

# Smart Contracts

## IOHK & Cardano

Lars Brünjes



January 9 2020

# About myself

Dr. Lars Brünjes, Director of Education at IOHK



- PhD in Pure Mathematics from Regensburg University (Germany).
- Postdoc at Cambridge University (UK).
- Ten years working in Software Development prior to joining IOHK.
- Haskell enthusiast for more than 15 years.
- Joined IOHK November 2016.
- Director of Education at IOHK: Haskell courses (Athens, Barbados, Addis Ababa, . . . ), responsible for internal and external trainings.
- Leading the “Incentives” team.

# IOHK & Cardano

## Motto

Providing financial services to the three billion people that don't have them.

## Motto

Providing financial services to the three billion people that don't have them.

- Founded 2015 by Charles Hoskinson and Jeremy Wood.

## Motto

Providing financial services to the three billion people that don't have them.

- Founded 2015 by Charles Hoskinson and Jeremy Wood.
- Company building Cardano.

## Motto

Providing financial services to the three billion people that don't have them.

- Founded 2015 by Charles Hoskinson and Jeremy Wood.
- Company building Cardano.
- Distributed around the globe.

## Motto

Providing financial services to the three billion people that don't have them.

- Founded 2015 by Charles Hoskinson and Jeremy Wood.
- Company building Cardano.
- Distributed around the globe.
- Invested in functional programming (Haskell, Scala, . . . ).

## Motto

Providing financial services to the three billion people that don't have them.

- Founded 2015 by Charles Hoskinson and Jeremy Wood.
- Company building Cardano.
- Distributed around the globe.
- Invested in functional programming (Haskell, Scala, . . . ).
- Research focused (peer-reviewed research, research centers, . . . ).

- Proof of Stake blockchain.

- Proof of Stake blockchain.
- Cryptocurrency Ada.

# Cardano

- Proof of Stake blockchain.
- Cryptocurrency Ada.
- Roadmap: <https://cardanoroadmap.com/>.

# Cardano

- Proof of Stake blockchain.
- Cryptocurrency Ada.
- Roadmap: <https://cardanoroadmap.com/>.
- **Smart Contracts**: IELT VM, Plutus, Marlowe.

# Proof of Work versus Proof of Stake

PoW

Leader selection based on Hashing  
Power: “One CPU, one vote!”.

PoS

Leader selection based on Stake: “Follow the Satoshi!”

# Proof of Work versus Proof of Stake

PoW

Leader selection based on Hashing Power: “One CPU, one vote!”.

Huge energy consumption to guarantee security.

PoS

Leader selection based on Stake: “Follow the Satoshi!”

Consensus is relatively cheap.

# Proof of Work versus Proof of Stake

PoW

Leader selection based on Hashing Power: “One CPU, one vote!”.

Huge energy consumption to guarantee security.

Well established and provably secure.

PoS

Leader selection based on Stake: “Follow the Satoshi!”

Consensus is relatively cheap.

Provably secure, but hotly debated.

# Ouroboros

# Ouroboros

- First provably secure PoS protocol.

# Ouroboros

- First provably secure PoS protocol.
- Elect leader for each time-slot based on stake.

# Ouroboros

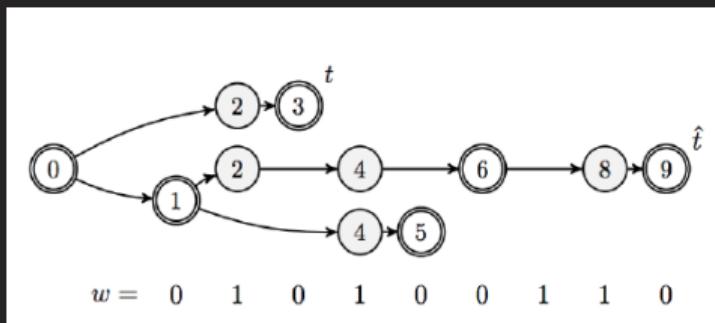
- First provably secure PoS protocol.
- Elect leader for each time-slot based on stake.
- Stakeholders agree on randomness for next epoch.

# Ouroboros

- First provably secure PoS protocol.
- Elect leader for each time-slot based on stake.
- Stakeholders agree on randomness for next epoch.
- Running in production in Cardano since October 2017.

# Ouroboros

- First provably secure PoS protocol.
- Elect leader for each time-slot based on stake.
- Stakeholders agree on randomness for next epoch.
- Running in production in Cardano since October 2017.
- Provably secure against adversary with less than 50% stake.



Adversary	BTC	OB Covert	OB General
0.10	50	3	5
0.15	80	5	8
0.20	110	7	12
0.25	150	11	18
0.30	240	18	31
0.35	410	34	60
0.40	890	78	148
0.45	3400	317	663

# Ouroboros Praos

- Extension of Ouroboros to semi-synchronous setting.

# Ouroboros Praos

- Extension of Ouroboros to semi-synchronous setting.
- Deals gracefully with message delays.

# Ouroboros Praos

- Extension of Ouroboros to semi-synchronous setting.
- Deals gracefully with message delays.
- Currently being implemented for future versions of Cardano.

# Ouroboros Genesis

- No checkpointing: New Players can safely join the protocol without any trusted advice.

# Ouroboros Genesis

- No checkpointing: New Players can safely join the protocol without any trusted advice.
- Security Proof in the UC-framework, making it easier to compare with Bitcoin (and other PoW systems).

# Ouroboros BFT

- Ouroboros BFT (Byzantine Fault Tolerance) is a minimal consensus-protocol which is impressively simple.

# Ouroboros BFT

- Ouroboros BFT (Byzantine Fault Tolerance) is a minimal consensus-protocol which is impressively simple.
- It only works for a fixed number  $n$  of nodes.

# Ouroboros BFT

- Ouroboros BFT (Byzantine Fault Tolerance) is a minimal consensus-protocol which is impressively simple.
- It only works for a fixed number  $n$  of nodes.
- It is secure for an honest majority of  $\frac{2}{3}n$  nodes.

# Ouroboros BFT

- Ouroboros BFT (Byzantine Fault Tolerance) is a minimal consensus-protocol which is impressively simple.
- It only works for a fixed number  $n$  of nodes.
- It is secure for an honest majority of  $\frac{2}{3}n$  nodes.
- If dishonest nodes (so-called Byzantine nodes) are not allowed to commit publicly visible protocol violations, only  $\frac{n}{2}$  honest nodes are needed. (This is the so-called Covert Byzantine Setting.)

# Covert Byzantine Setting

- Normal Byzantine nodes can do what they like. They can ignore messages and send misleading messages whenever and however often they want.

# Covert Byzantine Setting

- Normal Byzantine nodes can do what they like. They can ignore messages and send misleading messages whenever and however often they want.
- In the *covert Byzantine setting*, such nodes are still allowed to violate the protocol (for example by being quiet or by ignoring messages from other nodes, pretending to face network problems), but they are not allowed to violate the protocol in a publicly visible way.

# Covert Byzantine Setting

- Normal Byzantine nodes can do what they like. They can ignore messages and send misleading messages whenever and however often they want.
- In the *covert Byzantine setting*, such nodes are still allowed to violate the protocol (for example by being quiet or by ignoring messages from other nodes, pretending to face network problems), but they are not allowed to violate the protocol in a publicly visible way.
- In particular they cannot create more than one block with the same time stamp and distribute those blocks in the network.

# Covert Byzantine Setting

- Normal Byzantine nodes can do what they like. They can ignore messages and send misleading messages whenever and however often they want.
- In the **covert Byzantine setting**, such nodes are still allowed to violate the protocol (for example by being quiet or by ignoring messages from other nodes, pretending to face network problems), but they are not allowed to violate the protocol in a publicly visible way.
- In particular they cannot create more than one block with the same time stamp and distribute those blocks in the network.
- This is obviously a restriction, which hopefully makes it plausible why in this setting, a majority of 50% honest nodes suffices.

# Covert Byzantine Setting

- Normal Byzantine nodes can do what they like. They can ignore messages and send misleading messages whenever and however often they want.
  - In the **covert Byzantine setting**, such nodes are still allowed to violate the protocol (for example by being quiet or by ignoring messages from other nodes, pretending to face network problems), but they are not allowed to violate the protocol in a publicly visible way.
  - In particular they cannot create more than one block with the same time stamp and distribute those blocks in the network.
  - This is obviously a restriction, which hopefully makes it plausible why in this setting, a majority of 50% honest nodes suffices.
  - In practice this setting can be enforced by requiring an upfront deposit of all nodes, which will be forfeit if two blocks with the same time stamp signed by them are discovered.

# The Ouroboros BFT Protocol

- The  $n$  nodes are numbered from 0 to  $(n - 1)$ . Their public keys are fixed and publicly known.

# The Ouroboros BFT Protocol

- The  $n$  nodes are numbered from 0 to  $(n - 1)$ . Their public keys are fixed and publicly known.
- Time is divided into **slots** of a fixed length, slots are numbered starting with 1.

# The Ouroboros BFT Protocol

- The  $n$  nodes are numbered from 0 to  $(n - 1)$ . Their public keys are fixed and publicly known.
- Time is divided into **slots** of a fixed length, slots are numbered starting with 1.
- Newly created blocks contain the current slot-number as a time stamp.

# The Ouroboros BFT Protocol

- The  $n$  nodes are numbered from 0 to  $(n - 1)$ . Their public keys are fixed and publicly known.
- Time is divided into **slots** of a fixed length, slots are numbered starting with 1.
- Newly created blocks contain the current slot-number as a time stamp.
- In slot  $i$ , only node  $k$  with  $k \equiv i \pmod{n}$  has the right to create a block.

# The Ouroboros BFT Protocol

- The  $n$  nodes are numbered from 0 to  $(n - 1)$ . Their public keys are fixed and publicly known.
- Time is divided into **slots** of a fixed length, slots are numbered starting with 1.
- Newly created blocks contain the current slot-number as a time stamp.
- In slot  $i$ , only node  $k$  with  $k \equiv i \pmod{n}$  has the right to create a block.
- A block is valid if
  - Its time stamp is not from the future and
  - it contains the signature of the node associated with the slot.

## Illustration: Ouroboros BFT with Seven Nodes

Nodes	Slots
0	7, 14, 21, 28,...
1	1, 8, 15, 22,...
2	2, 9, 16, 23,...
3	3, 10, 17, 24,...
4	4, 11, 18, 25,...
5	5, 12, 19, 26,...
6	6, 13, 20, 27,...

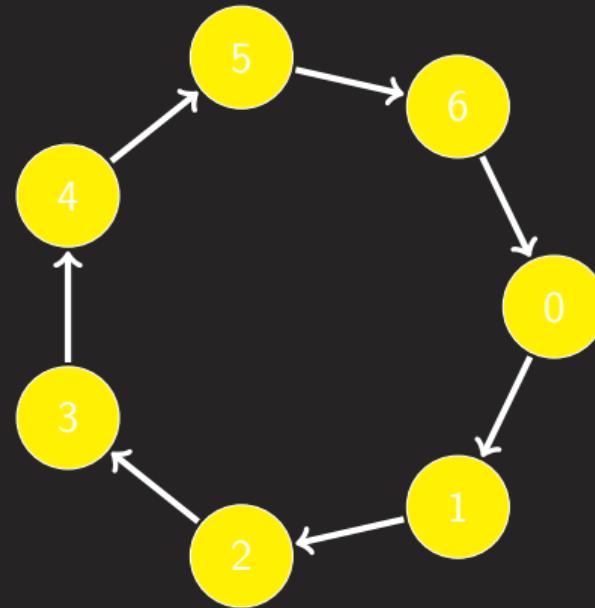


Figure: Ouroboros BFT with seven nodes

# Formal Methods

# From Mathematical Paper...

## Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain

Bernardo David\*, Peter Gaži\*\*, Aggelos Kiayias\*\*\*, and Alexander Russell†

October 6, 2017

**Abstract.** We present “Ouroboros Praos”, a proof-of-stake blockchain that, for the first time, provides security against *fully-adaptive corruptions*. Specifically, the adversary can corrupt any part of the population of stakeholders at any moment as long as there is an honest majority of stake; furthermore, the protocol tolerates message delivery delay unknown to protocol participants. To achieve these guarantees we formalize and realize in the suitable form of forward secure digital signatures and a new that maintains unpredictability under malicious key generation a general combinatorial framework for the analysis of semi-synchronous protocols. We prove our protocol secure in the random oracle model.

The protocol  $\pi_{\text{SPoS}}$  is run by stakeholders  $U_1, \dots, U_n$  interacting among themselves and with ideal functionalities  $\mathcal{F}_{\text{BTR}}, \mathcal{F}_{\text{VRF}}, \mathcal{F}_{\text{KES}}, \mathcal{F}_{\text{DSIG}}, \mathcal{H}$  over a sequence of slots  $S = \{sl_1, \dots, sl_R\}$ . Define  $T_i \triangleq 2^{\ell_{\text{VRF}}} \phi_f(\alpha_i)$  as the threshold for a stakeholder  $U_i$ , where  $\alpha_i$  is the relative stake of  $U_i$ ,  $\ell_{\text{VRF}}$  denotes the output length of  $\mathcal{F}_{\text{VRF}}$ ,  $f$  is the active slots coefficient and  $\phi_f$  is the mapping from Definition 1. Then  $\pi_{\text{SPoS}}$  proceeds as follows:

- 1. Initialization.** The stakeholder  $U_i$  sends  $(\text{KeyGen}, sid, U_i)$  to  $\mathcal{F}_{\text{VRF}}, \mathcal{F}_{\text{KES}}$  and  $\mathcal{F}_{\text{DSIG}}$ ; receiving  $(\text{VerificationKey}, sid, v_i^{\text{ver}})$ ,  $(\text{VerificationKey}, sid, v_i^{\text{kes}})$  and  $(\text{VerificationKey}, sid, v_i^{\text{dsig}})$ , respectively. Then, in case it is the first round, it sends  $(\text{ver.keys}, sid, U_i, v_i^{\text{ver}}, v_i^{\text{kes}}, v_i^{\text{dsig}})$  to  $\mathcal{F}_{\text{BTR}}$  (to claim stake from the genesis block). In any case, it terminates the round by returning  $(U_i, v_i^{\text{ver}}, v_i^{\text{kes}}, v_i^{\text{dsig}})$  to  $\mathcal{Z}$ . In the next round,  $U_i$  sends  $(\text{genblock.req}, sid, U_i)$  to  $\mathcal{F}_{\text{BTR}}$ , receiving  $(\text{genblock}, sid, S_0, \eta)$  as the answer.  $U_i$  sets the local blockchain  $C = B_0 = (S_0, \eta)$  and its initial internal state  $st = H(B_0)$ .
- 2. Chain Extension.** After initialization, for every slot  $sl_j \in S$ , every online stakeholder  $U_i$  performs the following steps:
  - $U_i$  receives from the environment the transaction data  $d \in \{0, 1\}^*$  to be inserted into the blockchain.
  - $U_i$  collects all valid chains received via diffusion into a set  $\mathbb{C}$ , pruning blocks belonging to future slots and verifying that for every chain  $\mathcal{C}' \in \mathbb{C}$  and every block  $B' = (st', d', sl', B_{st'}, \sigma_{st'}) \in \mathcal{C}'$  it holds that the stakeholder who created it is in the slot leader set of slot  $sl'$  (by parsing  $B_{st'}$  as  $(U_s, y', \pi')$  for some  $s$ , verifying that  $\mathcal{F}_{\text{VRF}}$  responds to  $(\text{Verify}, sid, \eta \parallel d', y', \pi', v_s^{\text{ver}})$  by  $(\text{Verified}, sid, \eta \parallel d', y', \pi', 1)$ , and that  $y' < T_s$ ), and that  $\mathcal{F}_{\text{KES}}$  responds to  $(\text{Verify}, sid, (st', d', sl', B_{st'}), st', \sigma_{st'}, v_s^{\text{kes}})$  by  $(\text{Verified}, sid, (st', d', sl', B_{st'}), st', 1)$ .  $U_i$  computes  $\mathcal{C}' = \text{maxvalid}(\mathbb{C}, \mathcal{C})$ , sets  $\mathcal{C}'$  as the new local chain and sets state  $st = H(\text{head}(\mathcal{C}'))$ .
  - $U_i$  sends  $(\text{EvalProve}, sid, \eta \parallel sl_j)$  to  $\mathcal{F}_{\text{VRF}}$ , receiving  $(\text{Evaluated}, sid, y, \pi)$ .  $U_i$  checks whether it is in the slot leader set of slot  $sl_j$  by checking that  $y < T_i$ . If yes, it generates a new block  $B = (st, d, sl_j, B_{st}, \sigma)$  where  $st$  is its current state,  $d \in \{0, 1\}^*$  is the transaction data,  $B_{st} = (U_i, y, \pi)$  and  $\sigma$  is a signature obtained by sending  $(\text{USign}, sid, U_i, (st, d, sl_j, B_{st}), sl_j)$  to  $\mathcal{F}_{\text{KES}}$  and receiving  $(\text{Signature}, sid, (st, d, sl_j, B_{st}), sl_j, \sigma)$ .  $U_i$  computes  $\mathcal{C}' = \mathcal{C} \mid B$ , sets  $\mathcal{C}'$  as the new local chain and sets state  $st = H(\text{head}(\mathcal{C}'))$ . Finally, if  $U_i$  has generated a block in this step, it diffuses  $\mathcal{C}'$ .
  - 3. Signing Transactions.** Upon receiving  $(\text{sign.tx}, sid', tx)$  from the environment,  $U_i$  sends  $(\text{Sign}, sid, U_i, tx)$  to  $\mathcal{F}_{\text{DSIG}}$ , receiving  $(\text{Signature}, sid, tx, \sigma)$ . Then,  $U_i$  sends  $(\text{signed.tx}, sid', tx, \sigma)$  back to the environment.

Fig. 4: Protocol  $\pi_{\text{SPoS}}$ .

# From Mathematical Paper...

## Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain

Bernardo David\*, Peter Gaži\*\*, Aggelos Kiayias\*\*\*, and Alexander Russell†

October 6, 2017

Written in English.

**Abstract.** We present “Ouroboros Praos”, a proof-of-stake blockchain that, for the first time, provides security against *fully-adaptive corruptions*. Specifically, the adversary can corrupt any part of the population of stakeholders at any moment as long as there is an honest majority of stakeholders; furthermore, the protocol tolerates message delivery delay unknown to protocol participants. To achieve these guarantees we formalize and realize in the suitable form of forward secure digital signatures and a new that maintains unpredictability under malicious key generation a general combinatorial framework for the analysis of semi-independent interest. We prove our protocol secure in the random oracle model.

The protocol  $\pi_{\text{SPoS}}$  is run by stakeholders  $U_1, \dots, U_n$  interacting among themselves and with ideal functionalities  $\mathcal{F}_{\text{BMR}}, \mathcal{F}_{\text{VRF}}, \mathcal{F}_{\text{KES}}, \mathcal{F}_{\text{DSIG}}, \mathcal{H}$  over a sequence of slots  $S = \{sl_1, \dots, sl_R\}$ . Define  $T_i \triangleq 2^{\ell_{\text{VRF}}} \phi_f(\alpha_i)$  as the threshold for a stakeholder  $U_i$ , where  $\alpha_i$  is the relative stake of  $U_i$ ,  $\ell_{\text{VRF}}$  denotes the output length of  $\mathcal{F}_{\text{VRF}}$ ,  $f$  is the active slots coefficient and  $\phi_f$  is the mapping from Definition 1. Then  $\pi_{\text{SPoS}}$  proceeds as follows:

- Initialization.** The stakeholder  $U_i$  sends  $(\text{KeyGen}, sid, U_i)$  to  $\mathcal{F}_{\text{VRF}}, \mathcal{F}_{\text{KES}}$  and  $\mathcal{F}_{\text{DSIG}}$ ; receiving  $(\text{VerificationKey}, sid, v_i^{\text{ver}})$ ,  $(\text{VerificationKey}, sid, v_i^{\text{kes}})$  and  $(\text{VerificationKey}, sid, v_i^{\text{dsig}})$ , respectively. Then, in case it is the first round, it sends  $(\text{ver.keys}, sid, U_i, v_i^{\text{ver}}, v_i^{\text{kes}}, v_i^{\text{dsig}})$  to  $\mathcal{F}_{\text{BMR}}$  (to claim stake from the genesis block). In any case, it terminates the round by returning  $(U_i, v_i^{\text{ver}}, v_i^{\text{kes}}, v_i^{\text{dsig}})$  to  $\mathcal{Z}$ . In the next round,  $U_i$  sends  $(\text{genblock.req}, sid, U_i)$  to  $\mathcal{F}_{\text{BMR}}$ , receiving  $(\text{genblock}, sid, S_0, \eta)$  as the answer.  $U_i$  sets the local blockchain  $C = B_0 = (S_0, \eta)$  and its initial internal state  $st = H(B_0)$ .
- Chain Extension.** After initialization, for every slot  $sl_j \in S$ , every online stakeholder  $U_i$  performs the following steps:
  - $U_i$  receives from the environment the transaction data  $d \in \{0, 1\}^*$  to be inserted into the blockchain.
  - $U_i$  collects all valid chains received via diffusion into a set  $\mathbb{C}$ , pruning blocks belonging to future slots and verifying that for every chain  $\mathcal{C}' \in \mathbb{C}$  and every block  $B' = (st', d', sl', B'_s, \sigma_s) \in \mathcal{C}'$  it holds that the stakeholder who created it is in the slot leader set of slot  $sl'$  (by parsing  $B'_s$  as  $(U_s, y', \pi')$  for some  $s$ , verifying that  $\mathcal{F}_{\text{VRF}}$  responds to  $(\text{Verify}, sid, \eta \parallel d', y', \pi', v_s^{\text{ver}})$  by  $(\text{Verified}, sid, \eta \parallel d', y', \pi', 1)$ , and that  $y' < T_s$ ), and that  $\mathcal{F}_{\text{KES}}$  responds to  $(\text{Verify}, sid, (st', d', sl', B'_s), sl', \sigma_s, v_s^{\text{kes}})$  by  $(\text{Verified}, sid, (st', d', sl', B'_s), sl', 1)$ .  $U_i$  computes  $\mathcal{C}' = \text{maxvalid}(\mathbb{C}, \mathcal{C})$ , sets  $\mathcal{C}'$  as the new local chain and sets state  $st = H(\text{head}(\mathcal{C}'))$ .
  - $U_i$  sends  $(\text{EvalProve}, sid, \eta \parallel sl_j)$  to  $\mathcal{F}_{\text{VRF}}$ , receiving  $(\text{Evaluated}, sid, y, \pi)$ .  $U_i$  checks whether it is in the slot leader set of slot  $sl_j$  by checking that  $y < T_i$ . If yes, it generates a new block  $B = (st, d, sl_j, B_s, \sigma)$  where  $st$  is its current state,  $d \in \{0, 1\}^*$  is the transaction data,  $B_s = (U_i, y, \pi)$  and  $\sigma$  is a signature obtained by sending  $(\text{USign}, sid, U_i, (st, d, sl_j, B_s), sl_j)$  to  $\mathcal{F}_{\text{KES}}$  and receiving  $(\text{Signature}, sid, (st, d, sl_j, B_s), sl_j, \sigma)$ .  $U_i$  computes  $\mathcal{C}' = \mathcal{C} \mid B$ , sets  $\mathcal{C}'$  as the new local chain and sets state  $st = H(\text{head}(\mathcal{C}'))$ . Finally, if  $U_i$  has generated a block in this step, it diffuses  $\mathcal{C}'$ .
  - Signing Transactions.** Upon receiving  $(\text{sign.tx}, sid', tx)$  from the environment,  $U_i$  sends  $(\text{Sign}, sid, U_i, tx)$  to  $\mathcal{F}_{\text{DSIG}}$ , receiving  $(\text{Signature}, sid, tx, \sigma)$ . Then,  $U_i$  sends  $(\text{signed.tx}, sid', tx, \sigma)$  back to the environment.

Fig. 4: Protocol  $\pi_{\text{SPoS}}$ .

# From Mathematical Paper...

## Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain

Bernardo David\*, Peter Gaži\*\*, Aggelos Kiayias\*\*\*, and Alexander Russell†

October 6, 2017

**Abstract.** We present “Ouroboros Praos”, a proof-of-stake protocol, the first time, provides security against *fully-adaptive corruptions*: Specifically, the adversary can corrupt any part of the population of stakeholders at any moment as long as there is an honest majority of stake; furthermore, the protocol has message delivery delay unknown to protocol participants. To achieve these guarantees we formalize and realize in the suitable form of forward secure digital signatures and a new that maintains unpredictability under malicious key generation a general combinatorial framework for the analysis of semi-independent interest. We prove our protocol see assumptions in the random oracle model.

The protocol  $\pi_{\text{SPoS}}$  is run by stakeholders  $U_1, \dots, U_n$  interacting among themselves and with ideal functionalities  $\mathcal{F}_{\text{BMR}}, \mathcal{F}_{\text{VRF}}, \mathcal{F}_{\text{KES}}, \mathcal{F}_{\text{DSIG}}, \mathcal{H}$  over a sequence of slots  $S = \{sl_1, \dots, sl_R\}$ . Define  $T_i \triangleq 2^{\ell_{\text{VRF}}} \phi_f(\alpha_i)$  as the threshold for a stakeholder  $U_i$ , where  $\alpha_i$  is the relative stake of  $U_i$ ,  $\ell_{\text{VRF}}$  denotes the output length of  $\mathcal{F}_{\text{VRF}}$ ,  $f$  is the active slots coefficient and  $\phi_f$  is the mapping from Definition 1. Then  $\pi_{\text{SPoS}}$  proceeds as follows:

1. **Initialization.** The stakeholder  $U_i$  sends  $(\text{KeyGen}, sid, U_i)$  to  $\mathcal{F}_{\text{VRF}}, \mathcal{F}_{\text{KES}}$  and  $\mathcal{F}_{\text{DSIG}}$ ; receiving  $(\text{VerificationKey}, sid, v_i^{\text{ver}})$ ,  $(\text{VerificationKey}, sid, v_i^{\text{kes}})$  and  $(\text{VerificationKey}, sid, v_i^{\text{dsig}})$ , respectively. Then, in case it is the first round, it sends  $(\text{ver.keys}, sid, U_i, v_i^{\text{ver}}, v_i^{\text{kes}}, v_i^{\text{dsig}})$  to  $\mathcal{F}_{\text{BMR}}$  (to claim stake from the genesis block). In any case, it terminates the round by returning  $(U_i, v_i^{\text{ver}}, v_i^{\text{kes}}, v_i^{\text{dsig}})$  to  $\mathcal{Z}$ . In the next round,  $U_i$  sends  $(\text{genblock.req}, sid, U_i)$  to  $\mathcal{F}_{\text{BMR}}$ , receiving  $(\text{genblock}, sid, S_0, \eta)$  as the answer.  $U_i$  sets the local blockchain  $C = B_0 = (S_0, \eta)$  and its initial internal state  $st = H(B_0)$ .
2. **Chain Extension.** After initialization, for every slot  $sl_j \in S$ , every online stakeholder  $U_i$  performs the following steps:
  - (a)  $U_i$  receives from the environment the transaction data  $d \in \{0, 1\}^*$  to be inserted into the blockchain.
  - (b)  $U_i$  collects all valid chains received via diffusion into a set  $\mathcal{C}$ , pruning blocks belonging to future slots and verifying that for every chain  $\mathcal{C}' \in \mathcal{C}$  and every block  $B' = (st', d', sl', B_{sl'}, \sigma_{sl'}) \in \mathcal{C}'$  it holds that the stakeholder who created it is the slot leader set of slot  $sl'$  (by parsing  $B_{sl'}$  as  $(U_s, y', \pi')$  for some  $s$ , verifying that  $\mathcal{F}_{\text{VRF}}$  responds to  $(\text{Verify}, sid, \eta \parallel d', y', \pi', v_s^{\text{ver}})$  by  $(\text{Verified}, sid, \eta \parallel d', y', \pi', 1)$ , and that  $y' < T_s$ ), and that  $\mathcal{F}_{\text{KES}}$  responds to  $(\text{Verify}, sid, (st', d', sl', B_{sl'}), sl', \sigma_{sl'}, v_{sl'}^{\text{kes}})$  by  $(\text{Verified}, sid, (st', d', sl', B_{sl'}), sl', 1)$ .  $U_i$  computes  $\mathcal{C}' = \text{maxvalid}(\mathcal{C}, \mathcal{C})$ , sets  $\mathcal{C}'$  as the new local chain and sets state  $st = H(\text{head}(\mathcal{C}'))$ .
  - (c)  $U_i$  sends  $(\text{EvalProve}, sid, \eta \parallel sl_j)$  to  $\mathcal{F}_{\text{VRF}}$ , receiving  $(\text{Evaluated}, sid, y, \pi)$ .  $U_i$  checks whether it is in the slot leader set of slot  $sl_j$  by checking that  $y < T_i$ . If yes, it generates a new block  $B = (st, d, sl_j, B_{slj}, \sigma)$  where  $st$  is its current state,  $d \in \{0, 1\}^*$  is the transaction data,  $B_{slj} = (U_i, y, \pi)$  and  $\sigma$  is a signature obtained by sending  $(\text{USign}, sid, U_i, (st, d, sl_j, B_{slj}), sl_j)$  to  $\mathcal{F}_{\text{KES}}$  and receiving  $(\text{Signature}, sid, (st, d, sl_j, B_{slj}), sl_j, \sigma)$ .  $U_i$  computes  $\mathcal{C}' = \mathcal{C} \mid B$ , sets  $\mathcal{C}'$  as the new local chain and sets state  $st = H(\text{head}(\mathcal{C}'))$ . Finally, if  $U_i$  has generated a block in this step, it diffuses  $\mathcal{C}'$ .
3. **Signing Transactions.** Upon receiving  $(\text{sign.tx}, sid', tx)$  from the environment,  $U_i$  sends  $(\text{Sign}, sid, U_i, tx)$  to  $\mathcal{F}_{\text{DSIG}}$ , receiving  $(\text{Signature}, sid, tx, \sigma)$ . Then,  $U_i$  sends  $(\text{signed.tx}, sid', tx, \sigma)$  back to the environment.

Fig. 4: Protocol  $\pi_{\text{SPoS}}$ .

- Written in English.
- Written by mathematicians.

# From Mathematical Paper...

## Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain

Bernardo David\*, Peter Gaži\*\*, Aggelos Kiayias\*\*\*, and Alexander Russell†

October 6, 2017

**Abstract.** We present “Ouroboros Praos”, a proof-of-stake blockchain that, for the first time, provides security against *fully-adaptive corruptions*. Specifically, the adversary can corrupt any part of the population of stakeholders at any moment as long as there is an honest majority of stake; furthermore, the protocol tolerates message delivery delay unknown to protocol participants. To achieve these guarantees we formalize and realize in the suitable form of forward secure digital signatures and a new proof system that maintains unpredictability under malicious key generation. A general combinatorial framework for the analysis of semi-synchronous systems may be of independent interest. We prove our protocol secure in the random oracle model.

1. **Initialization.** The stakeholder  $U_i$  sends  $(\text{KeyGen}, sid, U_i)$  to  $\mathcal{F}_{\text{VER}}$ ,  $\mathcal{F}_{\text{PKS}}$  and  $\mathcal{F}_{\text{OSIG}}$ ; receiving  $(\text{VerificationKey}, sid, v_i^{\text{ver}})$ ,  $(\text{VerificationKey}, sid, v_i^{\text{kes}})$  and  $(\text{VerificationKey}, sid, v_i^{\text{osig}})$ , respectively. Then, in case it is the first round, it sends  $(\text{ver.keys}, sid, U_i, v_i^{\text{ver}}, v_i^{\text{kes}}, v_i^{\text{osig}})$  to  $\mathcal{F}_{\text{MUR}}$  (to claim stake from the genesis block). In any case, it terminates the round by returning  $(U_i, v_i^{\text{ver}}, v_i^{\text{kes}}, v_i^{\text{osig}})$  to  $\mathcal{Z}$ . In the next round,  $U_i$  sends  $(\text{genblock.req}, sid, U_i)$  to  $\mathcal{F}_{\text{MUR}}$ , receiving  $(\text{genblock}, sid, S_0, \eta)$  as the answer.  $U_i$  sets the local blockchain  $C = B_0 = (S_0, \eta)$  and its initial internal state  $st = H(B_0)$ .
2. **Chain Extension.** After initialization, for every slot  $sl_j \in S$ , every online stakeholder  $U_i$  performs the following steps:
- $U_i$  receives from the environment the transaction data  $d \in \{0, 1\}^*$  to be inserted into the blockchain.
  - $U_i$  collects all valid chains received via diffusion into a set  $\mathbb{C}$ , pruning blocks belonging to future slots and verifying that for every chain  $\mathcal{C}' \in \mathbb{C}$  and every block  $B' = (st', d', sl', B_{sl'}, \sigma_{sl'}) \in \mathcal{C}'$  it holds that the stakeholder who created it is in the slot leader set of slot  $sl'$  (by parsing  $B_{sl'}$  as  $(U_s, y', \pi')$  for some  $s$ , verifying that  $\mathcal{F}_{\text{VER}}$  responds to  $(\text{Verify}, sid, \eta \parallel d', y', \pi', v_s^{\text{ver}})$  by  $(\text{Verified}, sid, \eta \parallel d', y', \pi', 1)$ , and that  $y' < T_s$ ), and that  $\mathcal{F}_{\text{KES}}$  responds to  $(\text{Verify}, sid, (st', d', sl', B_{sl'}), sl', \sigma_{sl'}, v_s^{\text{kes}})$  by  $(\text{Verified}, sid, (st', d', sl', B_{sl'}), sl', 1)$ .  $U_i$  computes  $\mathcal{C}' = \text{maxvalid}(\mathbb{C}, \mathcal{C})$ , sets  $\mathcal{C}'$  as the new local chain and sets state  $st = H(\text{head}(\mathcal{C}'))$ .
  - $U_i$  sends  $(\text{EvalProve}, sid, \eta \parallel sl_j)$  to  $\mathcal{F}_{\text{VER}}$ , receiving  $(\text{Evaluated}, sid, y, \pi)$ .  $U_i$  checks whether it is in the slot leader set of slot  $sl_j$  by checking that  $y < T_i$ . If yes, it generates a new block  $B = (st, d, sl_j, B_{slj}, \sigma)$  where  $st$  is its current state,  $d \in \{0, 1\}^*$  is the transaction data,  $B_{slj} = (U_i, y, \pi)$  and  $\sigma$  is a signature obtained by sending  $(\text{USign}, sid, U_i, (st, d, sl_j, B_{slj}), sl_j)$  to  $\mathcal{F}_{\text{KES}}$  and receiving  $(\text{Signature}, sid, (st, d, sl_j, B_{slj}), sl_j, \sigma)$ .  $U_i$  computes  $\mathcal{C}' = \mathcal{C} \mid B$ , sets  $\mathcal{C}'$  as the new local chain and sets state  $st = H(\text{head}(\mathcal{C}'))$ . Finally, if  $U_i$  has generated a block in this step, it diffuses  $\mathcal{C}'$ .
  - Signing Transactions.** Upon receiving  $(\text{sign.tx}, sid', tx)$  from the environment,  $U_i$  sends  $(\text{Sign}, sid, U_i, tx)$  to  $\mathcal{F}_{\text{OSIG}}$ , receiving  $(\text{Signature}, sid, tx, \sigma)$ . Then,  $U_i$  sends  $(\text{signed.tx}, sid', tx, \sigma)$  back to the environment.

Fig. 4: Protocol  $\pi_{\text{SPoS}}$ .

- Written in English.
- Written by mathematicians.
- Very abstract.

## ... To Efficient Code

```
235  -- CHECK: @verifyEncShare
236  -- | Verify encrypted shares
237  verifyEncShares
238      :: MonadRandom m
239      => SecretProof
240      -> Scrape.Threshold
241      -> [(VssPublicKey, EncShare)]
242      -> m Bool
243  verifyEncShares SecretProof{..} threshold (sortWith fst -> pairs)
244      | threshold <= 1     = error "verifyEncShares: threshold must be > 1"
245      | threshold >= n - 1 = error "verifyEncShares: threshold must be < n-1"
246      | otherwise =
247          Scrape.verifyEncryptedShares
248              spExtraGen
249              threshold
250              spCommitments
251              spParallelProofs
252              (coerce $ map snd pairs) -- shares
253              (coerce $ map fst pairs) -- participants
254  where
255      n = fromIntegral (length pairs)
```

# ... To Efficient Code

- Written in Haskell.

```
235  -- CHECK: @verifyEncShare
236  -- | Verify encrypted shares
237  verifyEncShares
238      :: MonadRandom m
239      => SecretProof
240      -> Scrape.Threshold
241      -> [(VssPublicKey, EncShare)]
242      -> m Bool
243  verifyEncShares SecretProof{..} threshold (sortWith fst -> pairs)
244      | threshold <= 1     = error "verifyEncShares: threshold must be > 1"
245      | threshold >= n - 1 = error "verifyEncShares: threshold must be < n-1"
246      | otherwise =
247          Scrape.verifyEncryptedShares
248              spExtraGen
249              threshold
250              spCommitments
251              spParallelProofs
252              (coerce $ map snd pairs) -- shares
253              (coerce $ map fst pairs) -- participants
254  where
255      n = fromIntegral (length pairs)
```

# ... To Efficient Code

- Written in Haskell.
- Written by Software Engineers.

```
235  -- CHECK: @verifyEncShare
236  -- | Verify encrypted shares
237  verifyEncShares
238      :: MonadRandom m
239      => SecretProof
240      -> Scrape.Threshold
241      -> [(VssPublicKey, EncShare)]
242      -> m Bool
243  verifyEncShares SecretProof{..} threshold (sortWith fst -> pairs)
244      | threshold <= 1     = error "verifyEncShares: threshold must be > 1"
245      | threshold >= n - 1 = error "verifyEncShares: threshold must be < n-1"
246      | otherwise =
247          Scrape.verifyEncryptedShares
248              spExtraGen
249              threshold
250              spCommitments
251              spParallelProofs
252              (coerce $ map snd pairs) -- shares
253              (coerce $ map fst pairs) -- participants
254  where
255      n = fromIntegral (length pairs)
```

## ... To Efficient Code

- Written in Haskell.
- Written by Software Engineers.
- Efficient code.

```
235  -- CHECK: @verifyEncShare
236  -- | Verify encrypted shares
237  verifyEncShares
238      :: MonadRandom m
239      => SecretProof
240      -> Scrape.Threshold
241      -> [(VssPublicKey, EncShare)]
242      -> m Bool
243  verifyEncShares SecretProof{..} threshold (sortWith fst -> pairs)
244      | threshold <= 1     = error "verifyEncShares: threshold must be > 1"
245      | threshold >= n - 1 = error "verifyEncShares: threshold must be < n-1"
246      | otherwise =
247          Scrape.verifyEncryptedShares
248          spExtraGen
249          threshold
250          spCommitments
251          spParallelProofs
252          (coerce $ map snd pairs) -- shares
253          (coerce $ map fst pairs) -- participants
254  where
255      n = fromIntegral (length pairs)
```

# The Problem

- We start from a mathematical paper written by mathematicians.

# The Problem

- We start from a mathematical paper written by mathematicians.
- The paper will undergo rigid peer review and contain mathematical proofs of correctness.

# The Problem

- We start from a mathematical paper written by mathematicians.
- The paper will undergo rigid peer review and contain mathematical proofs of correctness.
- The outcome should be correct and efficient Haskell code.

# The Problem

- We start from a mathematical paper written by mathematicians.
- The paper will undergo rigid peer review and contain mathematical proofs of correctness.
- The outcome should be correct and efficient Haskell code.
- Our mathematicians do not know Haskell, our engineers do not know cryptography.

# The Problem

- We start from a mathematical paper written by mathematicians.
- The paper will undergo rigid peer review and contain mathematical proofs of correctness.
- The outcome should be correct and efficient Haskell code.
- Our mathematicians do not know Haskell, our engineers do not know cryptography.
- How can we guarantee we deploy code that faithfully implements the original paper?

# Why does it matter?

- We are very proud of the quality of our research branch. We want to ensure this quality translates into equal quality of our software.

# Why does it matter?

- We are very proud of the quality of our research branch. We want to ensure this quality translates into equal quality of our software.
- Literally billions of dollars are managed by our code. A single mistake can be extremely costly.

# Why does it matter?

- We are very proud of the quality of our research branch. We want to ensure this quality translates into equal quality of our software.
- Literally billions of dollars are managed by our code. A single mistake can be extremely costly.
- We are interested in developing best practices that can be applied to a wide range of domains, pushing the envelope of what is possible and practicable.

# The Solution: Formal Methods

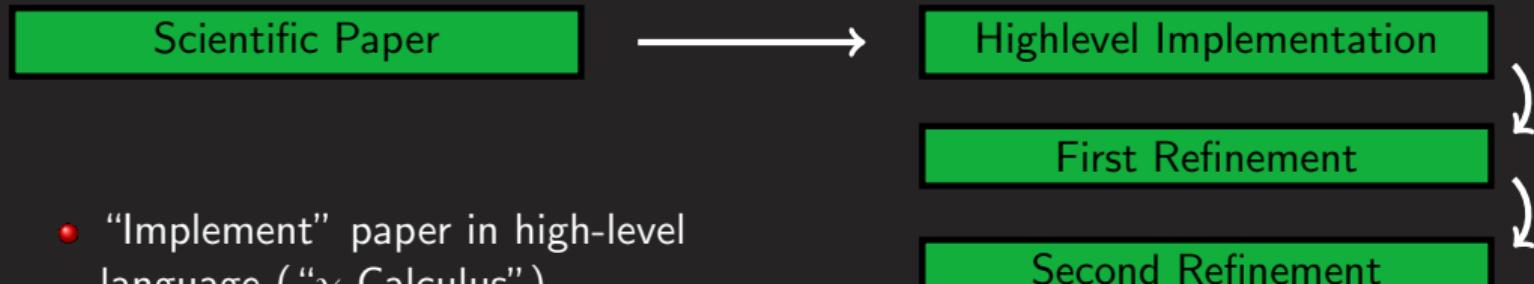
Scientific Paper

# The Solution: Formal Methods



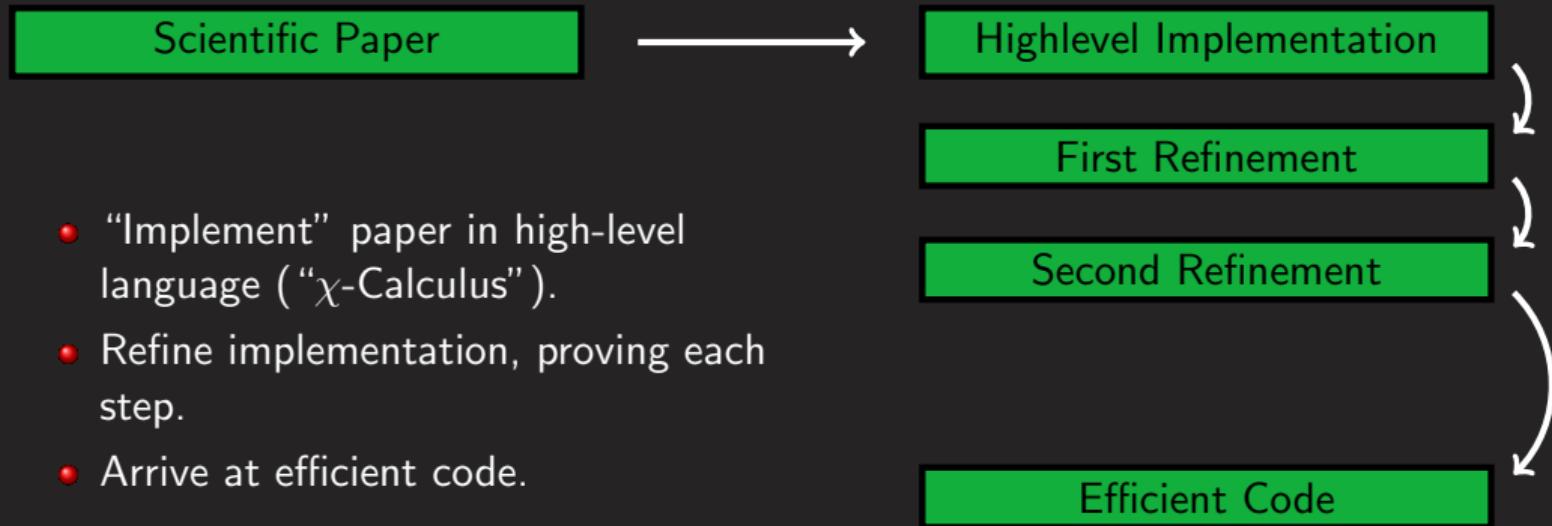
- “Implement” paper in high-level language (“ $\chi$ -Calculus”).

# The Solution: Formal Methods



- “Implement” paper in high-level language (“ $\chi$ -Calculus”).
- Refine implementation, proving each step.

# The Solution: Formal Methods



# $\chi$ -Calculus

- Our version of the  $\pi$ - (or  $\psi$ -) Calculus (like  $\lambda$ -Calculus, but for concurrent systems).

# $\chi$ -Calculus

- Our version of the  $\pi$ - (or  $\psi$ -) Calculus (like  $\lambda$ -Calculus, but for concurrent systems).
- Can be embedded in Haskell and then...

# $\chi$ -Calculus

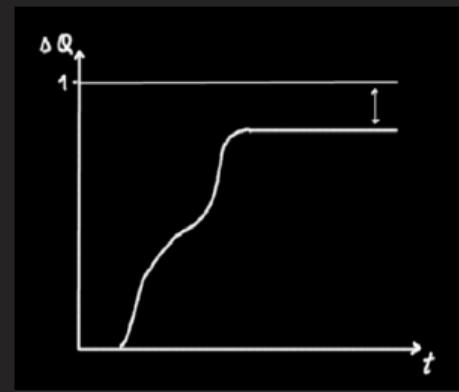
- Our version of the  $\pi$ - (or  $\psi$ -) Calculus (like  $\lambda$ -Calculus, but for concurrent systems).
- Can be embedded in Haskell and then...
  - ... be executed.

# $\chi$ -Calculus

- Our version of the  $\pi$ - (or  $\psi$ -) Calculus (like  $\lambda$ -Calculus, but for concurrent systems).
- Can be embedded in Haskell and then...
  - ... be executed.
  - ... be exported to a proof assistant.

# $\chi$ -Calculus

- Our version of the  $\pi$ - (or  $\psi$ -) Calculus (like  $\lambda$ -Calculus, but for concurrent systems).
- Can be embedded in Haskell and then...
  - ... be executed.
  - ... be exported to a proof assistant.
  - ... be analyzed for performance ( $\Delta Q$ ).



# Incentives

# The people doing all the hard work. . .



Prof. Aggelos Kiayias, University of Edinburgh (UK), Chief Scientist at IOHK.



Prof. Elias Koutsoupias, University of Oxford (UK), Senior Research Fellow at IOHK.



Aikaterini-Panagiota Stouka, University of Edinburgh (UK), Researcher at IOHK.

# What are Incentives?

- Incentives in the context of a cryptocurrency are ways of encouraging people to participate in the protocol and to follow it faithfully.

# What are Incentives?

- Incentives in the context of a cryptocurrency are ways of encouraging people to participate in the protocol and to follow it faithfully.
- In the case of Bitcoin, this means mining blocks and including as many valid transactions in those blocks as possible.

# What are Incentives?

- Incentives in the context of a cryptocurrency are ways of encouraging people to participate in the protocol and to follow it faithfully.
- In the case of Bitcoin, this means mining blocks and including as many valid transactions in those blocks as possible.
- In the case of Cardano, it means being online and creating a block when they have been elected slot leader and to participate in the election process.

# (Non-) Monetary Incentives

- When the Bitcoin mining pool Ghash.io accumulated 42% of total mining power, people voluntarily started leaving the pool and brought it down to 38% in only two days. (CoinDesk, 2014–01–09)

# (Non-) Monetary Incentives

- When the Bitcoin mining pool Ghash.io accumulated 42% of total mining power, people voluntarily started leaving the pool and brought it down to 38% in only two days. (CoinDesk, 2014–01–09)
- The people who left Ghash.io did not receive any Bitcoin for leaving. Rather, they believed that concentrating too much mining power was bad and that leaving was the right thing to do.

# (Non-) Monetary Incentives

- When the Bitcoin mining pool Ghash.io accumulated 42% of total mining power, people voluntarily started leaving the pool and brought it down to 38% in only two days. (CoinDesk, 2014–01–09)
- The people who left Ghash.io did not receive any Bitcoin for leaving. Rather, they believed that concentrating too much mining power was bad and that leaving was the right thing to do.
- Ideally, monetary and moral incentives should align perfectly.

# Incentives in Cardano

- The above example shows that in Bitcoin, this ideal is not always achieved. Sometimes people have to choose between doing the morally right thing and pursuing their financial gain.

# Incentives in Cardano

- The above example shows that in Bitcoin, this ideal is not always achieved. Sometimes people have to choose between doing the morally right thing and pursuing their financial gain.
- In Cardano, we strive for perfect alignment of incentives.

# Incentives in Cardano

- The above example shows that in Bitcoin, this ideal is not always achieved. Sometimes people have to choose between doing the morally right thing and pursuing their financial gain.
- In Cardano, we strive for perfect alignment of incentives.
- We use **Game Theory** and **Simulations** to develop and test our model.



# Smart Contracts

# IELE and K-Framework



- Prof. Grigore Roșu, University of Illinois in Urbana-Champaign (US), CEO of Runtime Verification.
- K-Framework: meta framework for specifying formal semantics of programming languages.
- IELE: formally specified smart-contract language.



- Prof. Phil Wadler, University of Edinburgh (UK), Senior Research Fellow and Area Leader Programming Languages at IOHK.
- Dr. Manuel Chakravarty, Language Architect at IOHK.
- Plutus: newly developed smart-contract language heavily inspired by Haskell.

# Marlowe



- Prof. Simon Thompson, University of Canterbury (UK), Senior Research Fellow at IOHK.
- Marlowe: newly developed smart-contract language for financial contracts.

# Haskell

- Statically typed: Every expression has a type at compile time.

- Statically typed: Every expression has a type at compile time.
- Lazy: Expressions are evaluated only when needed.

- Statically typed: Every expression has a type at compile time.
- Lazy: Expressions are evaluated only when needed.
- Pure: Side effects (I/O) are visible in the types.

- Statically typed: Every expression has a type at compile time.
- Lazy: Expressions are evaluated only when needed.
- Pure: Side effects (I/O) are visible in the types.
- Extremely expressive type system.

# Ouroboros BFT in Haskell — Commands

```
-- | Used to specify the length of a 'Delay'.
type Seconds = Double

-- | 'Command' is a simple DSL for the description of processes that can
-- communicate with each other via /broadcast/.
data Command =
    Stop
  | Delay Seconds Command
  | Broadcast String Command
  | Receive (String -> Command)
  | Say String Command
```

# Ouroboros BFT in Haskell — Commands

```
-- | Used to specify the length of a 'Delay'.
type Seconds = Double

-- | 'Command' is a simple DSL for the description of processes that can
-- communicate with each other via /broadcast/.
data Command =
    Stop
  | Delay Seconds Command
  | Broadcast String Command
  | Receive (String -> Command)
  | Say String Command
```

## Remark

The abstract `Command` type allows writing the protocol as a pure value in an ordinary Haskell data type. This can then later be interpreted in different ways.

# Ouroboros BFT in Haskell — Supporting Types

```
type Slot = Int
type NodeIndex = Int

data Block = Block
{ blSlot      :: !Slot
, blNodeIndex :: !NodeIndex
} deriving (Show, Read)

infixl 5 :>

data Chain =
  Genesis
| Chain :> Block
deriving (Show, Read)

data Message =
  Tick Int
| NewChain Chain
deriving (Show, Read)
```

# Ouroboros BFT in Haskell — Helper Functions

```
chainLength :: Chain -> Int
```

```
chainLength Genesis = 0
```

```
chainLength (c :> _) = 1 + chainLength c
```

```
slotLeader :: Int -> Slot -> NodeIndex
```

```
slotLeader nodeCount s = 1 + mod (s - 1) nodeCount
```

```
isValidChain :: Int -> Slot -> Chain -> Bool
```

```
isValidChain _ _ Genesis = True
```

```
isValidChain nodeCount s (c :> b) =
```

```
    ( blSlot b <= s)
```

```
    && ( blSlot b >= 1)
```

```
    && ( slotLeader nodeCount (blSlot b) == blNodeIndex b)
```

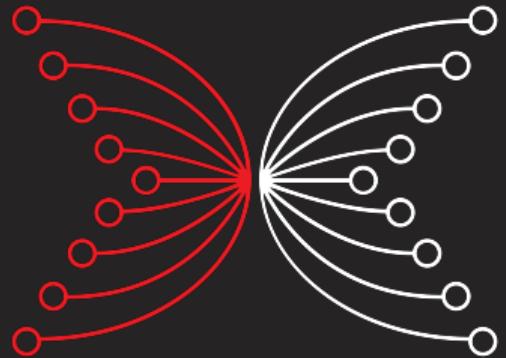
```
    && (isValidChain nodeCount (blSlot b - 1) c)
```

# Ouroboros BFT in Haskell — Ticker

```
ticker :: Seconds -> Command
ticker interval = go 0
where
  go :: Int -> Command
  go i =
    let j    = i + 1
        msg = show $ Tick j
    in Delay interval $ Broadcast msg $ Say ("tick " ++ show j) $ go j
```

# Ouroboros BFT in Haskell — The Protocol

```
bft :: Int -> NodeIndex -> Command
bft nodeCount i = go Genesis 0
  where
    go :: Chain -> Slot -> Command
    go c s = Receive $ \msg -> case read msg of
      Tick s'
        | s' > s ->
          Say ("entered slot " ++ show s') $
          if slotLeader nodeCount s' == i -- Am I leader?
            then let b    = Block s' i
                  c'   = c :> b
                  msg' = show $ NewChain c'
                  in  Say ("created " ++ show c') $ Broadcast msg' $ go c' s'
            else go c s'
      NewChain c'
        | (isValidChain nodeCount s c') && (chainLength c' > chainLength c) ->
          Say ("adopted chain " ++ show c') $ go c' s
        | chainLength c' <= chainLength c ->
          Say "rejected chain - too short" $ go c s
        | otherwise ->
          Say "rejected chain - invalid" $ go c s
    - -> go c s
```



INPUT | OUTPUT

# Smart Contracts

## Smart Contracts & Bitcoin-Script

Lars Brünjes



January 9 2020

## Reminder: The Simple UTxO-Model

# The Simple UTxO-Model

- UTxO is an acronym for **Unspent Transaction Output**.

# The Simple UTxO-Model

- UTxO is an acronym for **Unspent Transaction Output**.
- As in the **account-based** model (employed for example by Ethereum), the UTxO-model uses **(hashes of) public keys** as addresses.

# The Simple UTxO-Model

- UTxO is an acronym for **Unspent Transaction Output**.
- As in the **account-based** model (employed for example by Ethereum), the UTxO-model uses **(hashes of) public keys** as addresses.
- A **transaction** has UTxOs as **inputs** and one or more **outputs**.

# The Simple UTxO-Model

- UTxO is an acronym for **Unspent Transaction Output**.
- As in the **account-based** model (employed for example by Ethereum), the UTxO-model uses **(hashes of) public keys** as addresses.
- A **transaction** has UTxOs as **inputs** and one or more **outputs**.
- Transactions are authorized by the **digital signatures** of the owners of the inputs.

# The Simple UTxO-Model

- UTxO is an acronym for **Unspent Transaction Output**.
- As in the **account-based** model (employed for example by Ethereum), the UTxO-model uses **(hashes of) public keys** as addresses.
- A **transaction** has UTxOs as **inputs** and one or more **outputs**.
- Transactions are authorized by the **digital signatures** of the owners of the inputs.
- Transactions with several senders and/or receivers are possible.

# The Simple UTxO-Model

- UTxO is an acronym for **Unspent Transaction Output**.
- As in the **account-based** model (employed for example by Ethereum), the UTxO-model uses **(hashes of) public keys** as addresses.
- A **transaction** has UTxOs as **inputs** and one or more **outputs**.
- Transactions are authorized by the **digital signatures** of the owners of the inputs.
- Transactions with several senders and/or receivers are possible.
- The **state** of the blockchain is determined by the set of all UTxOs.

# The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus **transaction fees**) must equal the sum of all outputs.

# The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus **transaction fees**) must equal the sum of all outputs.
- Each transaction completely spends all its inputs. If the sum of inputs is too large, an output has to be created for the change.

# The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus **transaction fees**) must equal the sum of all outputs.
- Each transaction completely spends all its inputs. If the sum of inputs is too large, an output has to be created for the change.
- A transaction consists of:
  - A set of inputs (UTxOs).

# The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus **transaction fees**) must equal the sum of all outputs.
- Each transaction completely spends all its inputs. If the sum of inputs is too large, an output has to be created for the change.
- A transaction consists of:
  - A set of inputs (UTxOs).
  - An **ordered list** of outputs, where each output has an **address** and a **value**.

## The UTxO-Model (continued)

- For each transaction, the sum of all inputs (plus **transaction fees**) must equal the sum of all outputs.
  - Each transaction completely spends all its inputs. If the sum of inputs is too large, an output has to be created for the change.
  - A transaction consists of:
    - A set of inputs (UTxOs).
    - An **ordered list** of outputs, where each output has an **address** and a **value**.
    - A digital signature for each input.

# Transaction Validity in the UTxO-Model

A transaction in the UTxO-model is **valid** if the following three conditions are satisfied:

# Transaction Validity in the UTxO-Model

A transaction in the UTxO-model is **valid** if the following three conditions are satisfied:

- The transaction contains digital signature's belonging to the owners of the inputs.

# Transaction Validity in the UTxO-Model

A transaction in the UTxO-model is **valid** if the following three conditions are satisfied:

- The transaction contains digital signature's belonging to the owners of the inputs.
- The sum of all input values (plus transaction fees) equals the sum of all outputs.

# Transaction Validity in the UTxO-Model

A transaction in the UTxO-model is **valid** if the following three conditions are satisfied:

- The transaction contains digital signature's belonging to the owners of the inputs.
- The sum of all input values (plus transaction fees) equals the sum of all outputs.
- No output value is negative.

# Example: A Simple Transaction

Alice holds 100  $\text{฿}$  and wants to send 40  $\text{฿}$  to Bob, who has 50  $\text{฿}$ .

Alice

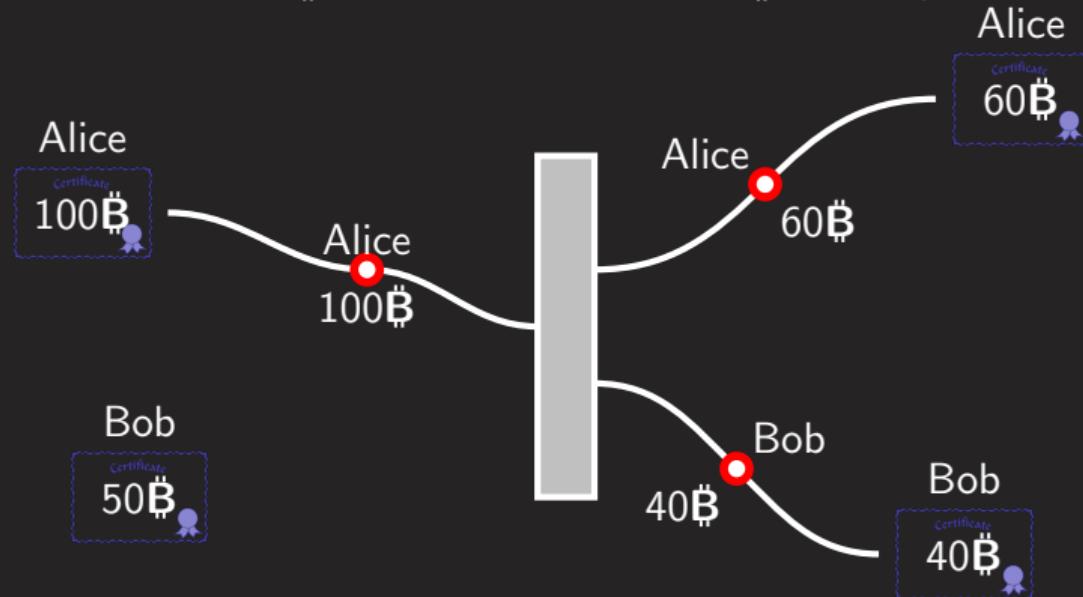


Bob



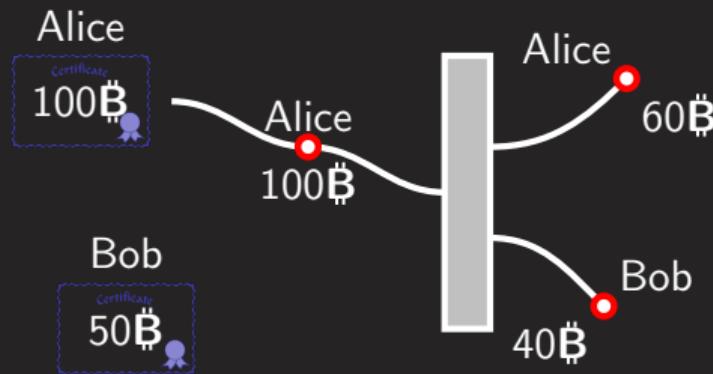
## Example: A Simple Transaction

Alice holds 100  $\text{฿}$  and wants to send 40  $\text{฿}$  to Bob, who has 50  $\text{฿}$ .



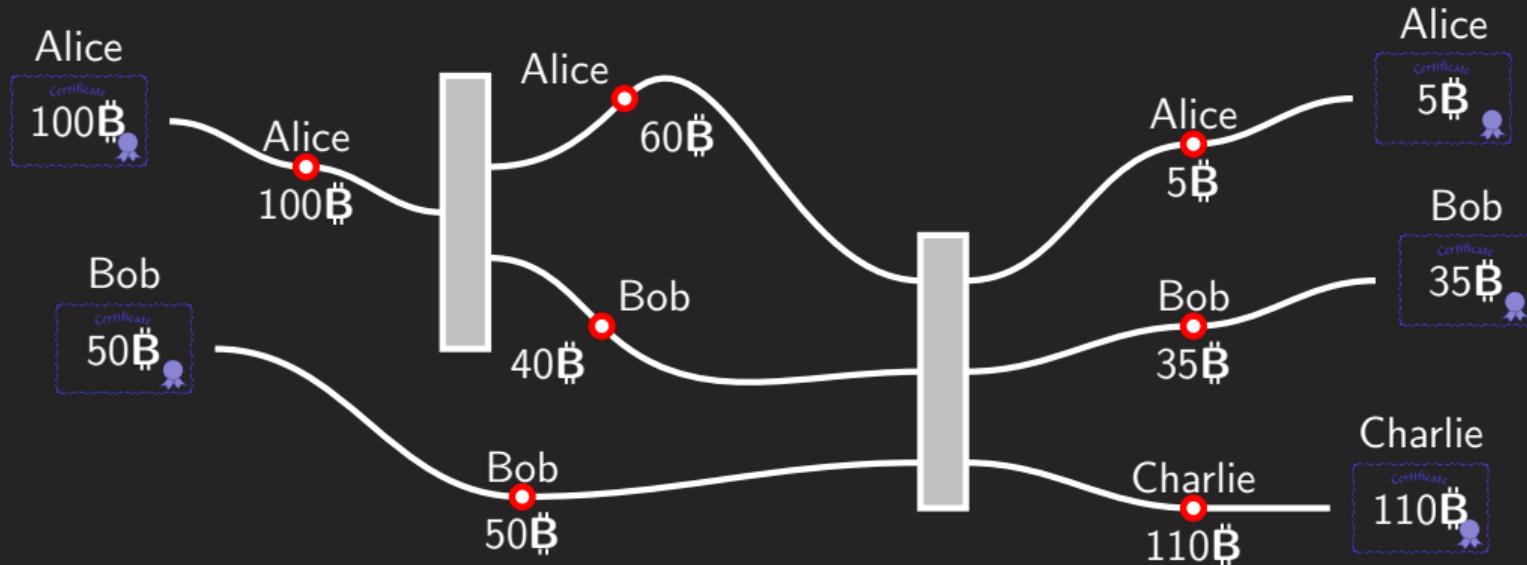
## Example: A More Complex Transaction

After sending 40 ₿ to Bob, Alice and Bob want to send 55 ₿ each to Charlie.



## Example: A More Complex Transaction

After sending 40  $\text{฿}$  to Bob, Alice and Bob want to send 55  $\text{฿}$  each to Charlie.



## Reminder: Smart Contracts

# Reminder: Smart Contracts

- **Smart Contracts** are contracts of varying complexity “on the blockchain”.

## Reminder: Smart Contracts

- **Smart Contracts** are contracts of varying complexity “on the blockchain”.
- Whereas conventional contracts are enforced by the court system, smart contracts are automatically enforced (like usual transactions in a cryptocurrency).

# Reminder: Smart Contracts

- **Smart Contracts** are contracts of varying complexity “on the blockchain”.
- Whereas conventional contracts are enforced by the court system, smart contracts are automatically enforced (like usual transactions in a cryptocurrency).
- Traditional contracts are often ambiguous and open to interpretation, whereas smart contracts are unambiguously defined with mathematical precision.

# Reminder: Smart Contracts

- Smart Contracts are contracts of varying complexity “on the blockchain”.
- Whereas conventional contracts are enforced by the court system, smart contracts are automatically enforced (like usual transactions in a cryptocurrency).
- Traditional contracts are often ambiguous and open to interpretation, whereas smart contracts are unambiguously defined with mathematical precision.
- Whether this is good or bad is up to discussion: Courts and human judges are fallible, but they are also able to use common sense to see the *intent* of a contract without sticking to the words.

## Reminder: Smart Contracts

- **Smart Contracts** are contracts of varying complexity “on the blockchain”.
  - Whereas conventional contracts are enforced by the court system, smart contracts are automatically enforced (like usual transactions in a cryptocurrency).
  - Traditional contracts are often ambiguous and open to interpretation, whereas smart contracts are unambiguously defined with mathematical precision.
  - Whether this is good or bad is up to discussion: Courts and human judges are fallible, but they are also able to use common sense to see the *intent* of a contract without sticking to the words.
  - In this sense, smart contracts are *only* “words”: Intent does not matter. All that matters is the actual code.

# Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, **addresses** (which identify senders and receivers) are simply (hashes of) public keys.

# Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, **addresses** (which identify senders and receivers) are simply (hashes of) public keys.
- Reality is more complicated than this even for Bitcoin, let alone for Ethereum.

# Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, **addresses** (which identify senders and receivers) are simply (hashes of) public keys.
- Reality is more complicated than this even for Bitcoin, let alone for Ethereum.
- Instead of static hashes for addresses, Bitcoin uses little programs called **scripts**.

# Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, **addresses** (which identify senders and receivers) are simply (hashes of) public keys.
- Reality is more complicated than this even for Bitcoin, let alone for Ethereum.
- Instead of static hashes for addresses, Bitcoin uses little programs called **scripts**.
- While verifying the inputs of a Bitcoin transaction, input- and output-scripts are combined and executed. The result from this execution decides whether the spending transaction is entitled to spend the output.

# Basic Idea of Smart Contracts

- For usual transactions in a cryptocurrency, **addresses** (which identify senders and receivers) are simply (hashes of) public keys.
- Reality is more complicated than this even for Bitcoin, let alone for Ethereum.
- Instead of static hashes for addresses, Bitcoin uses little programs called **scripts**.
- While verifying the inputs of a Bitcoin transaction, input- and output-scripts are combined and executed. The result from this execution decides whether the spending transaction is entitled to spend the output.
- Details depend on the specific cryptocurrency, but the principle stays the same: **Programs decide under which circumstances money may be spent.**

# Flavours of Smart Contracts

# Dilemma

- The programs which decide transaction validity must run on all nodes.

# Dilemma

- The programs which decide transaction validity must run on all nodes.
- It would therefore be fatal if such programs ran for a very long time or could get stuck in an infinite loop.

# Dilemma

- The programs which decide transaction validity must run on all nodes.
- It would therefore be fatal if such programs ran for a very long time or could get stuck in an infinite loop.
- One way out is to restrict the set of possible programs to make infinite loops impossible, which has the disadvantage of severely restricting how powerful such programs can be.

# Dilemma

- The programs which decide transaction validity must run on all nodes.
- It would therefore be fatal if such programs ran for a very long time or could get stuck in an infinite loop.
- One way out is to restrict the set of possible programs to make infinite loops impossible, which has the disadvantage of severely restricting how powerful such programs can be.
- Another option is to allow arbitrarily complex programs, but to make their execution “expensive”: The initiator of a transaction has to pay a fee for each step the programs takes.

## Dilemma

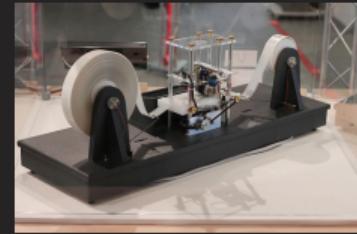
- The programs which decide transaction validity must run on all nodes.
  - It would therefore be fatal if such programs ran for a very long time or could get stuck in an infinite loop.
  - One way out is to restrict the set of possible programs to make infinite loops impossible, which has the disadvantage of severely restricting how powerful such programs can be.
  - Another option is to allow arbitrarily complex programs, but to make their execution “expensive”: The initiator of a transaction has to pay a fee for each step the programs takes.
  - The creators of Bitcoin have chosen the first option, Ethereum and Cardano use the second.

## Interlude: Turing-Completeness

- Programming languages can be **Turing-complete** or not.

## Interlude: Turing-Completeness

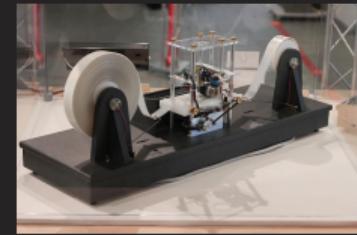
- Programming languages can be **Turing-complete** or not.
- A programming language is Turing-complete if every **Turing-machine** can be simulated in it.



**Figure:** Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

## Interlude: Turing-Completeness

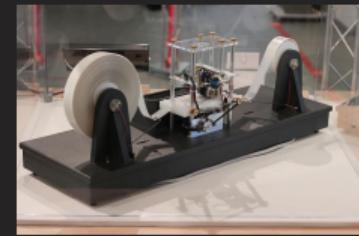
- Programming languages can be **Turing-complete** or not.
- A programming language is Turing-complete if every **Turing-machine** can be simulated in it.
- Intuitively, this means that such a language can compute everything that is computable.



**Figure:** Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

## Interlude: Turing-Completeness

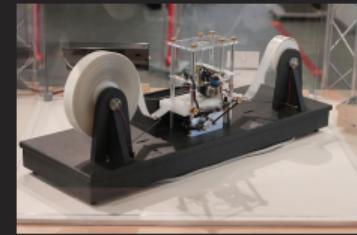
- Programming languages can be **Turing-complete** or not.
- A programming language is Turing-complete if every **Turing-machine** can be simulated in it.
- Intuitively, this means that such a language can compute everything that is computable.
- Since Turing-machines can get stuck in an infinite loop, the same is true for Turing-complete languages.



**Figure:** Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

## Interlude: Turing-Completeness

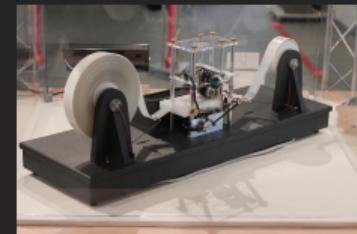
- Programming languages can be **Turing-complete** or not.
- A programming language is Turing-complete if every **Turing-machine** can be simulated in it.
- Intuitively, this means that such a language can compute everything that is computable.
- Since Turing-machines can get stuck in an infinite loop, the same is true for Turing-complete languages.
- All commonly used “higher” programming languages are Turing-complete: Python, Java, C, C++, Perl, JavaScript, Lisp, Haskell,...



**Figure:** Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

## Interlude: Turing-Completeness

- Programming languages can be **Turing-complete** or not.
- A programming language is Turing-complete if every **Turing-machine** can be simulated in it.
- Intuitively, this means that such a language can compute everything that is computable.
- Since Turing-machines can get stuck in an infinite loop, the same is true for Turing-complete languages.
- All commonly used “higher” programming languages are Turing-complete: Python, Java, C, C++, Perl, JavaScript, Lisp, Haskell,...
- Many more exotic systems (like  **$\lambda$ -calculus**) are Turing-complete as well.



**Figure:** Turing-machine, reconstructed by Mike Davey. Image by Rocky Acosta, Creative Commons Licence.

## Interlude: The Halting-Problem

- The famous **Halting-Problem** considers the question whether one can “see” if a program written in a Turing-complete language will get stuck in an infinite loop or **halt**.

## Interlude: The Halting-Problem

- The famous **Halting-Problem** considers the question whether one can “see” if a program written in a Turing-complete language will get stuck in an infinite loop or **halt**.
- To be more precise: Is there a program that will take an arbitrary program as input and then decide whether that program will halt?

## Interlude: The Halting-Problem

- The famous **Halting-Problem** considers the question whether one can “see” if a program written in a Turing-complete language will get stuck in an infinite loop or **halt**.
- To be more precise: Is there a program that will take an arbitrary program as input and then decide whether that program will halt?
- The answer is **no!** — Assume there was a Python-function `halt` solving the Halting-Problem. Then consider the following Python-function:

```
def paradox():
    if halt(paradox):
        while True:
            pass
```

If `halt` returns `True`, `paradox` gets stuck in an infinite loop, and if `halt` returns `False`, `paradox` will halt. Both are contradictions.

# Consequences for Smart Contracts

- Because of the negative answer to the Halting-problem, it is impossible to decide in advance whether an arbitrary program written in a Turing-complete language will eventually halt or run forever.

# Consequences for Smart Contracts

- Because of the negative answer to the Halting-problem, it is impossible to decide in advance whether an arbitrary program written in a Turing-complete language will eventually halt or run forever.
- So if one chooses a Turing-complete smart-contract language, it is impossible to guarantee in advance whether a script will stop. Nor is it possible to know for how long the script will run, even if it eventually halts.

# Consequences for Smart Contracts

- Because of the negative answer to the Halting-problem, it is impossible to decide in advance whether an arbitrary program written in a Turing-complete language will eventually halt or run forever.
- So if one chooses a Turing-complete smart-contract language, it is impossible to guarantee in advance whether a script will stop. Nor is it possible to know for how long the script will run, even if it eventually halts.
- As a consequence, one either has to decide against using a Turing-complete language or be prepared to interrupt a running script after some finite time has passed.

# Bitcoin Script

# Bitcoin Script

- Bitcoin uses a language which is **not** Turing-complete and which is called **Bitcoin Script**.

# Bitcoin Script

- Bitcoin uses a language which is **not** Turing-complete and which is called **Bitcoin Script**.
- Bitcoin Script does not allow *any* loops, so a program written in Bitcoin Script can never get stuck in an infinite loop.

# Bitcoin Script

- Bitcoin uses a language which is **not** Turing-complete and which is called **Bitcoin Script**.
- Bitcoin Script does not allow *any* loops, so a program written in Bitcoin Script can never get stuck in an infinite loop.
- In spite of its simplicity, Bitcoin Script is quite powerful and flexible and allows for a plethora of different kinds of transaction verification.

# Bitcoin Script

- Bitcoin uses a language which is **not** Turing-complete and which is called **Bitcoin Script**.
- Bitcoin Script does not allow *any* loops, so a program written in Bitcoin Script can never get stuck in an infinite loop.
- In spite of its simplicity, Bitcoin Script is quite powerful and flexible and allows for a plethora of different kinds of transaction verification.
- On the other hand, Bitcoin Script is too limited to allow for real smart contracts implementing complex financial transactions.

# Stacks

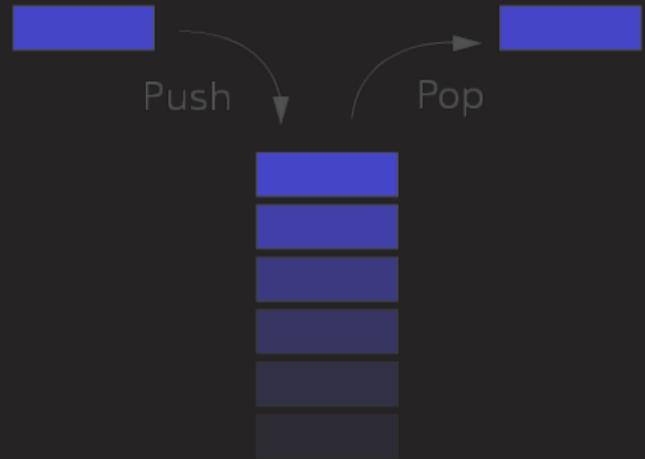
- Bitcoin Script is a so-called **Stack based** language.

# Stacks

- Bitcoin Script is a so-called **Stack based** language.
- **Forth** is a (relatively) popular higher programming language that is stack based as well. Other examples are the **Java Virtual Machine (JVM)** and Microsoft's **Common Language Runtime (CLR)**.

# Stacks

- Bitcoin Script is a so-called **Stack based** language.
- **Forth** is a (relatively) popular higher programming language that is stack based as well. Other examples are the **Java Virtual Machine (JVM)** and Microsoft's **Common Language Runtime (CLR)**.
- There are no variables in Bitcoin Script. Instead, data is put onto the **stack** and processed there.



# Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.

# Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
- To verify a transaction, each input is checked as follows:
  - The input-script is combined with the output-script (first the input-script, then the output-script).

# Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
- To verify a transaction, each input is checked as follows:
  - The input-script is combined with the output-script (first the input-script, then the output-script).
  - This combination is executed.

# Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
- To verify a transaction, each input is checked as follows:
  - The input-script is combined with the output-script (first the input-script, then the output-script).
  - This combination is executed.
  - Using the input is valid if the script does not report any errors and if in the end, there is a number on top of the stack which is not zero.

# Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
  - To verify a transaction, each input is checked as follows:
    - The input-script is combined with the output-script (first the input-script, then the output-script).
    - This combination is executed.
    - Using the input is valid if the script does not report any errors and if in the end, there is a number on top of the stack which is not zero.
  - The transaction is valid if all inputs are valid in this sense (and if all other conditions are satisfied, so the sum of all input-values is greater than the sum of all output-values etc.).

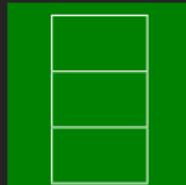
# Transaction-Verification with Bitcoin Script

- In Bitcoin, both transaction-inputs and transaction-outputs contain scripts.
- To verify a transaction, each input is checked as follows:
  - The input-script is combined with the output-script (first the input-script, then the output-script).
  - This combination is executed.
  - Using the input is valid if the script does not report any errors and if in the end, there is a number on top of the stack which is not zero.
- The transaction is valid if all inputs are valid in this sense (and if all other conditions are satisfied, so the sum of all input-values is greater than the sum of all output-values etc.).
- This script-mechanism **extends** the simple UTxO-model by allowing for more complex input-validation, going beyond digital signature verification.

## Example: Arithmetic

- The following program written in Bitcoin Script calculates  $(2 + 3) \cdot 4$ :
- 2 3 op\_add 4 op\_mul

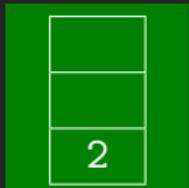
```
2  
3  
op_add  
4  
op_mul
```



## Example: Arithmetic

- The following program written in Bitcoin Script calculates  $(2 + 3) \cdot 4$ :
- 2 3 op\_add 4 op\_mul

- 2  
3  
op\_add  
4  
op\_mul



## Example: Arithmetic

- The following program written in Bitcoin Script calculates  $(2 + 3) \cdot 4$ :
- 2 3 op\_add 4 op\_mul

```
2  
• 3  
op_add  
4  
op_mul
```

3
2

## Example: Arithmetic

- The following program written in Bitcoin Script calculates  $(2 + 3) \cdot 4$ :
- 2 3 op\_add 4 op\_mul

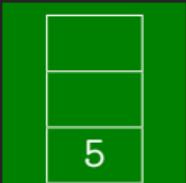
2

3

• op\_add

4

op\_mul



## Example: Arithmetic

- The following program written in Bitcoin Script calculates  $(2 + 3) \cdot 4$ :
- 2 3 op\_add 4 op\_mul

```
2  
3  
op_add  
• 4  
op_mul
```

4
5

## Example: Arithmetic

- The following program written in Bitcoin Script calculates  $(2 + 3) \cdot 4$ :
- 2 3 op\_add 4 op\_mul

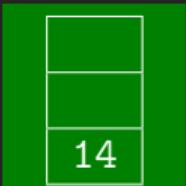
```
2
```

```
3
```

```
op_add
```

```
4
```

- op\_mul



## Exercises

- Compute  $(10 - 3) \cdot (4 + 7)$  using Bitcoin Script! (*Hint:* Use op\_sub!)
- Write a Bitcoin Script program which squares the number on top of the stack. (*Hint:* Use op\_dup!)
- Write a Bitcoin Script programm which computes  $x^2 + y^2$ , where  $x$  and  $y$  are the two top-most numbers on the stack. (*Hint:* Use op\_swap!)
- Write a Bitcoin Script Programm which computes  $x \cdot y$  if  $y < x$  and  $x + y$  if  $y \geq x$ , where  $x$  and  $y$  are the two top-most numbers on the stack ( $x$  on top,  $y$  below). (*Hint:* Use op\_2dup, op\_lessthan, op\_if, op\_else, and op\_endif!)
- You can find a list of all Bitcoin-Script commands on  
<https://en.bitcoin.it/wiki/Script>.
- There is a nice online simulator on  
<https://siminchen.github.io/bitcoinIDE/build/editor.html>.

# Pay to Public Key Hash in Bitcoin Script

- The vast majority of all Bitcoin transaction uses ordinary “hash of public key”-addresses.
- What do input- and output-scripts look like in this case?
  - <sig> <pubKey>
  - op\_dup op\_hash160 <pubKeyHash> op\_equalverify op\_checksig
  - The input-script puts the digital signature and the public key onto the stack. The output-script checks whether the hash of this public key has the right value and whether the signature is correct.

# Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01  
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceef8dc
```

- Output-Script:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

304402200cc8b0471a38ed...

02ce9f5972fe1473c9b694...

op\_dup

op\_hash160

1290b657a78e201967c22d...

op\_equalverify

op\_checksig



# Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01  
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceef8dc
```

- Output-Script:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

- 304402200cc8b0471a38ed...  
02ce9f5972fe1473c9b694...  
op\_dup  
op\_hash160  
1290b657a78e201967c22d...  
op\_equalverify  
op\_checksig

304402200cc8b0471a38ed...

# Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01  
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceef8dc
```

- Output-Script:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

304402200cc8b0471a38ed...

• 02ce9f5972fe1473c9b694...

op\_dup

op\_hash160

1290b657a78e201967c22d...

op\_equalverify

op\_checksig

02ce9f5972fe1473c9b694...

304402200cc8b0471a38ed...

# Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01  
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceef8dc
```

- Output-Script:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

304402200cc8b0471a38ed...

02ce9f5972fe1473c9b694...

- op\_dup
- op\_hash160
- 1290b657a78e201967c22d...
- op\_equalverify
- op\_checksig

02ce9f5972fe1473c9b694...

02ce9f5972fe1473c9b694...

304402200cc8b0471a38ed...

# Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01  
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceef8dc
```

- Output-Script:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

304402200cc8b0471a38ed...

02ce9f5972fe1473c9b694...

op\_dup

- op\_hash160

1290b657a78e201967c22d...

op\_equalverify

op\_checksig

1290b657a78e201967c22d...

02ce9f5972fe1473c9b694...

304402200cc8b0471a38ed...

# Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01  
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceef8dc
```

- Output-Script:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

304402200cc8b0471a38ed...

02ce9f5972fe1473c9b694...

op\_dup

op\_hash160

- 1290b657a78e201967c22d...

op\_equalverify

op\_checksig

1290b657a78e201967c22d...

1290b657a78e201967c22d...

02ce9f5972fe1473c9b694...

304402200cc8b0471a38ed...

# Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01  
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceef8dc
```

- Output-Script:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

304402200cc8b0471a38ed...

02ce9f5972fe1473c9b694...

op\_dup

op\_hash160

1290b657a78e201967c22d...

- op\_equalverify

op\_checksig

02ce9f5972fe1473c9b694...

304402200cc8b0471a38ed...

# Pay to Public Key Hash — Example

- Input-Script:

```
304402200cc8b0471a38edad2ff9f9799521b7d948054817793c980eaf3a6637ddfb939702201c1a801461d4c3cf4de4e7336454dba0dd70b89d71f221e991cb6a79df1a860d01  
02ce9f5972fe1473c9b6948949f676bbf7893a03c5b4420826711ef518ceef8dc
```

- Output-Script:

```
op_dup op_hash160 1290b657a78e201967c22d8022b348bd5e23ce17 op_equalverify op_checksig
```

304402200cc8b0471a38ed...

02ce9f5972fe1473c9b694...

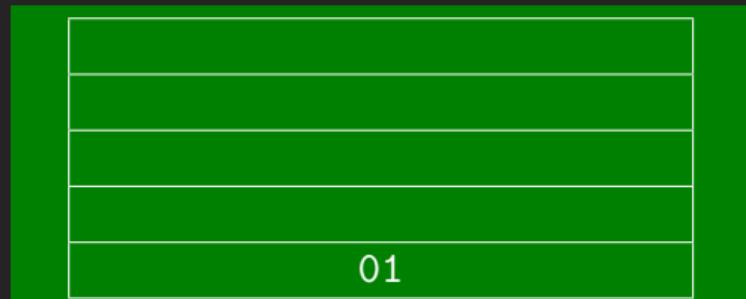
op\_dup

op\_hash160

1290b657a78e201967c22d...

op\_equalverify

- op\_checksig



# Other kinds of Bitcoin Scripts — Multisig

- Another common Bitcoin transaction type is **multisig**.

# Other kinds of Bitcoin Scripts — Multisig

- Another common Bitcoin transaction type is **multisig**.
- If an output uses multisig, it can only be spent if several parties provide their signatures.

## Other kinds of Bitcoin Scripts — Multisig

- Another common Bitcoin transaction type is **multisig**.
- If an output uses multisig, it can only be spent if several parties provide their signatures.
- Bitcoin Script supports this via `op_checkmultisig`.

# Other kinds of Bitcoin Scripts — Unspendable Outputs

- The script-command `op_return` renders an output **unspendable**.

# Other kinds of Bitcoin Scripts — Unspendable Outputs

- The script-command `op_return` renders an output **unspendable**.
- All commands following `op_return` are ignored.

# Other kinds of Bitcoin Scripts — Unspendable Outputs

- The script-command `op_return` renders an output **unspendable**.
- All commands following `op_return` are ignored.
- One possible application of this is to for example generate an output with value zero, then use a script which starts with `op_return` and contains arbitrary data afterwards.

# Other kinds of Bitcoin Scripts — Riddles

- The output of transaction

a4bfa8ab6435ae5f25dae9d89e4eb67dfa94283ca751f393c1ddc5a837bbc31b  
contained the following script:

```
op_hash256
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
op_equal
```

# Other kinds of Bitcoin Scripts — Riddles

- The output of transaction

a4bfa8ab6435ae5f25dae9d89e4eb67dfa94283ca751f393c1ddc5a837bbc31b

contained the following script:

```
op_hash256
```

```
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
```

```
op_equal
```

- To spend that output, one had to find a number with the given hash.

# Other kinds of Bitcoin Scripts — Riddles

- The output of transaction

a4bfa8ab6435ae5f25dae9d89e4eb67dfa94283ca751f393c1ddc5a837bbc31b

contained the following script:

```
op_hash256
```

```
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
```

```
op_equal
```

- To spend that output, one had to find a number with the given hash.
- This riddle was eventually solved: The given hash turned out to be the hash of the genesis-block-header.

## Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- In the year 2013, Peter Todd created scripts whose outputs could be spent by anybody who found a **hash collision** for SHA-1:

```
op_2dup op_equal op_not op_verify op_sha1 op_swap op_sha1 op_equal
```

## Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- In the year 2013, Peter Todd created scripts whose outputs could be spent by anybody who found a **hash collision** for SHA-1:  
`op_2dup op_equal op_not op_verify op_sha1 op_swap op_sha1 op_equal`
- You can send Bitcoin to script-addresses like these to reward people for finding a hash collisions.

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- In the year 2013, Peter Todd created scripts whose outputs could be spent by anybody who found a **hash collision** for SHA-1:  
`op_2dup op_equal op_not op_verify op_sha1 op_swap op_sha1 op_equal`
- You can send Bitcoin to script-addresses like these to reward people for finding a hash collisions.
- As long as the reward was high enough and nobody claimed it, one could be relatively certain that SHA-1 was still safe and that nobody had found a collision.

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- In the year 2013, Peter Todd created scripts whose outputs could be spent by anybody who found a **hash collision** for SHA-1:  
`op_2dup op_equal op_not op_verify op_sha1 op_swap op_sha1 op_equal`
- You can send Bitcoin to script-addresses like these to reward people for finding a hash collisions.
- As long as the reward was high enough and nobody claimed it, one could be relatively certain that SHA-1 was still safe and that nobody had found a collision.
- In February 2017, somebody claimed the reward of 2.48 ₿.

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

<Datum-1>

<Datum-2>

op\_2dup

op\_equal

op\_not

op\_verify

op\_sha1

op\_swap

op\_sha1

op\_equal

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- <Datum-1>  
<Datum-2>  
op\_2dup  
op\_equal  
op\_not  
op\_verify  
op\_sha1  
op\_swap  
op\_sha1  
op\_equal

<Datum-1>

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- <Datum-1>
- <Datum-2>
  - op\_2dup
  - op\_equal
  - op\_not
  - op\_verify
  - op\_sha1
  - op\_swap
  - op\_sha1
  - op\_equal

<Datum-2>
<Datum-1>

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

- <Datum-1>
- <Datum-2>
- op\_2dup
- op\_equal
- op\_not
- op\_verify
- op\_sha1
- op\_swap
- op\_sha1
- op\_equal

<Datum-2>
<Datum-1>
<Datum-2>
<Datum-1>

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

```
<Datum-1>
<Datum-2>
op_2dup
• op_equal
  op_not
  op_verify
  op_sha1
  op_swap
  op_sha1
  op_equal
```

00
<Datum-2>
<Datum-1>

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

```
<Datum-1>
<Datum-2>
op_2dup
op_equal
• op_not
op_verify
op_sha1
op_swap
op_sha1
op_equal
```

01
<Datum-2>
<Datum-1>

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

```
<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
• op_verify
op_sha1
op_swap
op_sha1
op_equal
```

<Datum-2>
<Datum-1>

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

```
<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
op_verify
• op_sha1
op_swap
op_sha1
op_equal
```

<Hash>
<Datum-1>

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

```
<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
op_verify
op_sha1
• op_swap
op_sha1
op_equal
```

<Datum-1>
<Hash>

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

<Datum-1>

<Datum-2>

op\_2dup

op\_equal

op\_not

op\_verify

op\_sha1

op\_swap

- op\_sha1

op\_equal

<Hash>

<Hash>

# Other kinds of Bitcoin Script — Incentivize Finding Hash Collisions

```
<Datum-1>
<Datum-2>
op_2dup
op_equal
op_not
op_verify
op_sha1
op_swap
op_sha1
• op_equal
```

01

# Other kinds of Bitcoin Script — Time Locks

- Bitcoin Script supports **time locks** via `op_checklocktimeverify` and `op_checksequenceverify`.

# Other kinds of Bitcoin Script — Time Locks

- Bitcoin Script supports **time locks** via `op_checklocktimeverify` and `op_checksequenceverify`.
- `op_checklocktimeverify` only allows spending an output when a certain **point in time** (measured in block height) has been reached.

Other kinds of Bitcoin Script — Time Locks

- Bitcoin Script supports **time locks** via `op_checklocktimeverify` and `op_checksequenceverify`.
  - `op_checklocktimeverify` only allows spending an output when a certain **point in time** (measured in block height) has been reached.
  - `op_checksequenceverify`, on the other hand, uses **relative time**: Spending the output is only possible after the transaction has reached a given depth in the blockchain.

# Other kinds of Bitcoin Script — Time Locks

- Bitcoin Script supports **time locks** via `op_checklocktimeverify` and `op_checksequenceverify`.
- `op_checklocktimeverify` only allows spending an output when a certain **point in time** (measured in block height) has been reached.
- `op_checksequenceverify`, on the other hand, uses **relative** time: Spending the output is only possible after the transaction has reached a given depth in the blockchain.
- Both types of time locks are for example used in [Bitcoin Lightning](#).

# Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).

# Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.

## Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
  - They are worried that they might lose their money if something happens to one of them.
  - Therefore they decide to ask their lawyer Charlie for help.

## Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.
- Therefore they decide to ask their lawyer Charlie for help.
- Were they to use ordinary 2-of-3-multisig, Alice or Bob could conspire with Charlie to steal the money.

# Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.
- Therefore they decide to ask their lawyer Charlie for help.
- Were they to use ordinary 2-of-3-multisig, Alice or Bob could conspire with Charlie to steal the money.
- Instead they can use the following script:

```
op_if <in drei Monaten> op_checklocktimeverify op_drop  
<PubKey-Charlie> op_checksigverify 1 op_else 2 op_endif  
<PubKey-Alice> <PubKey-Bob> 2 op_checkmultisig
```

# Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.
- Therefore they decide to ask their lawyer Charlie for help.
- Were they to use ordinary 2-of-3-multisig, Alice or Bob could conspire with Charlie to steal the money.
- Instead they can use the following script:

```
op_if <in drei Monaten> op_checklocktimeverify op_drop  
<PubKey-Charlie> op_checksigverify 1 op_else 2 op_endif  
<PubKey-Alice> <PubKey-Bob> 2 op_checkmultisig
```

- Alice and Bob can access their money at any time using this script:

```
0 <Sig-Alice> <Sig-Bob> 0
```

# Time Lock Example

- Alice and Bob are business partners and lock their money via 2-of-2-multisig (i.e. both have to sign).
- They are worried that they might lose their money if something happens to one of them.
- Therefore they decide to ask their lawyer Charlie for help.
- Were they to use ordinary 2-of-3-multisig, Alice or Bob could conspire with Charlie to steal the money.
- Instead they can use the following script:

```
op_if <in drei Monaten> op_checklocktimeverify op_drop  
<PubKey-Charlie> op_checksigverify 1 op_else 2 op_endif  
<PubKey-Alice> <PubKey-Bob> 2 op_checkmultisig
```

- Alice and Bob can access their money at any time using this script:  
0 <Sig-Alice> <Sig-Bob> 0
- After three months, Charlie and either Alice or Bob can use the following script instead:  
0 <Sig-Alice/Bob> <Sig-Charlie> 1

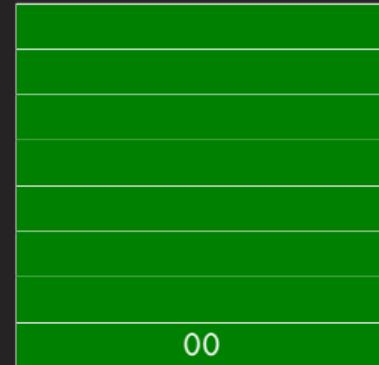
# Time Lock Example — First Case

```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



# Time Lock Example — First Case

- 00  
<Sig-Alice>  
<Sig-Bob>  
00  
op\_if  
<in drei Monaten>  
op\_checklocktimeverify  
op\_drop  
<PubKey-Charlie>  
op\_checksigverify  
01  
op\_else  
02  
op\_endif  
<PubKey-Alice>  
<PubKey-Bob>  
02  
op\_checkmultisig



# Time Lock Example — First Case

```
00
• <Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<Sig-Alice>
00

# Time Lock Example — First Case

```
00
<Sig-Alice>
• <Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<Sig-Bob>
<Sig-Alice>
00

# Time Lock Example — First Case

```
00
<Sig-Alice>
<Sig-Bob>
• 00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

00
<Sig-Bob>
<Sig-Alice>
00

# Time Lock Example — First Case

```
00
<Sig-Alice>
<Sig-Bob>
00
• op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<Sig-Bob>
<Sig-Alice>
00

# Time Lock Example — First Case

```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
• 02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

02
<Sig-Bob>
<Sig-Alice>
00

# Time Lock Example — First Case

```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
• <PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<PubKey-Alice>
02
<Sig-Bob>
<Sig-Alice>
00

# Time Lock Example — First Case

```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
• <PubKey-Bob>
02
op_checkmultisig
```

<PubKey-Bob>
<PubKey-Alice>
02
<Sig-Bob>
<Sig-Alice>
00

# Time Lock Example — First Case

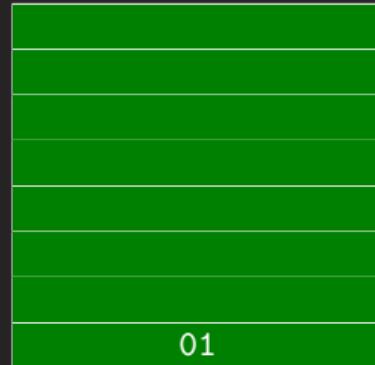
```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

02
<PubKey-Bob>
<PubKey-Alice>
02
<Sig-Bob>
<Sig-Alice>
00

- 02

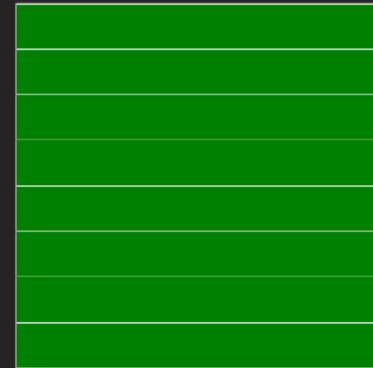
# Time Lock Example — First Case

```
00
<Sig-Alice>
<Sig-Bob>
00
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
• op_checkmultisig
```



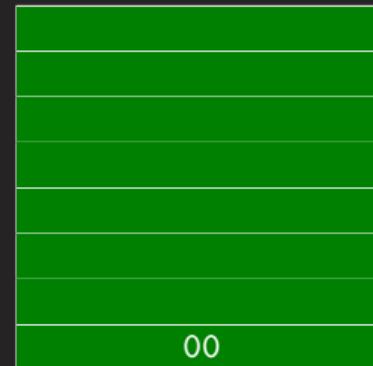
## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```



## Time Lock Example — Second Case

- 00  
<Sig-Alice/Bob>  
<Sig-Charlie>  
01  
op\_if  
<in drei Monaten>  
op\_checklocktimeverify  
op\_drop  
<PubKey-Charlie>  
op\_checksigverify  
01  
op\_else  
02  
op\_endif  
<PubKey-Alice>  
<PubKey-Bob>  
02  
op\_checkmultisig



## Time Lock Example — Second Case

```
00
• <Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
• <Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<Sig-Charlie>
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
• 01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

01
<Sig-Charlie>
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
• op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<Sig-Charlie>
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
• <in drei Monaten>
  op_checklocktimeverify
  op_drop
  <PubKey-Charlie>
  op_checksigverify
  01
  op_else
  02
  op_endif
  <PubKey-Alice>
  <PubKey-Bob>
  02
  op_checkmultisig
```

<in drei Monaten>
<Sig-Charlie>
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
• op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<in drei Monaten>
<Sig-Charlie>
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
• op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<Sig-Charlie>
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
• <PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<PubKey-Charlie>
<Sig-Charlie>
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
• op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
• 01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

01
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
• <PubKey-Alice>
<PubKey-Bob>
02
op_checkmultisig
```

<PubKey-Alice>
01
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00  
<Sig-Alice/Bob>  
<Sig-Charlie>  
01  
op_if  
<in drei Monaten>  
op_checklocktimeverify  
op_drop  
<PubKey-Charlie>  
op_checksigverify  
01  
op_else  
02  
op_endif  
<PubKey-Alice>  
• <PubKey-Bob>  
02  
op_checkmultisig
```

<PubKey-Bob>
<PubKey-Alice>
01
<Sig-Alice/Bob>
00

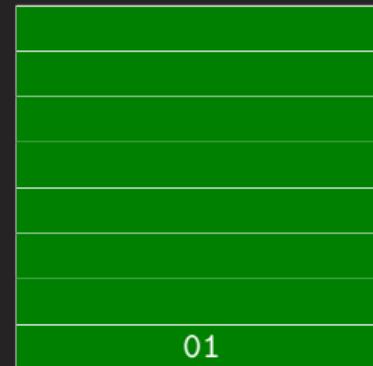
## Time Lock Example — Second Case

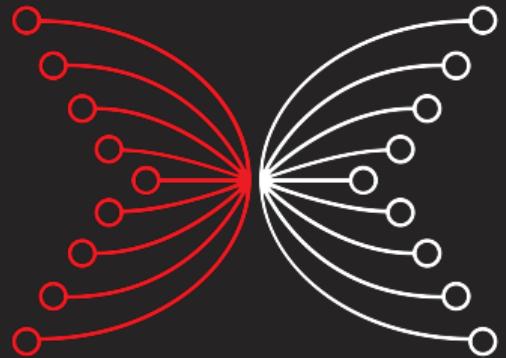
```
00  
<Sig-Alice/Bob>  
<Sig-Charlie>  
01  
op_if  
<in drei Monaten>  
op_checklocktimeverify  
op_drop  
<PubKey-Charlie>  
op_checksigverify  
01  
op_else  
02  
op_endif  
<PubKey-Alice>  
<PubKey-Bob>  
02  
op_checkmultisig
```

02
<PubKey-Bob>
<PubKey-Alice>
01
<Sig-Alice/Bob>
00

## Time Lock Example — Second Case

```
00
<Sig-Alice/Bob>
<Sig-Charlie>
01
op_if
<in drei Monaten>
op_checklocktimeverify
op_drop
<PubKey-Charlie>
op_checksigverify
01
op_else
02
op_endif
<PubKey-Alice>
<PubKey-Bob>
02
• op_checkmultisig
```





INPUT | OUTPUT

# Smart Contracts

## Lightning

Lars Brünjes



January 9 2020

# Introduction

# Problem

- Problem

- In current cryptocurrencies, each transaction has to be stored on each node.

# Problem

- Problem

- In current cryptocurrencies, each transaction has to be stored on each node.
- The consensus algorithm restricts the maximum rate at which new blocks can be created, and blocks are furthermore subject to size restrictions.

# Problem

- Problem

- In current cryptocurrencies, each transaction has to be stored on each node.
- The consensus algorithm restricts the maximum rate at which new blocks can be created, and blocks are furthermore subject to size restrictions.
- This leads to rather modest transaction rates in comparison to traditional, centralized payment networks like Visa.

# Problem

- Problem
  - In current cryptocurrencies, each transaction has to be stored on each node.
  - The consensus algorithm restricts the maximum rate at which new blocks can be created, and blocks are furthermore subject to size restrictions.
  - This leads to rather modest transaction rates in comparison to traditional, centralized payment networks like Visa.
- Lightning
  - The [Lightning-Network](#) is one way to solve this problem.

# Problem

- Problem
  - In current cryptocurrencies, each transaction has to be stored on each node.
  - The consensus algorithm restricts the maximum rate at which new blocks can be created, and blocks are furthermore subject to size restrictions.
  - This leads to rather modest transaction rates in comparison to traditional, centralized payment networks like Visa.
- Lightning
  - The [Lightning-Network](#) is one way to solve this problem.
  - It is initially planned for Bitcoin, but its idea is generic enough to apply to many other cryptocurrencies (including Cardano).

# Problem

- Problem
  - In current cryptocurrencies, each transaction has to be stored on each node.
  - The consensus algorithm restricts the maximum rate at which new blocks can be created, and blocks are furthermore subject to size restrictions.
  - This leads to rather modest transaction rates in comparison to traditional, centralized payment networks like Visa.
- Lightning
  - The [Lightning-Network](#) is one way to solve this problem.
  - It is initially planned for Bitcoin, but its idea is generic enough to apply to many other cryptocurrencies (including Cardano).
  - The basic idea is to offload work from nodes by creating parallel [side channels](#), which can process the bulk of all transactions.

# Idea

- The basic idea is for two parties who want to exchange funds to create a **payment channel** between themselves, which is independent of the Bitcoin network.

# Idea

- The basic idea is for two parties who want to exchange funds to create a **payment channel** between themselves, which is independent of the Bitcoin network.
- This channel can then process arbitrarily many transactions in a very fast and cheap manner.

# Idea

- The basic idea is for two parties who want to exchange funds to create a **payment channel** between themselves, which is independent of the Bitcoin network.
- This channel can then process arbitrarily many transactions in a very fast and cheap manner.
- In the normal case, when both parties play by the rules, only *two* transactions will ever have to be sent to the Bitcoin network, one to open the channel, one to close it again when it is no longer needed.

# Idea

- The basic idea is for two parties who want to exchange funds to create a **payment channel** between themselves, which is independent of the Bitcoin network.
- This channel can then process arbitrarily many transactions in a very fast and cheap manner.
- In the normal case, when both parties play by the rules, only *two* transactions will ever have to be sent to the Bitcoin network, one to open the channel, one to close it again when it is no longer needed.
- In spite of this, all payments using the channel are secure and are guaranteed by the blockchain.

## Idea

- The basic idea is for two parties who want to exchange funds to create a **payment channel** between themselves, which is independent of the Bitcoin network.
  - This channel can then process arbitrarily many transactions in a very fast and cheap manner.
  - In the normal case, when both parties play by the rules, only *two* transactions will ever have to be sent to the Bitcoin network, one to open the channel, one to close it again when it is no longer needed.
  - In spite of this, all payments using the channel are secure and are guaranteed by the blockchain.
  - After we will have understood how such a channel between two parties works, we will see how that system can be extended to allow payments between parties who do not possess a direct channel between each other.

# Direct Payment Channels

# Opening a Payment Channel

- Let us assume that Alice and Bob want to open a payment channel between each other, because they frequently exchange Bitcoin.

# Opening a Payment Channel

- Let us assume that Alice and Bob want to open a payment channel between each other, because they frequently exchange Bitcoin.
- The open the channel by each depositing a certain amount of Bitcoin on the blockchain (for example 5 ₿ each). (If Alice expects that in future, she will send more money to Bob than she will receive from him, she can also deposit more than he does.)
- They proceed as follows:

# Opening a Payment Channel

- Let us assume that Alice and Bob want to open a payment channel between each other, because they frequently exchange Bitcoin.
- The open the channel by each depositing a certain amount of Bitcoin on the blockchain (for example 5 ₿ each). (If Alice expects that in future, she will send more money to Bob than she will receive from him, she can also deposit more than he does.)
- They proceed as follows:
  - They create a 2-of-2-multisig-address and prepare a transaction, which will send their deposit to this address, but they do not yet sign that transaction.

# Opening a Payment Channel

- Let us assume that Alice and Bob want to open a payment channel between each other, because they frequently exchange Bitcoin.
- The open the channel by each depositing a certain amount of Bitcoin on the blockchain (for example 5 ₿ each). (If Alice expects that in future, she will send more money to Bob than she will receive from him, she can also deposit more than he does.)
- They proceed as follows:
  - They create a 2-of-2-multisig-address and prepare a transaction, which will send their deposit to this address, but they do not yet sign that transaction.
  - Each picks a random number, a **secret**, and sends its **hash** to the other.

# Opening a Payment Channel

- Let us assume that Alice and Bob want to open a payment channel between each other, because they frequently exchange Bitcoin.
- The open the channel by each depositing a certain amount of Bitcoin on the blockchain (for example 5 ₿ each). (If Alice expects that in future, she will send more money to Bob than she will receive from him, she can also deposit more than he does.)
- They proceed as follows:
  - They create a 2-of-2-multisig-address and prepare a transaction, which will send their deposit to this address, but they do not yet sign that transaction.
  - Each picks a random number, a **secret**, and sends its **hash** to the other.
  - Alice creates and signs a new transaction, which has the multisig-address as input and two outputs, her deposit to herself, Bob's deposit to a new multisig-address, which can **either** be unlocked by Bob after 1000 blocks **or** immediately by Alice, **if** she knows Bob's secret.
  - Bob creates and signs a new transaction, which has the multisig-address as input and two outputs, his deposit to himself, Alice's deposit to a new multisig-address, which can **either** be unlocked by Alice after 1000 blocks **or** immediately by Bob, **if** he knows Alice's secret.

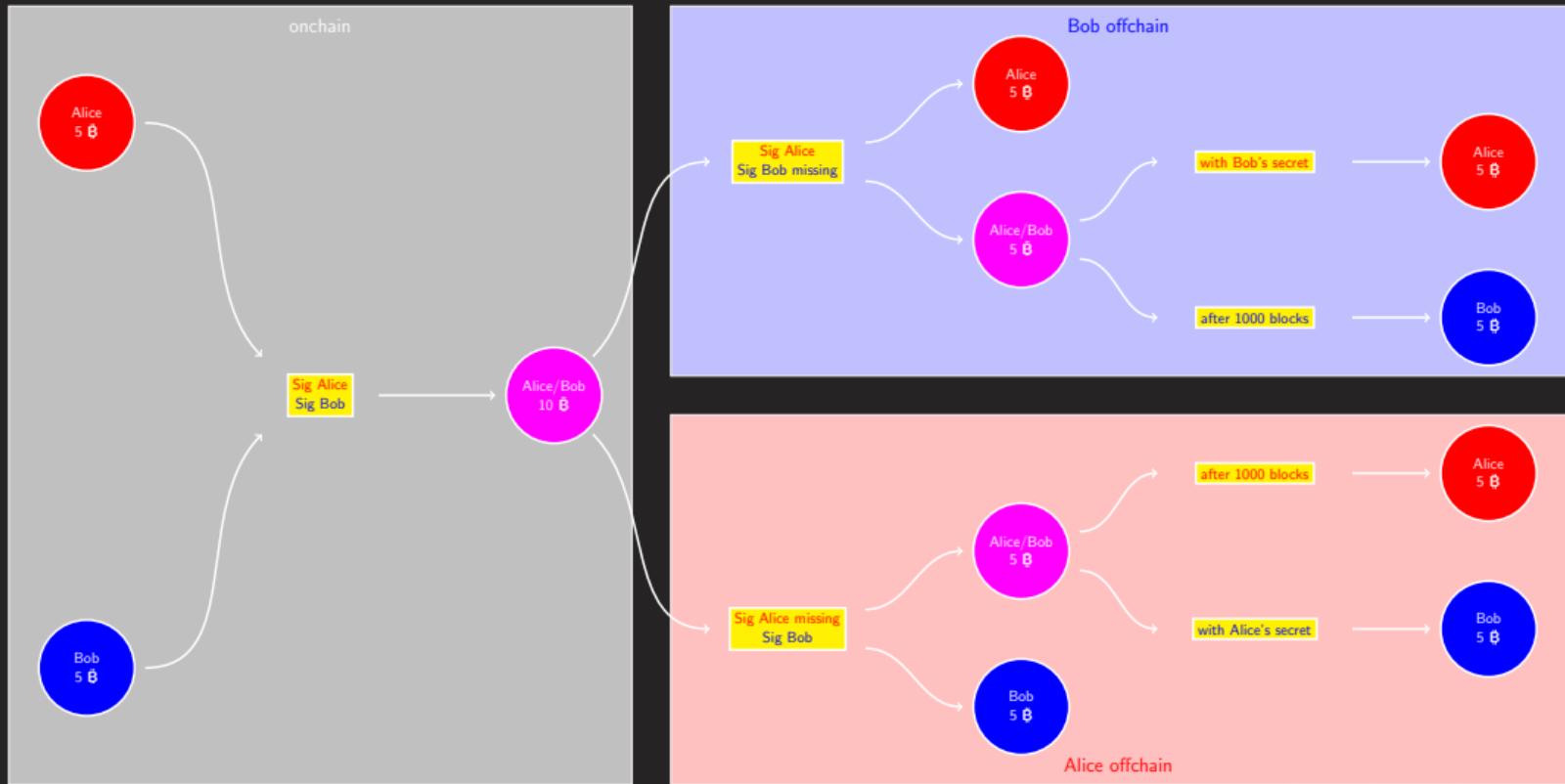
# Opening a Payment Channel

- Let us assume that Alice and Bob want to open a payment channel between each other, because they frequently exchange Bitcoin.
- They open the channel by each depositing a certain amount of Bitcoin on the blockchain (for example 5 ₿ each). (If Alice expects that in future, she will send more money to Bob than she will receive from him, she can also deposit more than he does.)
- They proceed as follows:
  - ① They create a 2-of-2-multisig-address and prepare a transaction, which will send their deposit to this address, but they do not yet sign that transaction.
  - ② Each picks a random number, a **secret**, and sends its **hash** to the other.
  - ③ Alice creates and signs a new transaction, which has the multisig-address as input and two outputs, her deposit to herself, Bob's deposit to a new multisig-address, which can **either** be unlocked by Bob after 1000 blocks **or** immediately by Alice, **if** she knows Bob's secret.
  - ④ Bob creates and signs a new transaction, which has the multisig-address as input and two outputs, his deposit to himself, Alice's deposit to a new multisig-address, which can **either** be unlocked by Alice after 1000 blocks **or** immediately by Bob, **if** he knows Alice's secret.
  - ⑤ Alice and Bob exchange the two new transactions from steps 3 Und 4.

# Opening a Payment Channel

- Let us assume that Alice and Bob want to open a payment channel between each other, because they frequently exchange Bitcoin.
- The open the channel by each depositing a certain amount of Bitcoin on the blockchain (for example 5 ₿ each). (If Alice expects that in future, she will send more money to Bob than she will receive from him, she can also deposit more than he does.)
- They proceed as follows:
  - They create a 2-of-2-multisig-address and prepare a transaction, which will send their deposit to this address, but they do not yet sign that transaction.
  - Each picks a random number, a **secret**, and sends its **hash** to the other.
  - Alice creates and signs a new transaction, which has the multisig-address as input and two outputs, her deposit to herself, Bob's deposit to a new multisig-address, which can **either** be unlocked by Bob after 1000 blocks **or** immediately by Alice, **if** she knows Bob's secret.
  - Bob creates and signs a new transaction, which has the multisig-address as input and two outputs, his deposit to himself, Alice's deposit to a new multisig-address, which can **either** be unlocked by Alice after 1000 blocks **or** immediately by Bob, **if** he knows Alice's secret.
  - Alice and Bob exchange the two new transactions from steps 3 Und 4.
  - Finally, Alice and Bob sign the transaction from step-1 and send it to the blockchain.

# Illustration



# Explanation

- This complicated arrangement guarantees that Alice and Bob will be able to retrieve their deposits, no matter what.

# Explanation

- This complicated arrangement guarantees that Alice and Bob will be able to retrieve their deposits, no matter what.
- Under normal circumstances — when both of them agree — they can send a simple 2-of-2-multisig-transaction to the blockchain, which is signed by both of them and returns their deposits to them.

# Explanation

- This complicated arrangement guarantees that Alice and Bob will be able to retrieve their deposits, no matter what.
- Under normal circumstances — when both of them agree — they can send a simple 2-of-2-multisig-transaction to the blockchain, which is signed by both of them and returns their deposits to them.
- If Alice wants to retrieve her deposit without Bob's help, she signs the transaction she received from Bob and sends it to the blockchain.
  - Bob gets his deposit immediately.
  - Alice must wait for 1000 blocks until she gets her deposit.
  - If Bob manages to learn Alice's secret in the meantime, he can get his hands on Alice's deposit before the 1000 blocks are over. (We will see later what this is good for.)

# Explanation

- This complicated arrangement guarantees that Alice and Bob will be able to retrieve their deposits, no matter what.
- Under normal circumstances — when both of them agree — they can send a simple 2-of-2-multisig-transaction to the blockchain, which is signed by both of them and returns their deposits to them.
- If Bob wants to retrieve his deposit without Alice's help, he signs the transaction he received from Alice and sends it to the blockchain.
  - Alice gets her deposit immediately.
  - Bob must wait for 1000 blocks until he gets his deposit.
  - If Alice manages to learn Bob's secret in the meantime, she can get her hands on Bob's deposit before the 1000 blocks are over. (We will see later what this is good for.)

# Using a Payment Channel

- After all this effort, Alice and Bob want to use their shining new payment channel to send Bitcoin!

# Using a Payment Channel

- After all this effort, Alice and Bob want to use their shining new payment channel to send Bitcoin!
- If Alice wants to send 1  $\text{฿}$  to Bob, they proceed as follows:

# Using a Payment Channel

- After all this effort, Alice and Bob want to use their shining new payment channel to send Bitcoin!
- If Alice wants to send 1  $\text{฿}$  to Bob, they proceed as follows:
  - Each chooses a **new** secret and sends its **hash** to the other.

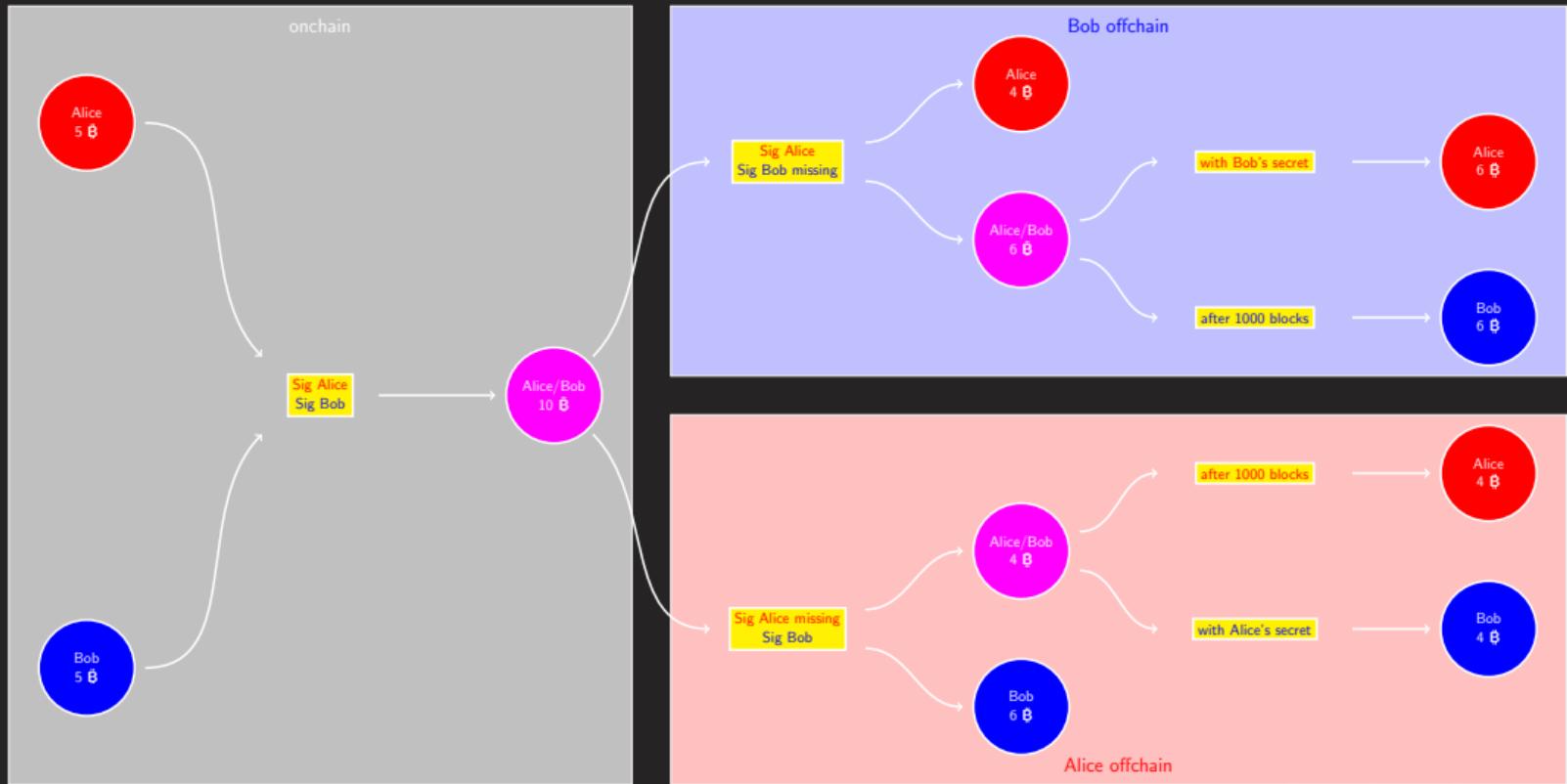
# Using a Payment Channel

- After all this effort, Alice and Bob want to use their shining new payment channel to send Bitcoin!
- If Alice wants to send 1 ₿ to Bob, they proceed as follows:
  - Each chooses a **new** secret and sends its **hash** to the other.
  - Alice creates, signs and sends Bob a new transaction with the multisig-address as input and two outputs — 4 ₿ to herself, 6 ₿ to a new multisig-address, which can be unlocked **either** after 1000 blocks by Bob **or** immediately by Alice, provided she has learned Bob's **new** secret.
  - Bob creates, signs and sends Alice a new transaction with the multisig-address as input and two outputs — 6 ₿ to himself, 4 ₿ to a new multisig-address, which can be unlocked **either** after 1000 blocks by Alice **or** immediately by Bob, provided he has learned Alice's **new** secret.

# Using a Payment Channel

- After all this effort, Alice and Bob want to use their shining new payment channel to send Bitcoin!
- If Alice wants to send 1 ₿ to Bob, they proceed as follows:
  - Each chooses a **new** secret and sends its **hash** to the other.
  - Alice creates, signs and sends Bob a new transaction with the multisig-address as input and two outputs — 4 ₿ to herself, 6 ₿ to a new multisig-address, which can be unlocked **either** after 1000 blocks by Bob **or** immediately by Alice, provided she has learned Bob's **new** secret.
  - Bob creates, signs and sends Alice a new transaction with the multisig-address as input and two outputs — 6 ₿ to himself, 4 ₿ to a new multisig-address, which can be unlocked **either** after 1000 blocks by Alice **or** immediately by Bob, provided he has learned Alice's **new** secret.
  - Alice and Bob exchange their **old** secrets.

# Illustration



# Explanation

- Under normal circumstance, if both agree, they can later send a common 2-of-2-multisig-transaction to the blockchain which is signed by both and gives 4 ₿ to Alice and 6 ₿ to Bob.

# Explanation

- Under normal circumstance, if both agree, they can later send a common 2-of-2-multisig-transaction to the blockchain which is signed by both and gives 4 ₿ to Alice and 6 ₿ to Bob.
- As before, Alice can retrieve her 4 ₿ without Bob's help by sending the new transaction she received from Bob to the blockchain.
- If she instead tries to use Bob's old transaction, she will have to wait for 1000 blocks for her 5 ₿. But Bob knows her old secret now and can get all the money for himself in this case. This means that Bob's old transaction is now worthless for Alice.

# Explanation

- Under normal circumstance, if both agree, they can later send a common 2-of-2-multisig-transaction to the blockchain which is signed by both and gives 4 ₿ to Alice and 6 ₿ to Bob.
- As before, Bob can retrieve his 6 ₿ without Alice's help by sending the new transaction he received from Alice to the blockchain.
- He has no interest in using Alice's old transaction, because that one only gives him 5 ₿ instead of 6 ₿. In addition to that, Alice could get all the money in this case, because she knows Bob's old secret now. Alice's old transaction is therefore worthless for Bob.

# Closing the Payment Channel

- In this manner Alice and Bob can send arbitrarily many transactions to each other (as long as the balance does not exceed the original deposit made to the blockchain).

# Closing the Payment Channel

- In this manner Alice and Bob can send arbitrarily many transactions to each other (as long as the balance does not exceed the original deposit made to the blockchain).
- As long as both play by the rules, **no further transaction is visible on the blockchain**. Communication between Alice and Bob is parallel to the blockchain, is “lightning fast” and (as good as) for free. Payments are therefore much faster and cheaper than normal Bitcoin transactions.

# Closing the Payment Channel

- In this manner Alice and Bob can send arbitrarily many transactions to each other (as long as the balance does not exceed the original deposit made to the blockchain).
- As long as both play by the rules, **no further transaction is visible on the blockchain**. Communication between Alice and Bob is parallel to the blockchain, is “lightning fast” and (as good as) for free. Payments are therefore much faster and cheaper than normal Bitcoin transactions.
- At any point in time, Alice’s and Bob’s money is safe. They can send the other’s most current transaction to the blockchain at any time to retrieve their money (after 1000 blocks).

# Closing the Payment Channel

- In this manner Alice and Bob can send arbitrarily many transactions to each other (as long as the balance does not exceed the original deposit made to the blockchain).
- As long as both play by the rules, **no further transaction is visible on the blockchain**. Communication between Alice and Bob is parallel to the blockchain, is “lightning fast” and (as good as) for free. Payments are therefore much faster and cheaper than normal Bitcoin transactions.
- At any point in time, Alice’s and Bob’s money is safe. They can send the other’s most current transaction to the blockchain at any time to retrieve their money (after 1000 blocks).
- If both agree to close the channel, they can do so using a common 2-of-2-multisig transaction. This means that under normal circumstances, only two “real” Bitcoin transactions are needed, one to open the channel and one to close it in the end.

# Indirect Payments

# Payments Without a Direct Channel

- Lightning also allows payments between parties who don't own a direct payment channel between them.

# Payments Without a Direct Channel

- Lightning also allows payments between parties who don't own a direct payment channel between them.
  - Let us assume that Alice wants to send 1  $\text{฿}$  to Charlie without first establishing a payment channel with him. Let us further assume that Bob and Charlie possess a payment channel between each other.

# Payments Without a Direct Channel

- Lightning also allows payments between parties who don't own a direct payment channel between them.
  - Let us assume that Alice wants to send 1  $\text{฿}$  to Charlie without first establishing a payment channel with him. Let us further assume that Bob and Charlie possess a payment channel between each other.
  - The idea is to use the two channels between Alice and Bob and Bob and Charlie to enable Alice to pay Charlie.

# Payments Without a Direct Channel

- Lightning also allows payments between parties who don't own a direct payment channel between them.
- Let us assume that Alice wants to send 1 ₿ to Charlie without first establishing a payment channel with him. Let us further assume that Bob and Charlie possess a payment channel between each other.
- The idea is to use the two channels between Alice and Bob and Bob and Charlie to enable Alice to pay Charlie.
- Alice, Bob and Charlie proceed as follows:

# Payments Without a Direct Channel

- Lightning also allows payments between parties who don't own a direct payment channel between them.
- Let us assume that Alice wants to send 1  $\text{฿}$  to Charlie without first establishing a payment channel with him. Let us further assume that Bob and Charlie possess a payment channel between each other.
- The idea is to use the two channels between Alice and Bob and Bob and Charlie to enable Alice to pay Charlie.
- Alice, Bob and Charlie proceed as follows:
  - 1 Alice asks Charlie to create a secret X and send her its hash.

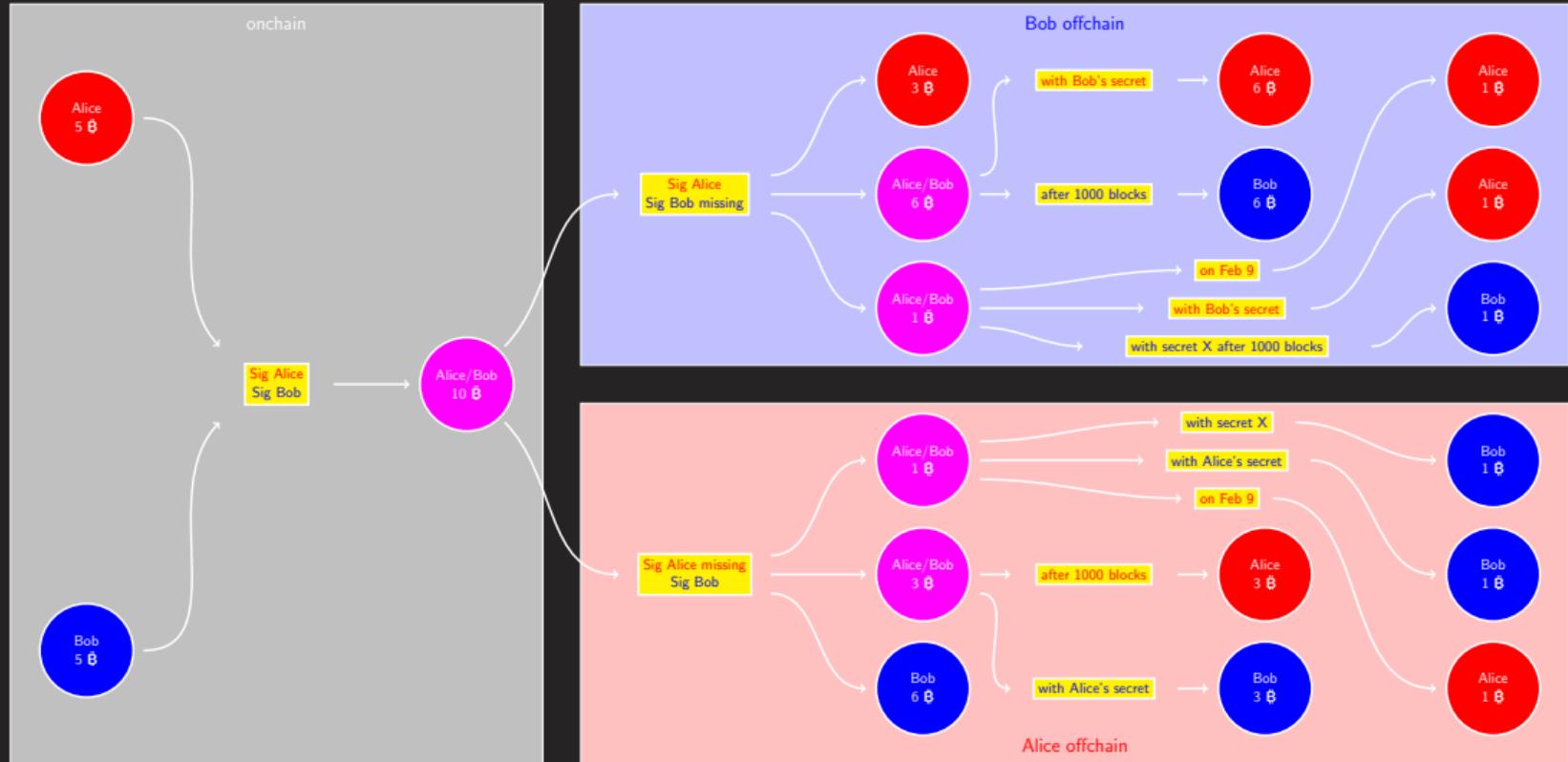
# Payments Without a Direct Channel

- Lightning also allows payments between parties who don't own a direct payment channel between them.
- Let us assume that Alice wants to send 1  $\text{฿}$  to Charlie without first establishing a payment channel with him. Let us further assume that Bob and Charlie possess a payment channel between each other.
- The idea is to use the two channels between Alice and Bob and Bob and Charlie to enable Alice to pay Charlie.
- Alice, Bob and Charlie proceed as follows:
  - 1 Alice asks Charlie to create a secret X and send her its hash.
  - 2 Bob uses his channel to Charlie to pay Charlie 1  $\text{฿}$  in exchange for secret X.

# Payments Without a Direct Channel

- Lightning also allows payments between parties who don't own a direct payment channel between them.
- Let us assume that Alice wants to send 1  $\text{฿}$  to Charlie without first establishing a payment channel with him. Let us further assume that Bob and Charlie possess a payment channel between each other.
- The idea is to use the two channels between Alice and Bob and Bob and Charlie to enable Alice to pay Charlie.
- Alice, Bob and Charlie proceed as follows:
  - 1 Alice asks Charlie to create a secret X and send her its hash.
  - 2 Bob uses his channel to Charlie to pay Charlie 1  $\text{฿}$  in exchange for secret X.
  - 3 Alice uses her channel to Bob to pay Bob 1  $\text{฿}$  in exchange for secret X.
  - 4 Similar to direct payments, steps 2 and 3 will use special **Hash Time-Locked Contracts (HTLCs)**, which will make use of **absolute** time locks instead of **relative** ones.

# Hash Time-Locked Contracts — Channel between Alice and Bob



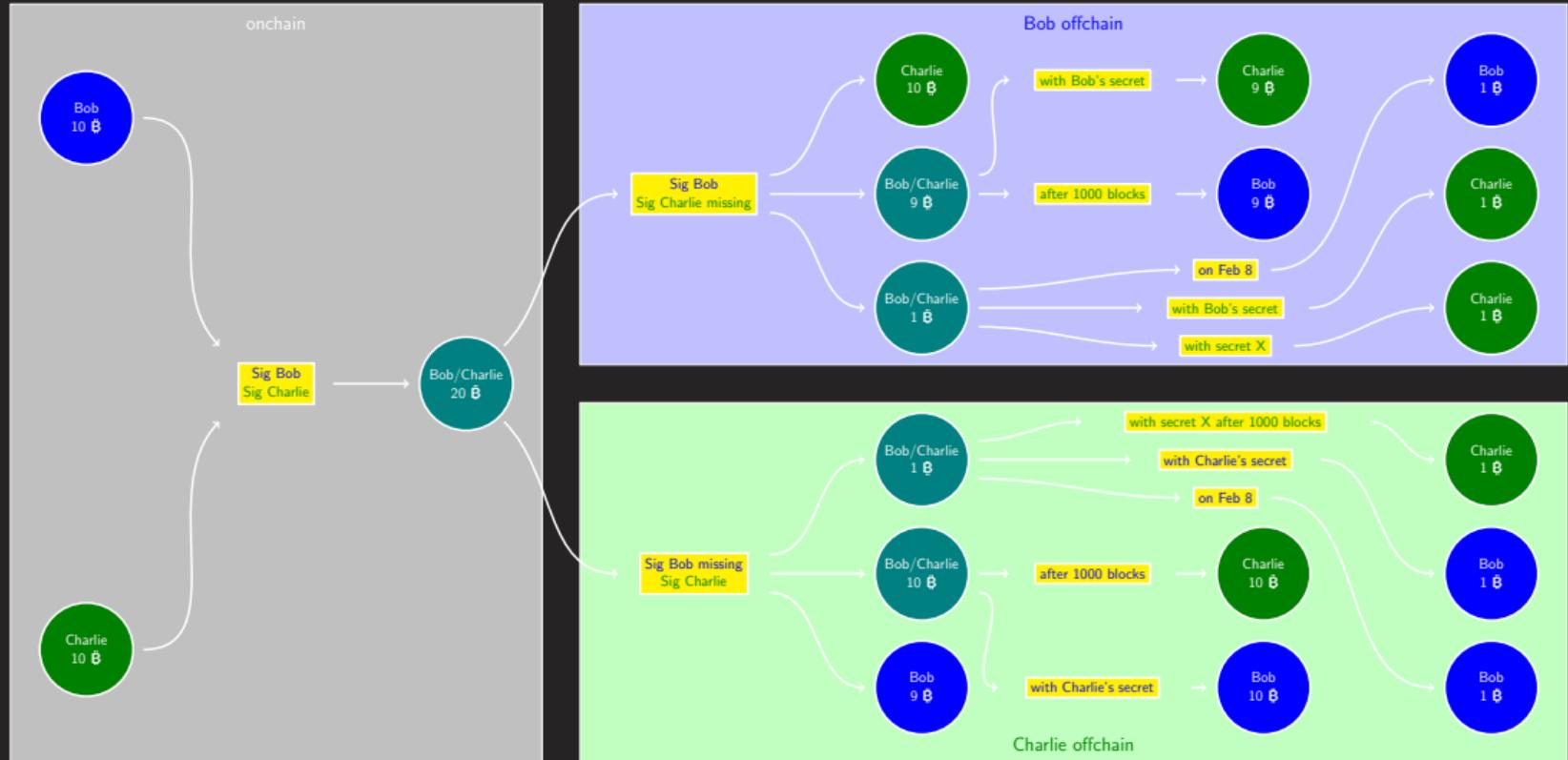
# Explanation

- The parts that are not effected by the payment of 1 ₿ are as before.

# Explanation

- The parts that are not effected by the payment of 1 ₿ are as before.
- For the payment of 1 ₿ a new multisig-address is created for both transactions, which can be unlocked in three different ways:
  - If Bob knows secret X and signs, he gets the money. However, he has to wait for 1000 blocks to receive it if he is the one that closes the channel. If he chooses this option, secret X will be publicly visible on the blockchain.
  - Whoever closes the channel gets the money if he or she knows the other's secret. As before, this makes outdated transactions useless.
  - Independent of who closes the channel, Alice can get her money back on February 9.

# Hash Time-Locked Contracts — Channel between Bob and Charlie



# Explanation

- The situation is analogous to the one between Alice and Bob.

# Explanation

- The situation is analogous to the one between Alice and Bob.
- However, the date when Bob can get his 1 ₿ back is **before** the date when Alice can get *her* 1 ₿ back, so that Bob has time to get his money from Alice as soon as Charlie has revealed secret X.

# Explanation

- The situation is analogous to the one between Alice and Bob.
- However, the date when Bob can get his 1 ₿ back is **before** the date when Alice can get *her* 1 ₿ back, so that Bob has time to get his money from Alice as soon as Charlie has revealed secret X.

## Remark

Of course it is possible to do transactions in the same way with more than one intermediary. The only thing to keep in mind is to set the time-lock dates in a way that give parties further down the chain enough time to react.

# Explanation

- The situation is analogous to the one between Alice and Bob.
- However, the date when Bob can get his 1 ₿ back is **before** the date when Alice can get *her* 1 ₿ back, so that Bob has time to get his money from Alice as soon as Charlie has revealed secret X.

## Remark

As for direct payments, payments using intermediaries normally do not require any transactions to be sent to the blockchain. As long as everybody plays by the rules, everything happens offchain.

# Summary

# Summary

- Scalability is one the biggest challenges facing blockchain technology: Centralized networks like Visa are orders of magnitude faster than current blockchain systems.

# Summary

- Scalability is one the biggest challenges facing blockchain technology: Centralized networks like Visa are orders of magnitude faster than current blockchain systems.
- Bitcoin Lightning is one possible solution to this problem, and its idea is general enough to be applicable to many other blockchain systems.

## Summary

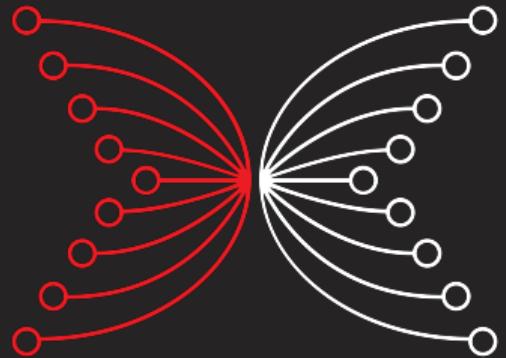
- Scalability is one the biggest challenges facing blockchain technology: Centralized networks like Visa are orders of magnitude faster than current blockchain systems.
  - Bitcoin Lightning is one possible solution to this problem, and its idea is general enough to be applicable to many other blockchain systems.
  - In Lightning, parties create bidirectional payment channels between each other and can then — using “channel chains” — do payments between parties they do not even share a channel with.

## Summary

- Scalability is one the biggest challenges facing blockchain technology: Centralized networks like Visa are orders of magnitude faster than current blockchain systems.
  - Bitcoin Lightning is one possible solution to this problem, and its idea is general enough to be applicable to many other blockchain systems.
  - In Lightning, parties create bidirectional payment channels between each other and can then — using “channel chains” — do payments between parties they do not even share a channel with.
  - As long as anybody plays by the rules, only two Bitcoin transactions are necessary per channel, one to open the channel, one to close it again. All other transactions can be processed fast and cheap “offchain”.

# Summary

- Scalability is one the biggest challenges facing blockchain technology: Centralized networks like Visa are orders of magnitude faster than current blockchain systems.
- Bitcoin Lightning is one possible solution to this problem, and its idea is general enough to be applicable to many other blockchain systems.
- In Lightning, parties create bidirectional payment channels between each other and can then — using “channel chains” — do payments between parties they do not even share a channel with.
- As long as anybody plays by the rules, only two Bitcoin transactions are necessary per channel, one to open the channel, one to close it again. All other transactions can be processed fast and cheap “offchain”.
- Bitcoin guarantees the security of the system: If somebody violates the rules, no honest party loses their money.



INPUT | OUTPUT

# Smart Contracts

Plutus & Marlowe

Lars Brünjes



January 10 2020

# Introduction

# Plutus & Marlowe

- Plutus and Marlowe are two smart-contract languages developed by IOHK.

# Plutus & Marlowe

- Plutus and Marlowe are two smart-contract languages developed by IOHK.
- Although both have been created to add smart-contract capabilities to Cardano, they could in principle be used on other blockchains as well.

# Plutus & Marlowe

- Plutus and Marlowe are two smart-contract languages developed by IOHK.
- Although both have been created to add smart-contract capabilities to Cardano, they could in principle be used on other blockchains as well.
- Both are written in Haskell.

# Plutus & Marlowe

- Plutus and Marlowe are two smart-contract languages developed by IOHK.
- Although both have been created to add smart-contract capabilities to Cardano, they could in principle be used on other blockchains as well.
- Both are written in Haskell.
- Plutus is Turing-complete and general, Marlowe is a *non-Turing-complete DSL for financial contracts*.



# Functional Programming and the UTxO-Model

- Both account-based and UTxO-based blockchains have advantages and disadvantages.

# Functional Programming and the UTxO-Model

- Both account-based and UTxO-based blockchains have advantages and disadvantages.
- In the account-based model, permanently changing account balances constitute **mutable state**, whereas outputs in the UTxO-model are immutable.

# Functional Programming and the UTxO-Model

- Both account-based and UTxO-based blockchains have advantages and disadvantages.
- In the account-based model, permanently changing account balances constitute **mutable state**, whereas outputs in the UTxO-model are immutable.
- This is similar to the relationship between **imperative** programming languages (like Java and Python) and **functional** languages (like Haskell).

# Functional Programming and the UTxO-Model

- Both account-based and UTxO-based blockchains have advantages and disadvantages.
- In the account-based model, permanently changing account balances constitute **mutable state**, whereas outputs in the UTxO-model are immutable.
- This is similar to the relationship between **imperative** programming languages (like Java and Python) and **functional** languages (like Haskell).
- This makes it a good fit to write smart contracts for an account-based blockchain like Ethereum in an imperative language like Solidity and contracts for the UTxO-based Cardano in the functional language Haskell.

# Plutus

# Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds “smart” capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.



# Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds “smart” capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.
- In contrast to Bitcoin, Plutus does not only use input- and output-scripts, but additionally so-called **data scripts**, which are attached to each script-output (but are not part of the output address). Such data scripts can be used to attach arbitrarily complex **state** to outputs, which is not possible in Bitcoin.



# Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds “smart” capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.
- In contrast to Bitcoin, Plutus does not only use input- and output-scripts, but additionally so-called **data scripts**, which are attached to each script-output (but are not part of the output address). Such data scripts can be used to attach arbitrarily complex **state** to outputs, which is not possible in Bitcoin.
- For validation, input-, output- and data-scripts are combined with the transaction which is being validated, so both the state from the data scripts and the transaction itself (with all its inputs and outputs) are available for scrutiny by the validation logic.



# Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds “smart” capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.
- In contrast to Bitcoin, Plutus does not only use input- and output-scripts, but additionally so-called **data scripts**, which are attached to each script-output (but are not part of the output address). Such data scripts can be used to attach arbitrarily complex **state** to outputs, which is not possible in Bitcoin.
- For validation, input-, output- and data-scripts are combined with the transaction which is being validated, so both the state from the data scripts and the transaction itself (with all its inputs and outputs) are available for scrutiny by the validation logic.
- Plutus is also Turing-complete (and uses a similar system to the one Ethereum uses in order to prevent infinite loops).



# Plutus

- Cardano uses UTxO-based accounting like Bitcoin, and similar to how Bitcoin Script adds “smart” capabilities to Bitcoin, Plutus extends the normal UTxO-model by adding scripts to inputs and outputs, whose combination decides whether a transaction is valid or not.
- In contrast to Bitcoin, Plutus does not only use input- and output-scripts, but additionally so-called **data scripts**, which are attached to each script-output (but are not part of the output address). Such data scripts can be used to attach arbitrarily complex **state** to outputs, which is not possible in Bitcoin.
- For validation, input-, output- and data-scripts are combined with the transaction which is being validated, so both the state from the data scripts and the transaction itself (with all its inputs and outputs) are available for scrutiny by the validation logic.
- Plutus is also Turing-complete (and uses a similar system to the one Ethereum uses in order to prevent infinite loops).
- Taken together, these properties make it possible to write smart contracts in Plutus which are as least as powerful as Ethereum smart contracts.



# Plutus Core

- The “machine language” underlying Plutus is called **Plutus Core**.



# Plutus Core

- The “machine language” underlying Plutus is called **Plutus Core**.
- In contrast to Bitcoin Script and the EVM, Plutus Core is *not* stack based, but is instead **System  $F_\omega$** , a variant of the **lambda calculus** with a powerful type system, which also happens to be the foundation **Haskell** is built upon.



# Plutus Core

- The “machine language” underlying Plutus is called **Plutus Core**.
- In contrast to Bitcoin Script and the EVM, Plutus Core is *not* stack based, but is instead **System  $F_\omega$** , a variant of the **lambda calculus** with a powerful type system, which also happens to be the foundation **Haskell** is built upon.
- Not only has Plutus been implemented in Haskell, you also program in Haskell, which means that Haskell is for Plutus Core what Solidity is for the EVM.



# Plutus Core

- The “machine language” underlying Plutus is called **Plutus Core**.
- In contrast to Bitcoin Script and the EVM, Plutus Core is *not* stack based, but is instead **System  $F_\omega$** , a variant of the **lambda calculus** with a powerful type system, which also happens to be the foundation **Haskell** is built upon.
- Not only has Plutus been implemented in Haskell, you also program in Haskell, which means that Haskell is for Plutus Core what Solidity is for the EVM.
- This enables a seamless interplay between onchain code and offchain code (whereas Solidity only supports onchain code, so that offchain code has to be written in something like JavaScript.)



# Lambda Calculus

# Untyped Lambda Calculus

- Provides simple semantics and a formal model for computation.

# Untyped Lambda Calculus

- Provides simple semantics and a formal model for computation.
- Turing complete.

# Untyped Lambda Calculus

- Provides simple semantics and a formal model for computation.
- Turing complete.
- Makes two simplifications:
  - only anonymous functions
  - only functions of one argument (curried functions)

# Untyped Lambda Calculus

- Provides simple semantics and a formal model for computation.
- Turing complete.
- Makes two simplifications:
  - only anonymous functions
  - only functions of one argument (curried functions)
- Haskell is based upon and compiles to a (typed!) version of the lambda Calculus (as first intermediate compiler target, Core).

# History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.

# History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.
- Introduced by Alonzo Church in the 1930s.

# History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.
- Introduced by Alonzo Church in the 1930s.
- Shown to be logically inconsistent in 1935 by Stephen Kleene and J. B. Rosser.

# History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.
- Introduced by Alonzo Church in the 1930s.
- Shown to be logically inconsistent in 1935 by Stephen Kleene and J. B. Rosser.
- Fixed by Church in 1936 – Untyped Lambda Calculus.

# History

- Based upon work by Frege from 1893 and Schönfinkel from the 1920s.
- Introduced by Alonzo Church in the 1930s.
- Shown to be logically inconsistent in 1935 by Stephen Kleene and J. B. Rosser.
- Fixed by Church in 1936 – Untyped Lambda Calculus.
- Relation to programming languages clarified in the 1960s.

# Lambda expressions

Lambda expressions (or lambda terms) are composed of

- variables  $v_1, v_2, \dots, v_n, \dots$ ,
- the abstraction symbols  $\lambda$  and  $.$ ,
- parentheses  $()$ .

The set of lambda expressions  $\Lambda$  is inductively defined as:

- If  $x$  is a variable, then  $x \in \Lambda$ . (variable)
- If  $x$  is a variable and  $M \in \Lambda$ , then  $(\lambda x.M) \in \Lambda$ . (lambda abstraction)
- If  $M, N \in \Lambda$ , then  $(MN) \in \Lambda$ . (application)

# Free variables

- Let  $V$  be the set of variables. For each lambda expression  $M \in \Lambda$ , we define the set of free variables  $FV(M) \subset V$  as follows:

# Free variables

- Let  $V$  be the set of variables. For each lambda expression  $M \in \Lambda$ , we define the set of **free variables**  $FV(M) \subset V$  as follows:
  - For a variable  $x \in V$ ,  $FV(x) = \{x\}$ .

# Free variables

- Let  $V$  be the set of variables. For each lambda expression  $M \in \Lambda$ , we define the set of **free variables**  $FV(M) \subset V$  as follows:
  - For a variable  $x \in V$ ,  $FV(x) = \{x\}$ .
  - For an abstraction,  $FV(\lambda x.M) = FV(M) \setminus \{x\}$ .

# Free variables

- Let  $V$  be the set of variables. For each lambda expression  $M \in \Lambda$ , we define the set of **free variables**  $FV(M) \subset V$  as follows:
  - For a variable  $x \in V$ ,  $FV(x) = \{x\}$ .
  - For an abstraction,  $FV(\lambda x.M) = FV(M) \setminus \{x\}$ .
  - For an application,  $FV(MN) = FV(M) \cup FV(N)$ .

# Free variables

- Let  $V$  be the set of variables. For each lambda expression  $M \in \Lambda$ , we define the set of **free variables**  $FV(M) \subset V$  as follows:
  - For a variable  $x \in V$ ,  $FV(x) = \{x\}$ .
  - For an abstraction,  $FV(\lambda x.M) = FV(M) \setminus \{x\}$ .
  - For an application,  $FV(MN) = FV(M) \cup FV(N)$ .
- Given a lambda expression  $M \in \Lambda$ , we call a variable  $x \in V$  **free** (in  $M$ ) if  $x \in FV(M)$ .

# Free variables

- Let  $V$  be the set of variables. For each lambda expression  $M \in \Lambda$ , we define the set of **free variables**  $FV(M) \subset V$  as follows:
  - For a variable  $x \in V$ ,  $FV(x) = \{x\}$ .
  - For an abstraction,  $FV(\lambda x.M) = FV(M) \setminus \{x\}$ .
  - For an application,  $FV(MN) = FV(M) \cup FV(N)$ .
- Given a lambda expression  $M \in \Lambda$ , we call a variable  $x \in V$  **free** (in  $M$ ) if  $x \in FV(M)$ .
- In an abstraction  $\lambda x.M$ , we call the variable  $x$  **bound**.

# $\beta$ -Reduction

- Consider a lambda expression of the form  $(\lambda x.M) N$ , i.e. an application where the first argument is an abstraction.

# $\beta$ -Reduction

- Consider a lambda expression of the form  $(\lambda x.M) N$ , i.e. an application where the first argument is an abstraction.
- By definition, this term  $\beta$ -reduces to  $M[x := N]$ , the substitution of all (free) occurrences of variable  $x$  in  $M$  by  $N$  (maybe after first renaming  $x$  to a variable which is not free in  $N$ ).

# $\beta$ -Reduction

- Consider a lambda expression of the form  $(\lambda x.M) N$ , i.e. an application where the first argument is an abstraction.
- By definition, this term  $\beta$ -reduces to  $M[x := N]$ , the substitution of all (free) occurrences of variable  $x$  in  $M$  by  $N$  (maybe after first renaming  $x$  to a variable which is not free in  $N$ ).
- This act of “plugging in” an expression for the bound variable in an abstraction is what constitutes the idea of computation in lambda Calculus.

# Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.

# Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.

# Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying  $f$  and  $x$  to a natural number is applying  $f$   $n$ -times to  $x$ :

# Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying  $f$  and  $x$  to a natural number is applying  $f$   $n$ -times to  $x$ :
- $\text{zero} := \lambda f x. x$ .

# Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying  $f$  and  $x$  to a natural number is applying  $f$   $n$ -times to  $x$ :
- $\text{zero} := \lambda f x. x$ .
- $\text{succ} := \lambda n f x. f (n f x)$ .

# Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying  $f$  and  $x$  to a natural number is applying  $f$   $n$ -times to  $x$ :
- $\text{zero} := \lambda f x. x$ .
- $\text{succ} := \lambda n f x. f(n f x)$ .
- We can define addition:  $\text{add} := \lambda m n f x. m f(n f x)$

# Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying  $f$  and  $x$  to a natural number is applying  $f$   $n$ -times to  $x$ :
- $\text{zero} := \lambda f x. x$ .
- $\text{succ} := \lambda n f x. f(n f x)$ .
- We can define addition:  $\text{add} := \lambda m n f x. m f(n f x)$
- and multiplication:  $\text{mul} := \lambda m n f x. m(n f)x$

# Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying  $f$  and  $x$  to a natural number is applying  $f$   $n$ -times to  $x$ :
- $\text{zero} := \lambda f x. x$ .
- $\text{succ} := \lambda n f x. f(n f x)$ .
- We can define addition:  $\text{add} := \lambda m n f x. m f(n f x)$
- and multiplication:  $\text{mul} := \lambda m n f x. m(n f x)$
- and predecessor (more complicated!):  $\text{pred} := \lambda n f x. n(\lambda g h. h(g f))(\lambda u. x)(\lambda u. u)$ .

# Church encoding

- It is possible to encode an amazing range of datatypes in the untyped lambda calculus.
- Examples are natural numbers, booleans, pairs, lists and sums.
- Natural numbers are encoded as functions taking two arguments, where the result of applying  $f$  and  $x$  to a natural number is applying  $f$   $n$ -times to  $x$ :
- $\text{zero} := \lambda f x. x$ .
- $\text{succ} := \lambda n f x. f(n f x)$ .
- We can define addition:  $\text{add} := \lambda m n f x. m f(n f x)$
- and multiplication:  $\text{mul} := \lambda m n f x. m(n f x)$
- and predecessor (more complicated!):  $\text{pred} := \lambda n f x. n(\lambda g h. h(g f))(\lambda u. x)(\lambda u. u)$ .
- and many more...

# Typed Lambda Calculi

- The **simply typed lambda calculus** is a typed interpretation of the lambda calculus with only one type constructor “ $\rightarrow$ ” that builds function types.

# Typed Lambda Calculi

- The **simply typed lambda calculus** is a typed interpretation of the lambda calculus with only one type constructor “ $\rightarrow$ ” that builds function types.
- **System F** is a **typed** lambda calculus that differs from the simply typed lambda calculus by the introduction of a mechanism of universal quantification over types (**polymorphism**).

$$\text{id} := \Lambda a. \lambda(x : a). x : \forall a. a \rightarrow a$$

# Typed Lambda Calculi

- The **simply typed lambda calculus** is a typed interpretation of the lambda calculus with only one type constructor “ $\rightarrow$ ” that builds function types.
- **System F** is a **typed** lambda calculus that differs from the simply typed lambda calculus by the introduction of a mechanism of universal quantification over types (**polymorphism**).

$$\text{id} := \Lambda a. \lambda(x : a). x : \forall a. a \rightarrow a$$

- **System  $F_\omega$**  adds functions from types to types (**type constructors**) to System F.

# Marlowe

# Marlowe

- Marlowe is a DSL (Domain Specific Language) for financial contracts.



# Marlowe

- Marlowe is a DSL (Domain Specific Language) for financial contracts.
- It is based on the seminal paper “Composing Contracts: An Adventure in Financial Engineering” by Simon Peyton Jones et al. from 2000 (Simon Peyton Jones is one of the creators of Haskell), which has been adapted from “real world” finance to the special circumstances on the blockchain.



# Marlowe

- Marlowe is a DSL (Domain Specific Language) for financial contracts.
- It is based on the seminal paper “Composing Contracts: An Adventure in Financial Engineering” by Simon Peyton Jones et al. from 2000 (Simon Peyton Jones is one of the creators of Haskell), which has been adapted from “real world” finance to the special circumstances on the blockchain.
- Marlowe is *not* Turing-complete, and it is simple enough to allow for static analysis, which can automatically derive and prove important properties of Marlowe contracts.



# Marlowe

- Marlowe is a DSL (Domain Specific Language) for financial contracts.
- It is based on the seminal paper “Composing Contracts: An Adventure in Financial Engineering” by Simon Peyton Jones et al. from 2000 (Simon Peyton Jones is one of the creators of Haskell), which has been adapted from “real world” finance to the special circumstances on the blockchain.
- Marlowe is *not* Turing-complete, and it is simple enough to allow for static analysis, which can automatically derive and prove important properties of Marlowe contracts.
- Marlowe is also powerful enough to implement a significant part of all commonly used financial contracts.



# Predictable Runtime

- All Marlowe contracts finish after finitely many steps (so there are no infinite loops), and static analysis can determine an upper bound for the maximum number of steps, so it is possible to know in advance for how many steps a given contract will run at most.



# Predictable Runtime

- All Marlowe contracts finish after finitely many steps (so there are no infinite loops), and static analysis can determine an upper bound for the maximum number of steps, so it is possible to know in advance for how many steps a given contract will run at most.
- Marlowe guarantees that no money will be “trapped” in a contract forever: By the end of the contract, all money that has been paid into the contract will have been paid back to one of the parties participating in the contract.



# EDSL

- Marlowe is a special type of DSL, a so-called **EDSL**, an **Embedded Domain Specific Language**, i.e. a DSL *embedded* into a “host language”.



# EDSL

- Marlowe is a special type of DSL, a so-called **EDSL**, an **Embedded Domain Specific Language**, i.e. a DSL *embedded* into a “host language”.
- Marlowe’s host language is **Haskell**.



# EDSL

- Marlowe is a special type of DSL, a so-called **EDSL**, an **Embedded Domain Specific Language**, i.e. a DSL *embedded* into a “host language”.
- Marlowe’s host language is **Haskell**.
- An EDSL has the advantage that all features of the host language are available to help facilitate creating expressions in the DSL.



# EDSL

- Marlowe is a special type of DSL, a so-called **EDSL**, an **Embedded Domain Specific Language**, i.e. a DSL *embedded* into a “host language”.
- Marlowe’s host language is **Haskell**.
- An EDSL has the advantage that all features of the host language are available to help facilitate creating expressions in the DSL.
- This means for Marlowe, that we can use Haskell to write Marlowe contracts and that Marlowe contracts are nothing more than values of a specific Haskell type.



# Blockly

- Alternatively, Marlowe contracts can be written in **Blockly**, a simple graphical language.



# Blockly

- Alternatively, Marlowe contracts can be written in **Blockly**, a simple graphical language.
- This is tedious for complex contracts, but a good way for beginners.



- The **Actus Financial Research Foundation** has created a standard taxonomy to categorize and specify financial contracts.

- The **Actus Financial Research Foundation** has created a standard taxonomy to categorize and specify financial contracts.
- The Actus Standard considers financial contracts as legally binding agreements between two or more parties about future payments. Historically, such contracts have often been written in “plain English”, which has led to ambiguity and misunderstandings.

- The **Actus Financial Research Foundation** has created a standard taxonomy to categorize and specify financial contracts.
- The Actus Standard considers financial contracts as legally binding agreements between two or more parties about future payments. Historically, such contracts have often been written in “plain English”, which has led to ambiguity and misunderstandings.
- Instead of using normal prose, the Actus Standard composes contracts from a well-defined set of contractual terms and deterministic functions, which map those terms to future payment obligations.

- The **Actus Financial Research Foundation** has created a standard taxonomy to categorize and specify financial contracts.
- The Actus Standard considers financial contracts as legally binding agreements between two or more parties about future payments. Historically, such contracts have often been written in “plain English”, which has led to ambiguity and misunderstandings.
- Instead of using normal prose, the Actus Standard composes contracts from a well-defined set of contractual terms and deterministic functions, which map those terms to future payment obligations.
- Using this method makes it possible to classify and describe the majority of all financial instruments in about 30 types and patterns.

- The **Actus Financial Research Foundation** has created a standard taxonomy to categorize and specify financial contracts.
- The Actus Standard considers financial contracts as legally binding agreements between two or more parties about future payments. Historically, such contracts have often been written in “plain English”, which has led to ambiguity and misunderstandings.
- Instead of using normal prose, the Actus Standard composes contracts from a well-defined set of contractual terms and deterministic functions, which map those terms to future payment obligations.
- Using this method makes it possible to classify and describe the majority of all financial instruments in about 30 types and patterns.
- IOHK plans to use Plutus and Marlowe to implement the complete Actus Standard in Cardano.

# Smart Contracts

## Marlowe

Lars Brünjes



January 10 2020

# Parties & Accounts

```
newtype PubKey = PubKey Text  
    deriving (Eq,Ord)
```

```
type Party = PubKey
```

```
type NumAccount = Integer
```

```
data AccountId = AccountId NumAccount Party  
    deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A **party** is a participant in the contract. Parties can perform **actions** like depositing money into an account. Marlowe also has a concept of **accounts** to make contract creation easier. Accounts are given by a combination of a number and a party. This party will get all remaining money at the end of the contract. Accounts are local to the contract.

# The Contract Type

```
data Contract = Close
  | Pay AccountId Payee Value Contract
  | If Observation Contract Contract
  | When [Case] Timeout Contract
  | Let Valueld Value Contract
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

# The Contract Type

```
data Contract = Close
  | Pay AccountId Payee Value Contract
  | If Observation Contract Contract
  | When [Case] Timeout Contract
  | Let Valueld Value Contract
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Close is the simplest contract: It closes the contract and provides refunds to the owners of accounts that contain a positive balance. This is performed one account per step, but all accounts will be refunded in a single transaction.

# The Contract Type

```
data Contract = Close
  | Pay AccountId Payee Value Contract
  | If Observation Contract Contract
  | When [Case] Timeout Contract
  | Let Valueld Value Contract
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A payment contract Pay a p v cont will make a payment of value v from the account a to a payee p, which will be one of the contract participants or another account in the contract. Warnings will be generated if the value v is negative, or if there is not enough in the account to make the payment in full. In that case a partial payment (of all the money available) is made. The continuation contract is the one given in the contract: cont.

# The Contract Type

```
data Contract = Close
  | Pay AccountId Payee Value Contract
  | If Observation Contract Contract
  | When [Case] Timeout Contract
  | Let Valueld Value Contract
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

The conditional If obs cont1 cont2 will continue as cont1 or cont2, depending on the Boolean value of the observation obs when this construct is executed.

# The Contract Type

```
data Contract = Close
  | Pay AccountId Payee Value Contract
  | If Observation Contract Contract
  | When [Case] Timeout Contract
  | Let Valueld Value Contract
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

When cases timeout cont is the most complex constructor for contracts. It is a contract that is triggered on actions, which may or may not happen at any particular slot: What happens when various actions happen is described by the cases in the contract.

The list cases contains a collection of cases. Each case has the form Case ac co where ac is an action and co a continuation. When a particular action happens, the contract will continue as the corresponding continuation.

In order to make sure that the contract makes progress eventually, the contract will continue as cont once timeout is reached.

# The Contract Type

```
data Contract = Close
  | Pay AccountId Payee Value Contract
  | If Observation Contract Contract
  | When [Case] Timeout Contract
  | Let Valueld Value Contract
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A contract Let id val cont allows a contract to name a value using an identifier. In this case, the expression val is evaluated, and stored with the name id. The contract then continues as cont.

As well as allowing us to use abbreviations, this mechanism also means that we can capture and save volatile values that might be changing with time, e.g. the current price of oil, or the current slot number, at a particular point in the execution of the contract, to be used later on in contract execution.

# The Observation Type

```
data Observation = AndObs Observation Observation
                  | OrObs Observation Observation
                  | NotObs Observation
                  | ChoseSomething Choiceld
                  | ValueGE Value Value
                  | ValueGT Value Value
                  | ValueLT Value Value
                  | ValueLE Value Value
                  | ValueEQ Value Value
                  | TrueObs
                  | FalseObs
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Observations are Boolean value that come from combining other observations, from comparing values or — in the case of ChoseSomething — if a party made a choice.

# The Value Type

```
data Value = AvailableMoney AccountId
            | Constant Integer
            | NegValue Value
            | AddValue Value Value
            | SubValue Value Value
            | ChoiceValue ChoicId Value
            | SlotIntervalStart
            | SlotIntervalEnd
            | UseValue Valueld
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Values are values that can sometimes change over time — like the money available in an account or the current slot number.

# The Payee Type

```
data Payee = Account AccountId  
          | Party Party  
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Payments can be made to in-contract accounts (constructor Account) or to parties (Party constructor).

# The Case Type

```
data Case = Case Action Contract  
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

# The Bound Type

```
data Bound = Bound Integer Integer  
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

# The Action Type

```
data Action = Deposit AccountId Party Value
            | Choice ChoicId [Bound]
            | Notify Observation
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Marlowe distinguishes between three different types of **actions** (which are triggered externally, outside of the contract's control).

# The Action Type

```
data Action = Deposit AccountId Party Value
            | Choice Choiceld [Bound]
            | Notify Observation
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A Deposit n p v makes a deposit of value v into account number n belonging to party p.

# The Action Type

```
data Action = Deposit AccountId Party Value
            | Choice ChoicId [Bound]
            | Notify Observation
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

A choice is made for a particular id with a list of bounds on the values that are acceptable.  
For example, [Bound 0 0, Bound 3 5] offers the choice of one of 0, 3, 4 and 5.

# The Action Type

```
data Action = Deposit AccountId Party Value
            | Choice Choiceld [Bound]
            | Notify Observation
deriving (Eq,Ord,Show,Read,Generic,Pretty)
```

Notify obs notifies the contract of an observation obs that has been made. Typically this would be done by one of the parties, or one of their wallets acting automatically.

## Exercises

- Write four Marlowe contracts in which Alice is supposed to first deposit 100 Lovelace into the contract. If she does not do this until Slot 5, nothing happens. If she does,
  - In the first contract, her money is paid **to Bob**.
  - in the second contract, her money should be paid **to Bob and Charlie** in equal parts,
  - in the third contract, she gets her money back **in Slot 10**, and
  - in the fourth contract, Bob **can choose** whether the money goes to himself or to Charlie. If Bob does not make a choice until Slot 10, the money goes back to Alice.
- Write a Marlowe contract in which Alice can choose an amount between 100 and 200 Lovelace and deposit it into the contract until Slot 3. If she does not do this until Slot 3, nothing happens. If she does, Bob gets the chosen amount.

# Example: Simple Crowd Sourcing

```
--{-# LANGUAGE OverloadedStrings #-}
module Main where

import Language.Marlowe

main :: IO ()
main = print . pretty $ contract
    "Alice" 300 ["Bob", "Charlie", "Dora", "Eve"] 100 10

contract :: Party -- campaign owner
    -> Integer -- funding target
    -> [Party] -- contributors
    -> Integer -- contribution
    -> Slot -- deadline
    -> Contract

contract owner target contributors contribution deadline
| not (enough contributors) = Close
| otherwise                 = go [] contributors

where
    go :: [Party] -> [Party] -> Contract
    go ps [] = check ps
    go ps qs =
        When
            [mkCase ps qs q | q <- qs]
        deadline
```

```
-- mkCase :: [Party] -> [Party] -> Party -> Case
mkCase ps qs q =
    Case
        (Deposit (account q) q $ Constant contribution) $
        go (q : ps) $ filter (/= q) qs

account :: Party -> AccountId
account = AccountId 1

enough :: [Party] -> Bool
enough ps = contribution * fromIntegral (length ps) >= target

check :: [Party] -> Contract
check ps
| enough ps = pay ps
| otherwise = Close

pay :: [Party] -> Contract
pay [] = Close
pay (p : ps) = Pay (account p) (Party owner) (Constant contribution) $ pay ps
```

# Example: Simple Crowd Sourcing

```
--{-# LANGUAGE OverloadedStrings #-}
module Main where

import Language.Marlowe

main :: IO ()
main = print . pretty $ contract
  "Alice" 300 ["Bob", "Charlie", "Dora", "Eve"] 100 10

contract :: Party -- campaign owner
  -> Integer -- funding target
  -> [Party] -- contributors
  -> Integer -- contribution
  -> Slot -- deadline
  -> Contract

contract owner target contributors contribution deadline
| not (enough contributors) = Close
| otherwise                 = go [] contributors

where
  go :: [Party] -> [Party] -> Contract
  go ps [] = check ps
  go ps qs =
    When
      [mkCase ps qs q | q <- qs]
      deadline
```

```
-- mkCase :: [Party] -> [Party] -> Party -> Case
mkCase ps qs q =
  Case
    (Deposit (account q) q $ Constant contribution) $
      go (q : ps) $ filter (/= q) qs

  account :: Party -> AccountId
  account = AccountId 1

  enough :: [Party] -> Bool
  enough ps = contribution * fromIntegral (length ps) >= target

  check :: [Party] -> Contract
  check ps
    | enough ps = pay ps
    | otherwise = Close

  pay :: [Party] -> Contract
  pay [] = Close
  pay (p : ps) = Pay (account p) (Party owner) (Constant contribution) $ pay ps
```

## Remark

For more complex contracts, Blockly becomes infeasible, and using the full power of Haskell makes things much more concise.

# Projects

- Tasks

- Write a Marlowe contract that simulates a **Dutch Auction**: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.

# Projects

- Tasks

- Write a Marlowe contract that simulates a **Dutch Auction**: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
- Write a Marlowe contract that simulates a **First Price Sealed Bid Auction**: There is one round of (normally **secret** bidding, but we cannot do this in Marlowe), and the highest bidder wins.

# Projects

- Tasks

- Write a Marlowe contract that simulates a **Dutch Auction**: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
- Write a Marlowe contract that simulates a **First Price Sealed Bid Auction**: There is one round of (normally **secret** bidding, but we cannot do this in Marlowe), and the highest bidder wins.
- Write a Marlowe contract that simulates an **English Auction**: There are several rounds in which bidders can increase their bids until the highest bidder wins.

# Projects

- Tasks

- Write a Marlowe contract that simulates a **Dutch Auction**: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
- Write a Marlowe contract that simulates a **First Price Sealed Bid Auction**: There is one round of (normally **secret** bidding, but we cannot do this in Marlowe), and the highest bidder wins.
- Write a Marlowe contract that simulates an **English Auction**: There are several rounds in which bidders can increase their bids until the highest bidder wins.

- Remarks

- You can model “winning” as the winner paying his bid to the auction owner. In reality, the auction owner would of course hand over the auctioned item in return, but we cannot model that in Marlowe.

# Projects

- Tasks

- Write a Marlowe contract that simulates a **Dutch Auction**: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
- Write a Marlowe contract that simulates a **First Price Sealed Bid Auction**: There is one round of (normally **secret** bidding, but we cannot do this in Marlowe), and the highest bidder wins.
- Write a Marlowe contract that simulates an **English Auction**: There are several rounds in which bidders can increase their bids until the highest bidder wins.

- Remarks

- You can model “winning” as the winner paying his bid to the auction owner. In reality, the auction owner would of course hand over the auctioned item in return, but we cannot model that in Marlowe.
- Ideally, you would parameterize your solution over the list of bidders, but for simplicity, you can use just two bidders.

# Projects

- Tasks

- Write a Marlowe contract that simulates a **Dutch Auction**: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
- Write a Marlowe contract that simulates a **First Price Sealed Bid Auction**: There is one round of (normally **secret** bidding, but we cannot do this in Marlowe), and the highest bidder wins.
- Write a Marlowe contract that simulates an **English Auction**: There are several rounds in which bidders can increase their bids until the highest bidder wins.

- Remarks

- You can model “winning” as the winner paying his bid to the auction owner. In reality, the auction owner would of course hand over the auctioned item in return, but we cannot model that in Marlowe.
- Ideally, you would parameterize your solution over the list of bidders, but for simplicity, you can use just two bidders.
- Make sure that no bidder can make a bid without being forced to actually pay if he wins the auction.

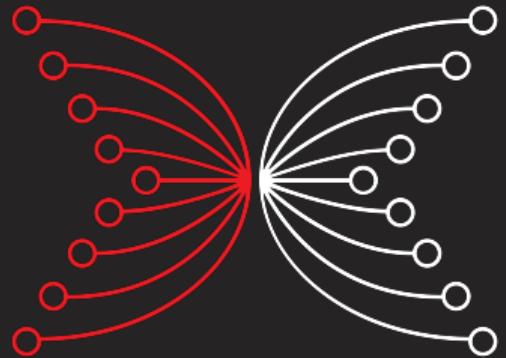
# Projects

- Tasks

- Write a Marlowe contract that simulates a **Dutch Auction**: Bidding starts at a maximum amount and is gradually lowered to a minimum amount. The first bidder that pays the current amount wins.
- Write a Marlowe contract that simulates a **First Price Sealed Bid Auction**: There is one round of (normally **secret** bidding, but we cannot do this in Marlowe), and the highest bidder wins.
- Write a Marlowe contract that simulates an **English Auction**: There are several rounds in which bidders can increase their bids until the highest bidder wins.

- Remarks

- You can model “winning” as the winner paying his bid to the auction owner. In reality, the auction owner would of course hand over the auctioned item in return, but we cannot model that in Marlowe.
- Ideally, you would parameterize your solution over the list of bidders, but for simplicity, you can use just two bidders.
- Make sure that no bidder can make a bid without being forced to actually pay if he wins the auction.
- Make also sure that everybody else gets back their money in the end.



INPUT | OUTPUT