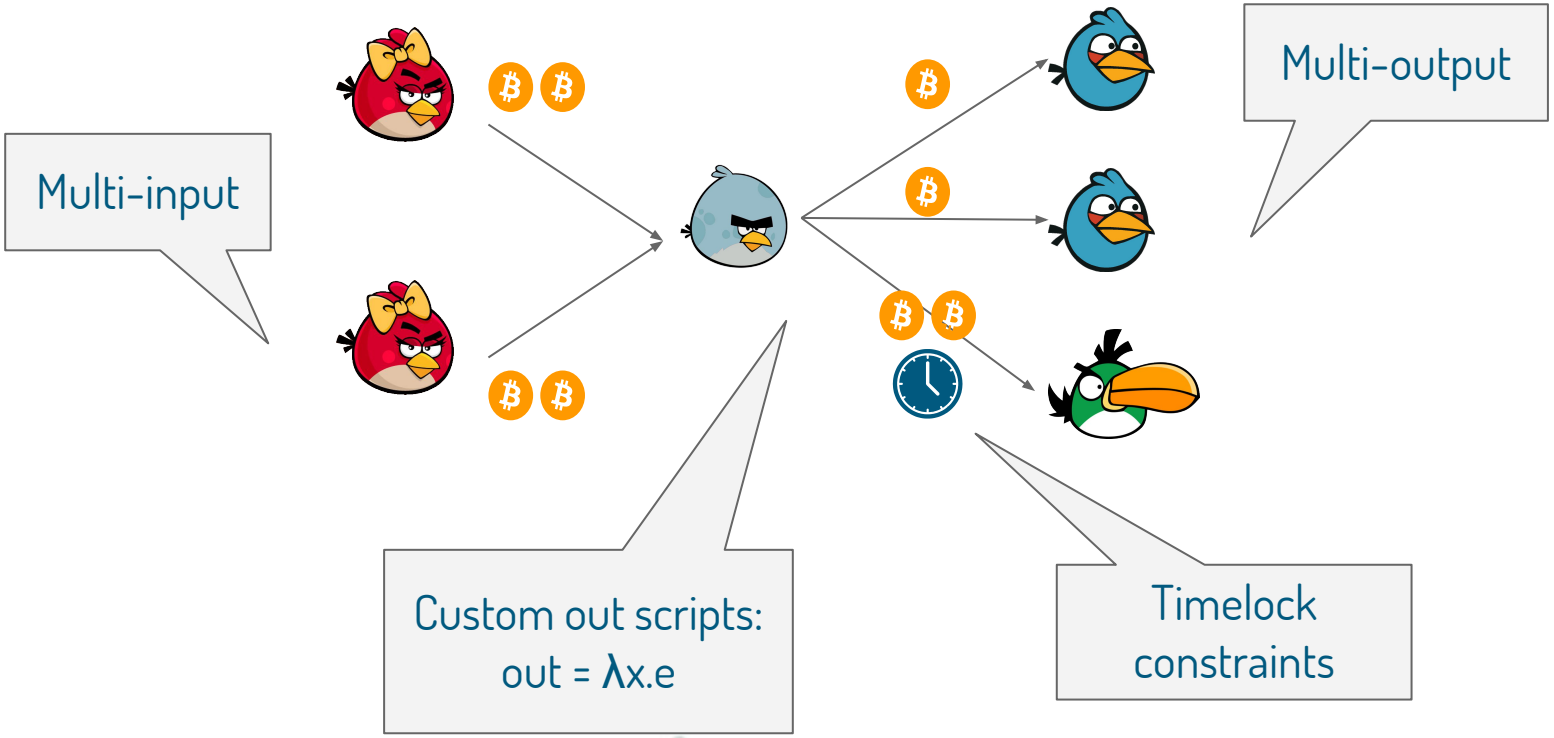




Bitcoin in a nutshell



More advanced features



Example: Bitcoin scripting language

- Stack-based language for out scripts

```
OP_SHA256 52..3e OP_EQUAL OP_SWAP 04..91
OP_CHECKSIG OP_BOOLOR OP_SWAP 04..40d
OP_CHECKSIG OP_BOOLAND
```

- Difficult to write and to read
- No formal semantics

Example: signature verification

Step 6: Copy TxNew to TxCopy (to be modified)

Step 7: Set all TxIn scripts in TxCopy to empty strings

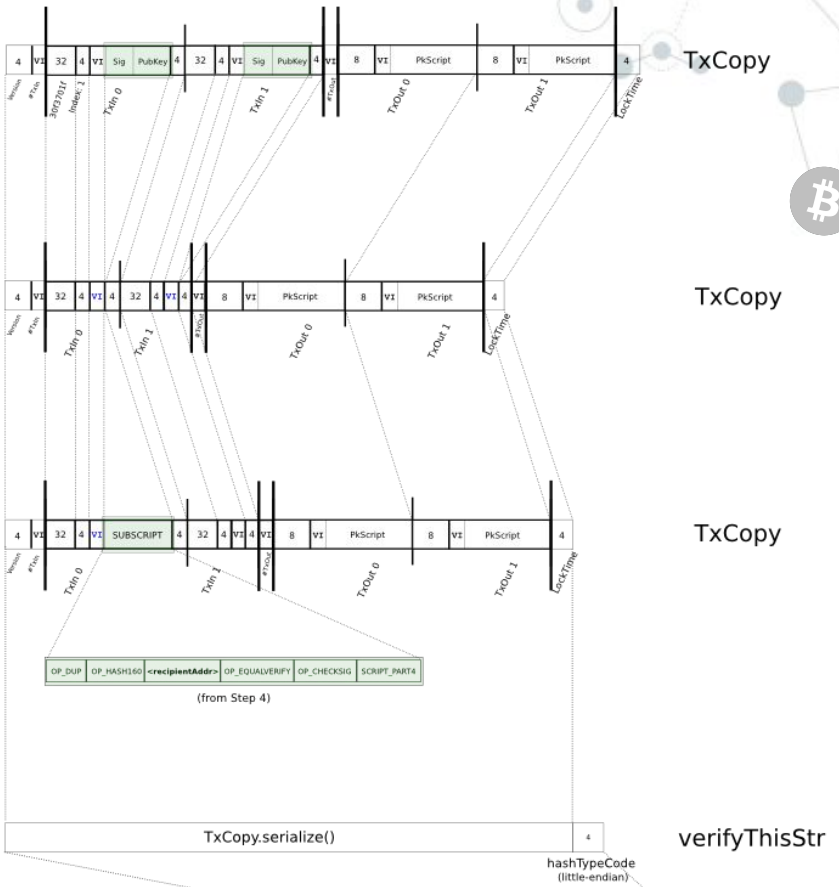
Make sure that the VAR_INT's representing script length are reevaluated to a single 0x00 byte for each TxIn

Step 8: Copy Subscript into the TxIn script you are checking:

Make sure VAR_INT preceding SUBSCRIPT is reevaluated to represent the size of SUBSCRIPT

Step 9: Serialize TxCopy, append 4-byte hashTypeCode:

Step 10: Verify signature against string in Step 9, (hashed string needs to be big-endian)

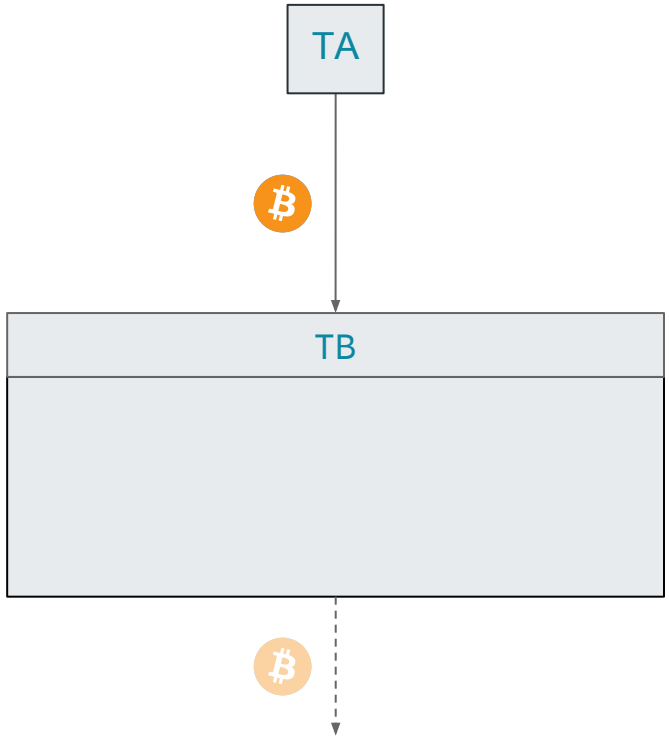


A formal model of Bitcoin transactions*

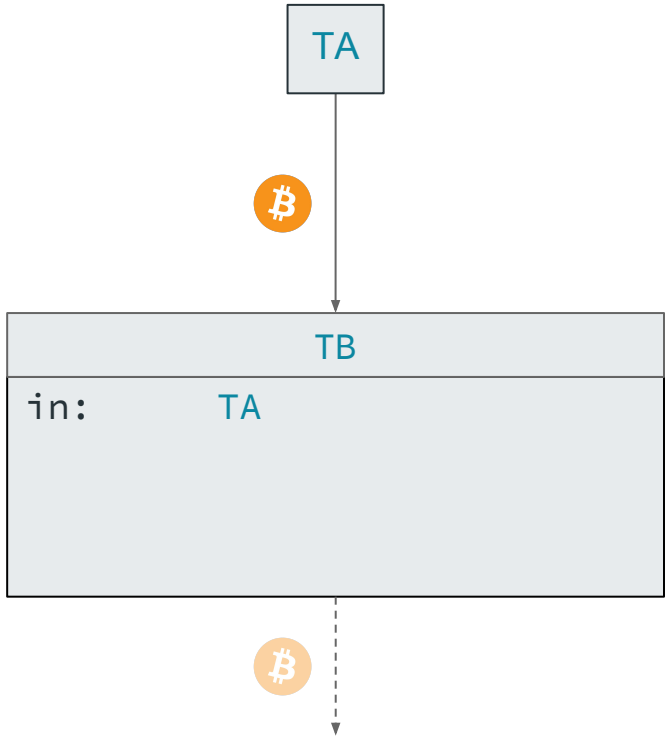
- Abstracts from low-level details
- Concrete enough to be used as an alternative documentation
- Formalises advanced features, e.g.:
  - Custom out scripts
  - Signature modifiers
  - SegWit
- Exploitable to build further abstractions...

* Atzei, Bartoletti, Lande, Zunino. Financial Cryptography 2018

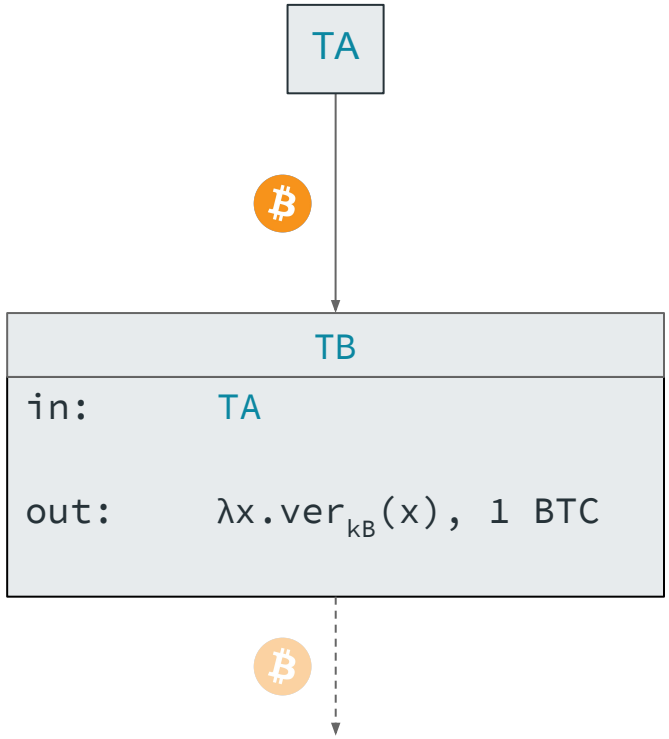
Bitcoin transactions



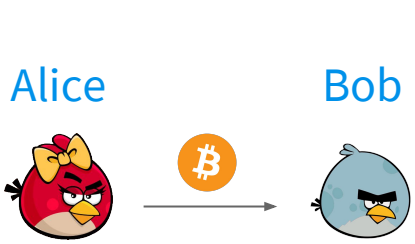
Bitcoin transactions

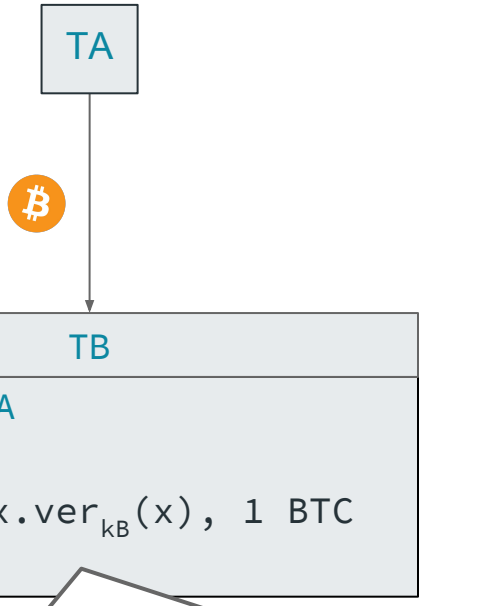


Bitcoin transactions



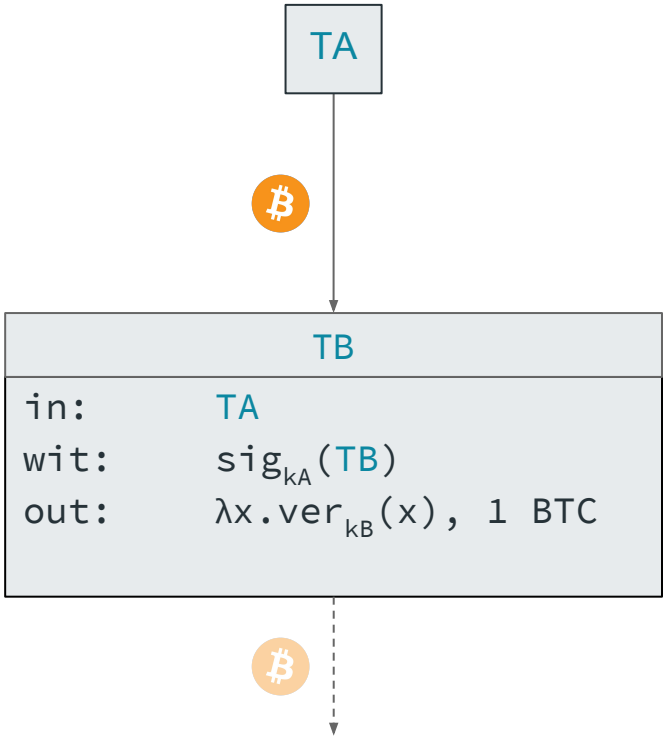
Bitcoin transactions






```
"out":[
{"value":"1.00",
"scriptPubKey":"OP_SHA256  3e..00
               OP_EQUALVERIFY b5..e2 OP_CHECKSIG"
}]
```

Bitcoin transactions



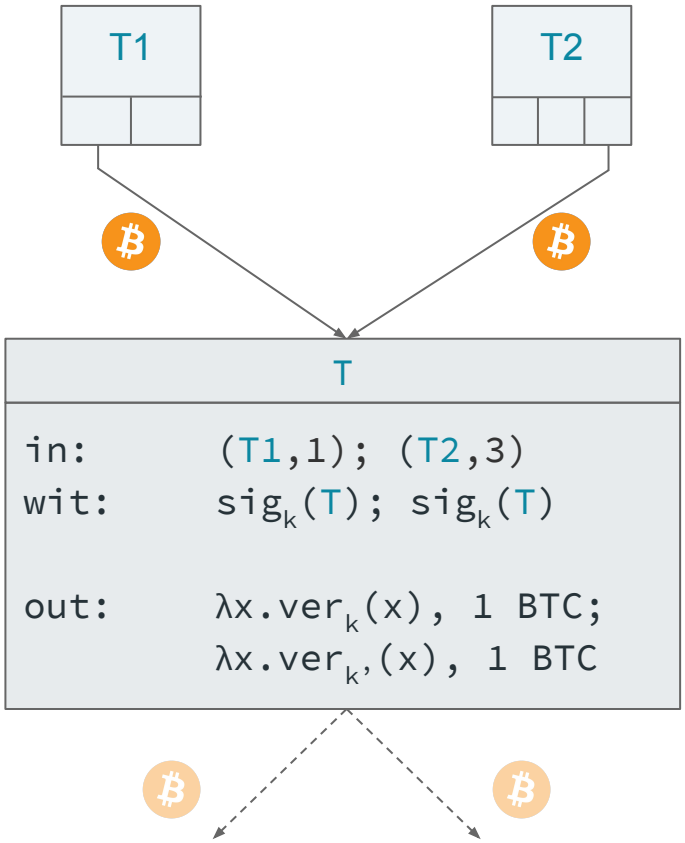
`TB.wit ⊨ TA.out`

The witness satisfies the out script

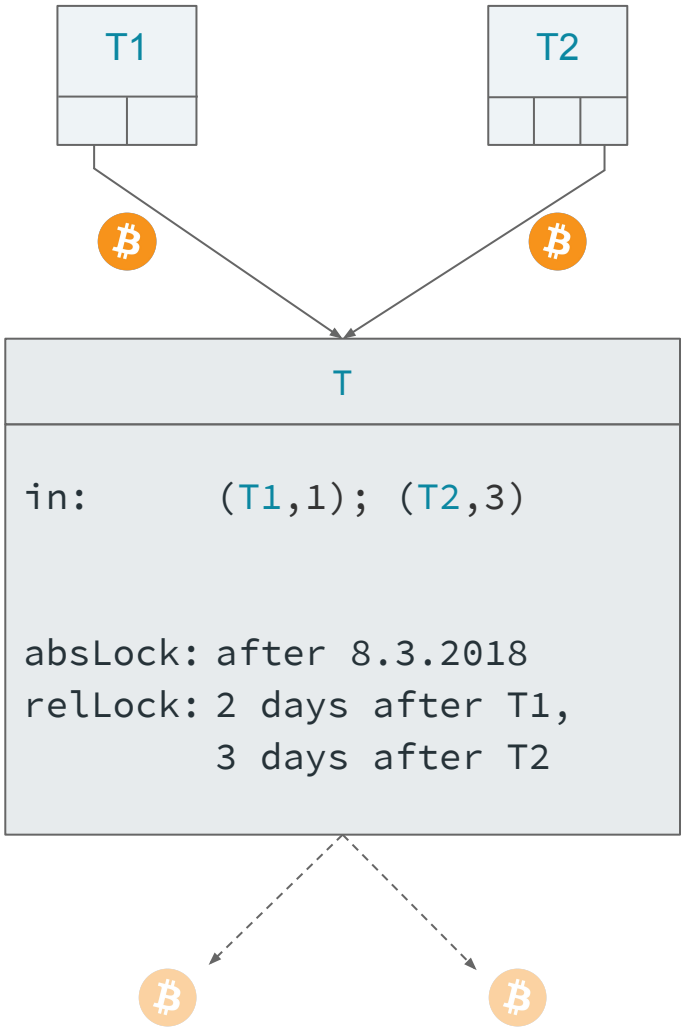
General form of transactions

$T.wit(1) \vDash T1.out(1)$

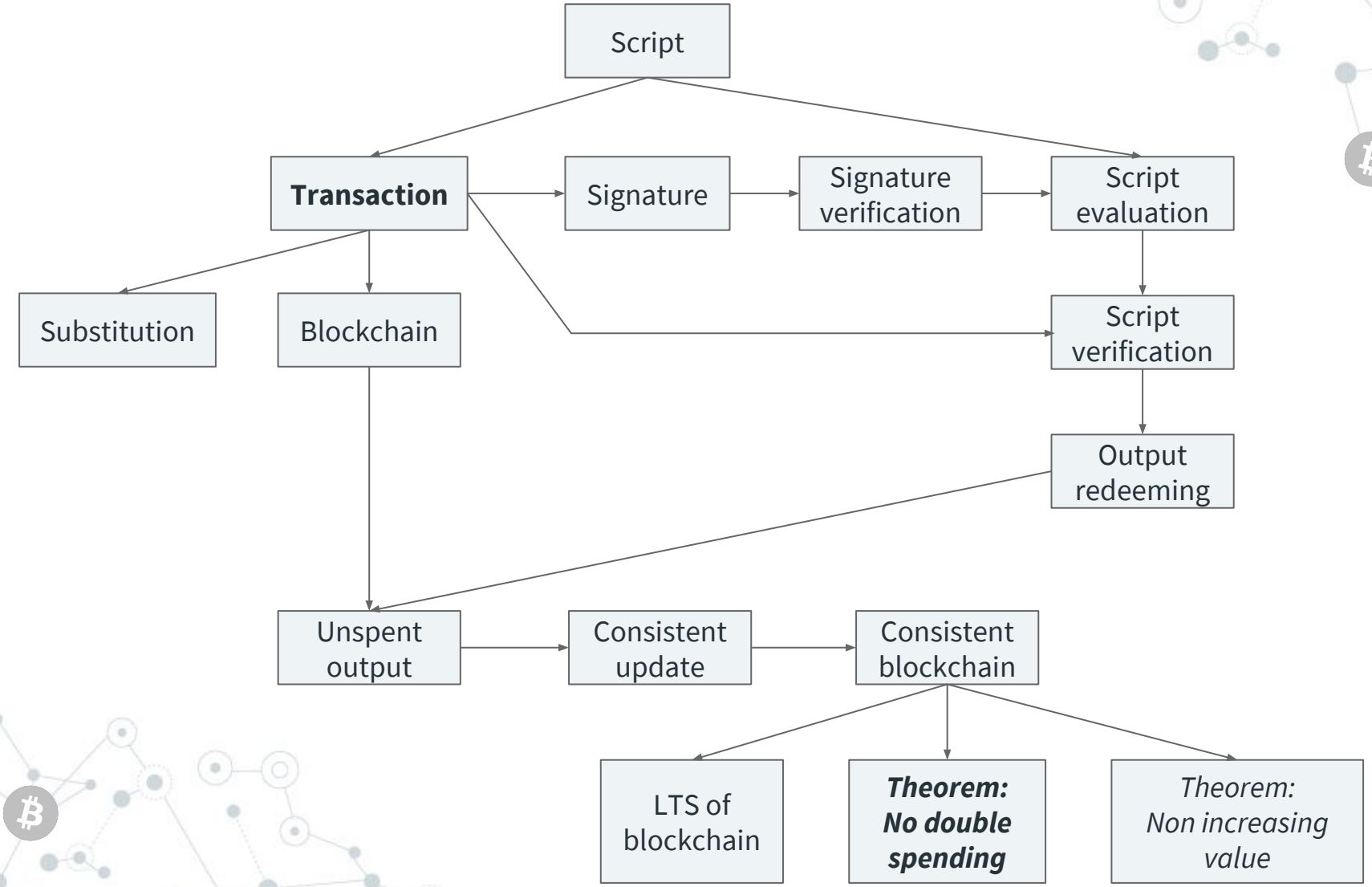
$T.wit(2) \vDash T2.out(3)$



Time constraints



A formal model of Bitcoin transactions

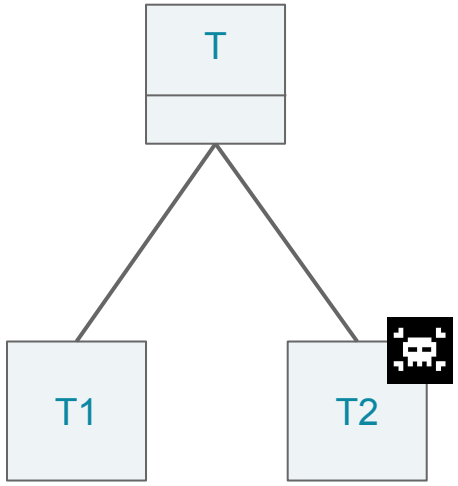


Theorem: No double spending

*If **B** is a consistent blockchain, then*

$$\forall T_1 \neq T_2 \in \boldsymbol{B} \ :$$

$$\text{ran}(T_1.\text{in}) \cap \text{ran}(T_2.\text{in}) = \emptyset$$

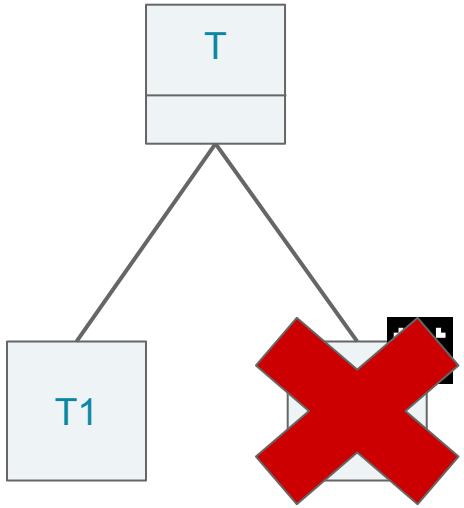


Theorem: No double spending

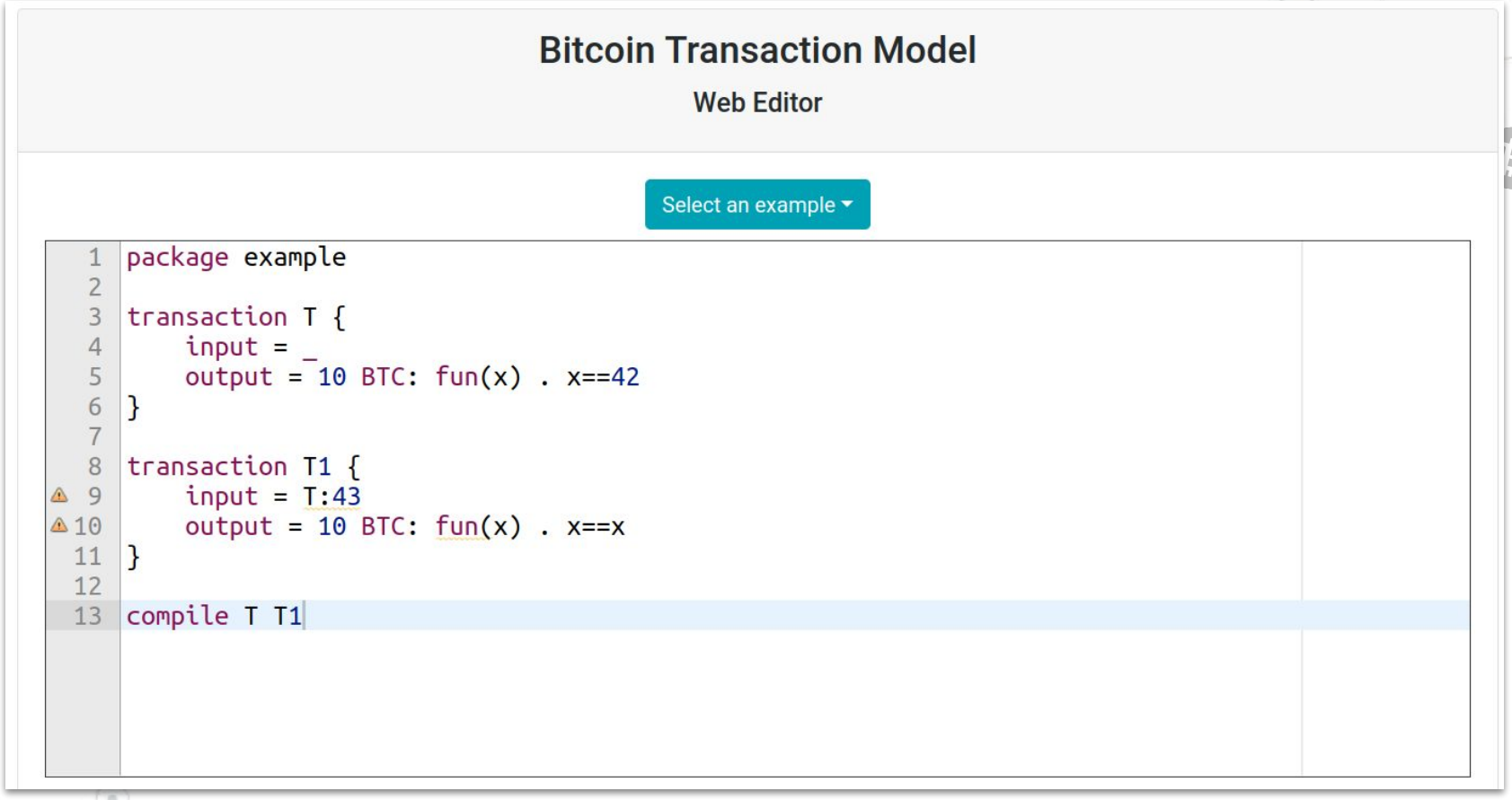
*If **B** is a consistent blockchain, then*

$$\forall T_1 \neq T_2 \in \boldsymbol{B} \ :$$

$$\text{ran}(T_1.\text{in}) \cap \text{ran}(T_2.\text{in}) = \emptyset$$



BALZaC: Bitcon Abstract Language, analyZer and Compiler



https://blockchain.unica.it/balzac
https://github.com/balzac-lang/balzac

BALZaC: Bitcon Abstract Language, analyZer and Compiler

```
1  package example
2
3  transaction T {
4      input = _
5      output = 10 BTC: fun(x) . x==42
6  }
7
8  transaction T1 {
9      input = T:43
```

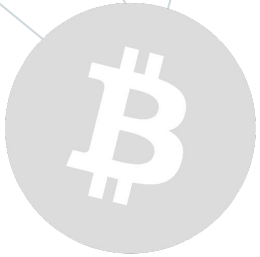
```
This input does not redeem the specified output script.

Details: P2SH script execution resulted in a non-true stack: []

INPUT:   PUSHDATA(1)[2b] PUSHDATA(3)[012a87]
OUTPUT:  HASH160 PUSHDATA(20)[afc0a51d093e9fc21dd64ef3add6130f4e402087] EQUAL
REDEEM SCRIPT:  PUSHDATA(1)[2a] EQUAL
REDEEM SCRIPT HASH:  afc0a51d093e9fc21dd64ef3add6130f4e402087
```

BALZaC: Bitcon Abstract Language, analyZer and Compiler

```
1  package example
2
3  transaction T {
4      input = _
5      output = 10 BTC: fun(x) . x==42
6  }
7
8  transaction T1 {
9      input = T:43
10     output = 10 BTC: fun(x) . x==x
       Script will always evaluate to true
13 compile T T1
```



Smart contracts

(as endpoint protocols)

Example: timed commitment

Bitcoin smart contracts are crypto protocols, which exploit advanced features of transactions

- Alice wants to commit a secret s but reveal it some time later
- Bob wants to be assured that he will either:
  - learn the secret within time t
  - or be economically compensated

[Andrychowicz et al. 2014]

Example: timed commitment

- A chooses a secret **s**
- She broadcasts **h** s.t. h=H(s)
- She can reveal the secret before time **t**

Commit

out: $\lambda x\, \sigma.$
(H(x)=**h** and $ver_{kA}(\sigma)$) or
afterAbs t: $ver_{kB}(\sigma)$, 1BTC

Example: timed commitment

- A chooses a secret **s**
- She broadcasts **h** s.t. h=H(s)
- She can reveal the secret before time **t**

Commit

```
out: λx σ.
    ( H(x)=h and ver_kA(σ) ) or
    afterAbs t: ver_kB(σ), 1BTC
```

Example: timed commitment

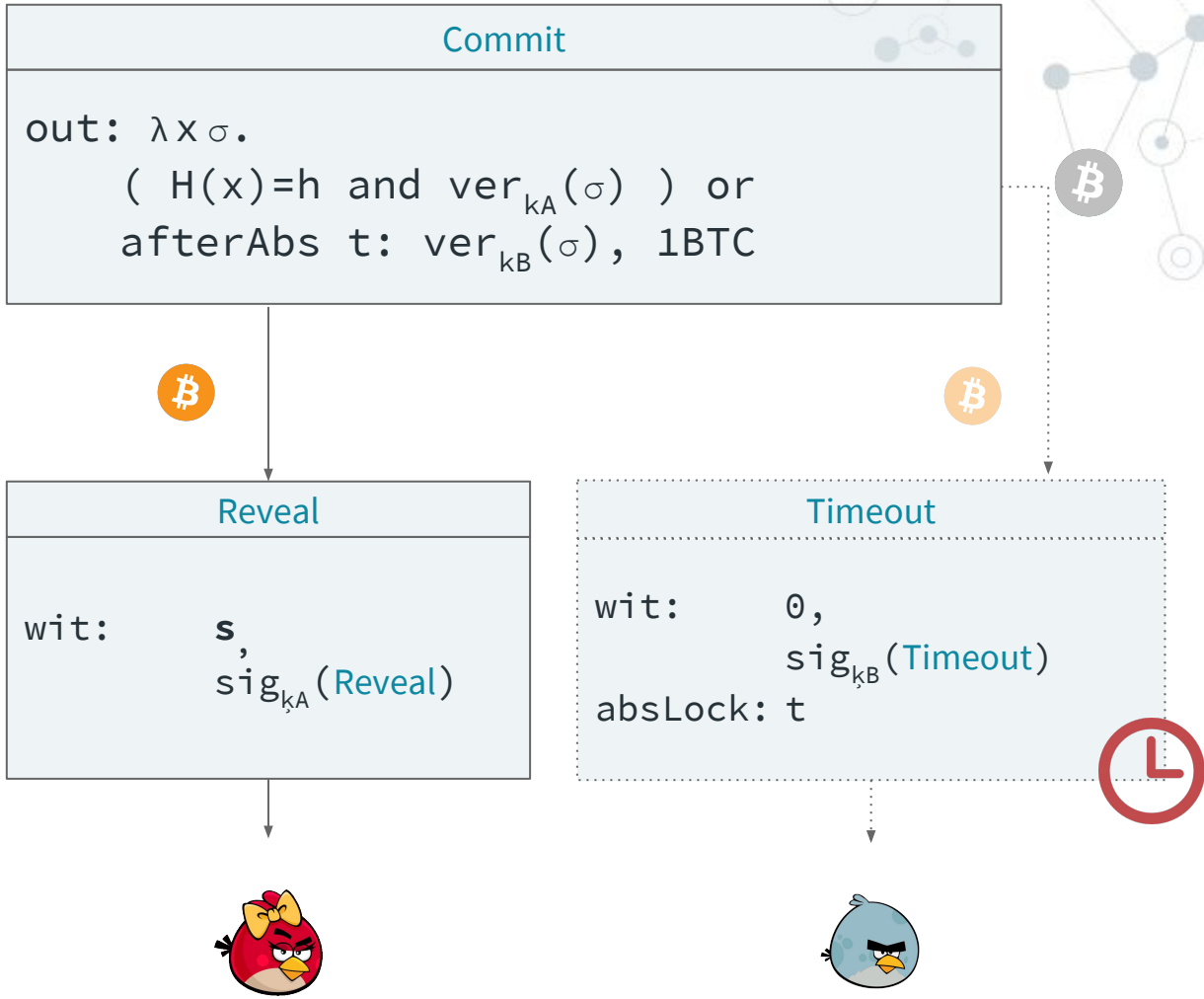
- A chooses a secret **s**
- She broadcasts **h** s.t. h=H(s)
- She can reveal the secret before time **t**

Commit

out: $\lambda x\, \sigma.$
(H(x)=h and $\mathrm{ver}_{kA}(\sigma)$) or
afterAbs t: $\mathrm{ver}_{kB}(\sigma)$, 1BTC

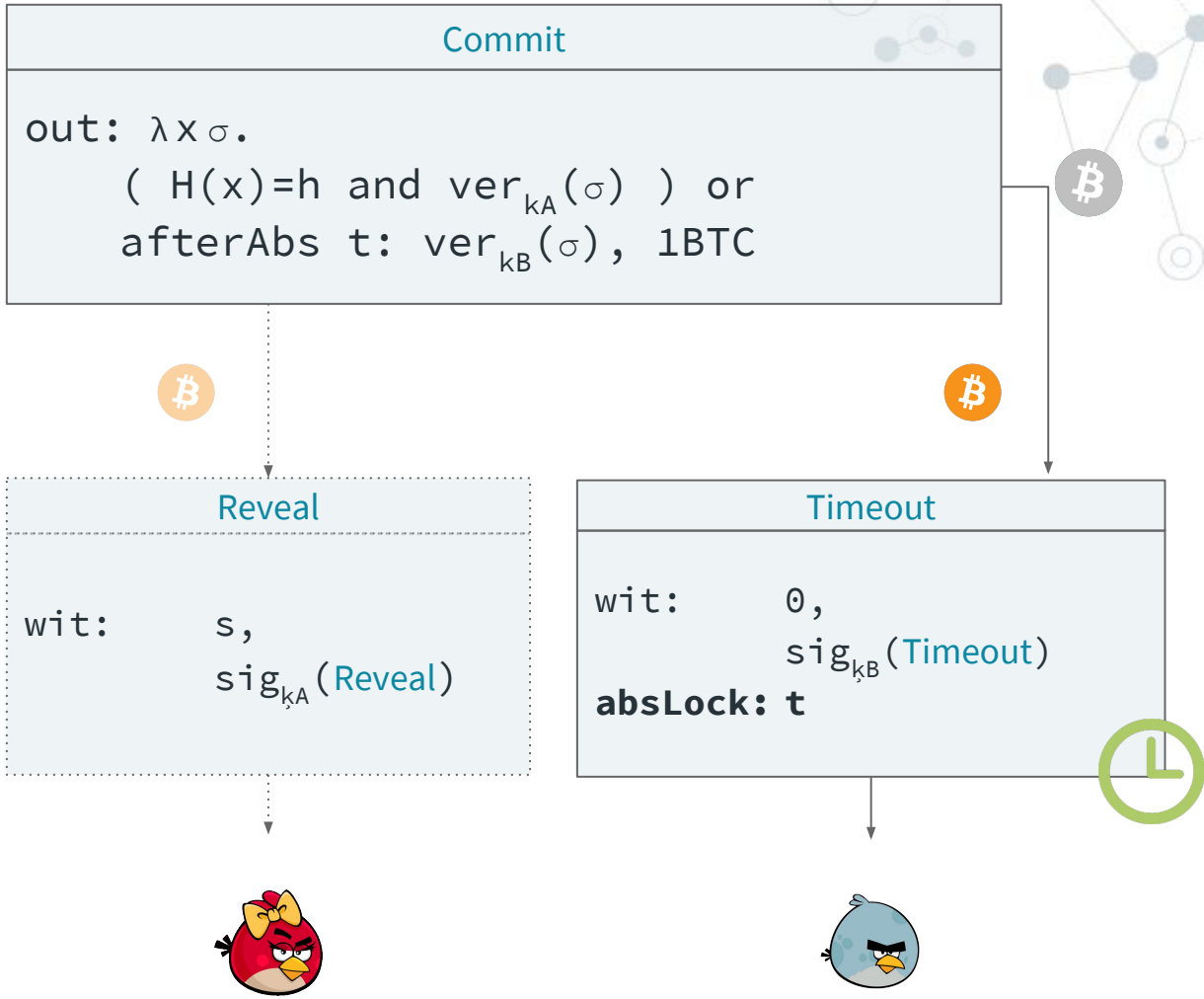
Example: timed commitment

- A chooses a secret **s**
- She broadcasts **h** s.t. h=H(s)
- She can reveal the secret before time **t**



Example: timed commitment

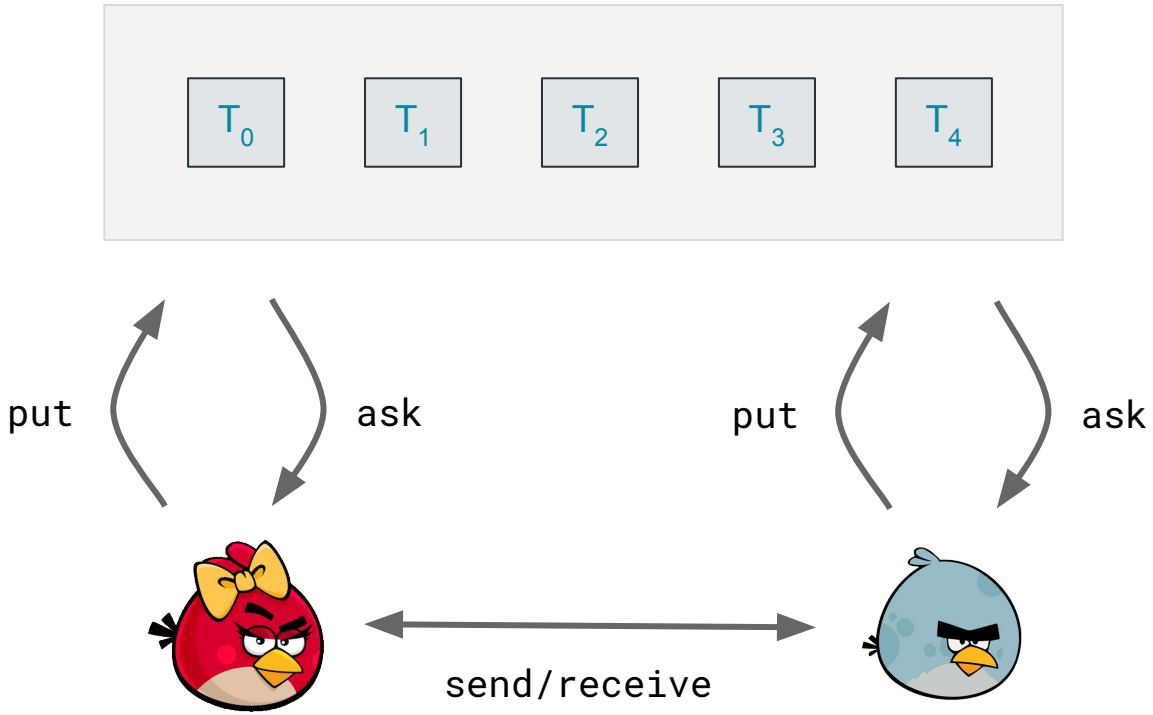
- After time **t**, B can redeem the deposit



Other Bitcoin contracts

- Oracles
- Crowdfunding
- Escrow and arbitration
- Micropayments channels (“Lighting network”)
- Fair multi-player lotteries
- Gambling games (Poker, ...)
- Contingent payments (via ZK proofs)
- …

Contracts as processes



SoK: unravelling Bitcoin smart contracts
(POST18)

Timed commitment (in prose)

Pre-condition:

1) The key pair of C is $\widetilde{C}$ and the key pair of each P_i is $\widetilde{P}_i$.
2) The Ledger contains n unredeemed transactions $U_1^{\mathsf{C}}, \ldots, U_n^{\mathsf{C}}$, which can be redeemed with key $\widetilde{C}$, each having value d ₿.

The $\mathsf{CS.Commit}(\mathsf{C}, d, t, s)$ **phase**

3) The Committer C computes $h = H(s)$. He sends to the Ledger the transactions $Commit_1, \ldots, Commit_n$. This obviously means that he reveals h, as it is a part of each $Commit_i$.
4) If within time $\max_{\mathsf{Ledger}}$ some of the $Commit_i$ transactions does not appear on the Ledger, or if they look incorrect (e.g. they differ in the h value) then the parties abort.
5) The Committer C creates the bodies of the transactions $PayDeposit_1, \ldots, PayDeposit_n$, signs them and for all i sends the signed body $[PayDeposit_i]$ to P_i. If an appropriate transaction does not arrive to P_i, then he halts.

The $\mathsf{CS.Open}(\mathsf{C}, d, t, s)$ **phase**

6) The Committer C sends to the Ledger the transactions $Open_1, \ldots, Open_n$, what reveals the secret s.
7) If within time t the transaction $Open_i$ does not appear on the Ledger then $\widetilde{P}_i$ signs and sends the transaction $PayDeposit_i$ to the Ledger and earns d ₿.

Timed commitment (as an endpoint protocol)

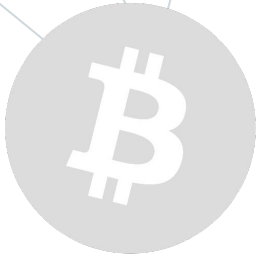
Compensation branch

P_A = put Commit . P'

P' = τ. put Reveal(s) + τ.P''

P_B = put Timeout

+ ask Reveal as x → Q(getSecret(x))



Contracts

(as choreographies)

BitML contracts

Endpoint protocols = **local** behaviour

- compensations lead to complex code
- still have to deal with Bitcoin transactions

BitML = **global** behaviour

- contracts as processes (no transactions)
- two-phases: stipulation and execution
- no compensations

Bartoletti & Zunino. BitML: a calculus for Bitcoin smart contracts, 2018

A basic example

A declares a 1฿ deposit:

```
{A:!1฿}
```

A authorizes to transfer the deposit to B, or
B authorizes to transfer the deposit to A

```
PayOrRefund =
   A:withdraw B + B:withdraw A
```

Mediating disputes (with oracles)

Resolve disputes via a mediator M (paid 0.1฿)

```
Escrow =  PayOrRefund  +
          A:Resolve       +
          B:Resolve


Resolve = split
     0.1฿ → withdraw M
   |  0.9฿ → M:withdraw A + M:withdraw B
```

Timed commitment

$$\{A:!1฿ \mid A:\mathrm{secret}\ a\}$$

$$TC = \quad \mathrm{reveal}\ a.\ \mathrm{withdraw}\ A$$

$$\quad + \quad \mathrm{after}\ t\ :\ \mathrm{withdraw}\ B$$

BitML: a 2-players lottery

```
{A:!3฿ | A:secret a | B:!3฿ | B:secret b}


split
 2฿ → reveal b if 0≤|b|≤1. withdraw B
    + after t : withdraw A
|2฿ → reveal a . withdraw A
    + after t : withdraw B
|2฿ → reveal a b if |a|=|b|. withdraw A
    +reveal a b if |a|≠|b|. withdraw B
```

Executing BitML on Bitcoin

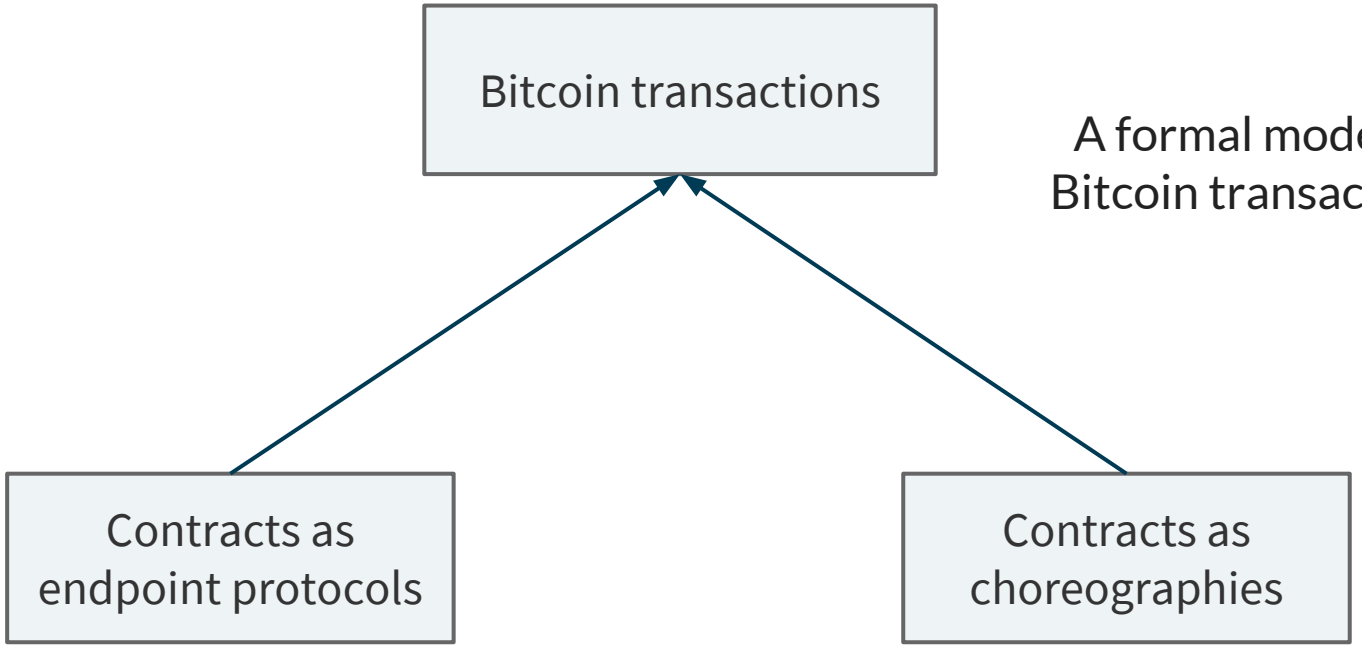
Compiler: BitML → BALZaC → Bitcoin txs

Theorem (Computational soundness):

For each computational run C, there exists a symbolic run S coherent with C

Therefore, computational attacks are always represented at the symbolic level.

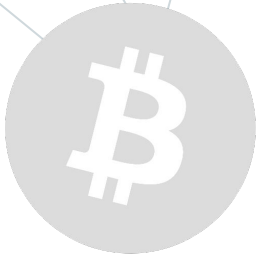
The whole talk in 1 slide



A formal model of Bitcoin transactions
(FC18)

SoK: unravelling Bitcoin smart contracts
(POST18)

BitML: a calculus for Bitcoin smart contracts
(submitted)



Thank you