UNIVERSITY *of* NICOSIA

Week 5 – Session 9

Programmable Chains

BLOC 512: Blockchain Systems and Architectures

Session Objectives

- Discuss various deployment options (Blockchains-as-a-Service)
- Explore the expressivity of smart contracts that are interpreted by various chains

Agenda

1. Introduction
2. Blockchains as a software
3. Smart contracts
4. EVM
5. WASM
6. eWASM

Introduction

5 stages in blockchain app development evolution

Blockchain 1.0

2009

Cryptocurrencies

Blockchain 2.0

2015

Smart contracts

Blockchain 3.0

2017

Decentralized Applications

Blockchain 4.0

2019

&

Interoperability and Scalability

Blockchain 5.0

2020

CBDCs

NFTs

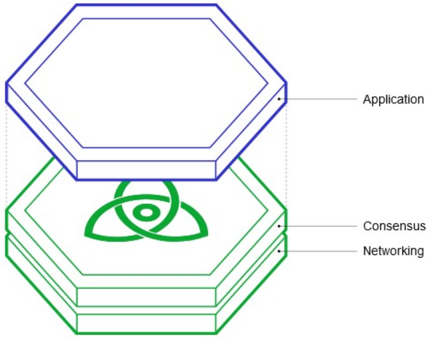
Introduction

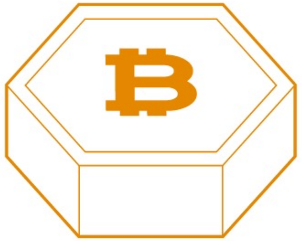
There is also an evolution in blockchain architectures

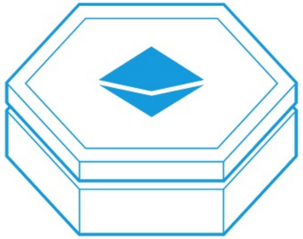
Blockchain Components

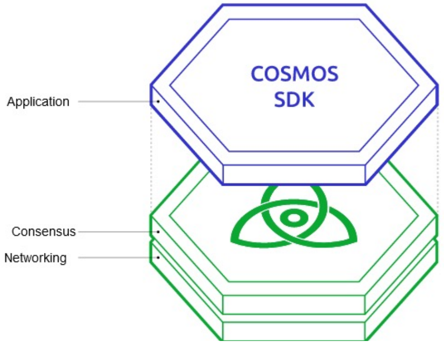
- Application
- Consensus
- Networking

Blockchain 1.0	Blockchain 2.0	Blockchain 3.0	Blockchain 4.0	Blockchain 5.0
2009	2015	2017	2019	2020
			BLOCKCHAIN INTEROPERABILITY & SCALABILITY	
Cryptocurrencies Evolution	Smart contracts Evolution	Decentralized Applications Evolution	Interoperability and Scalability Evolution	CBDC Evolution NFT Evolution



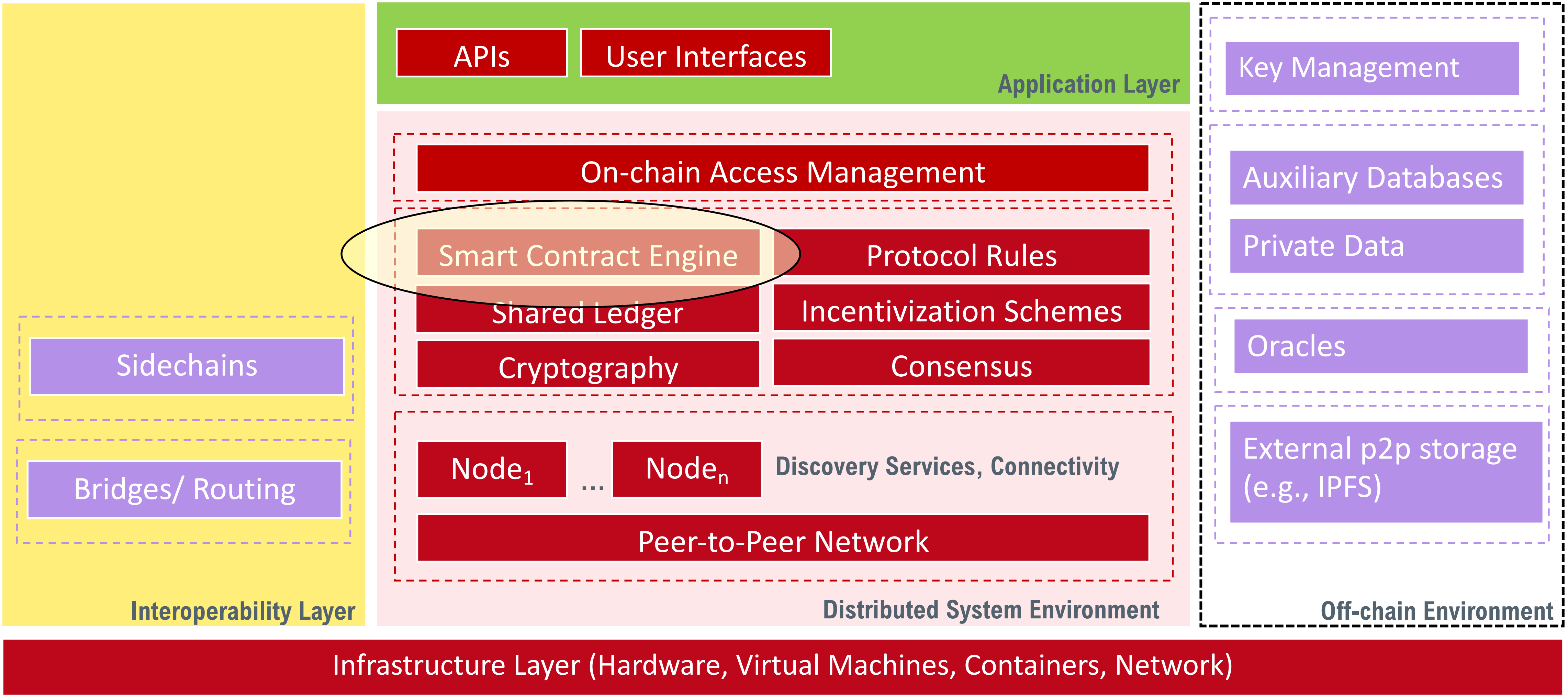






Asset Layer	Fungible Token: ERC20	Non-Fungible Token: ERC721/1155	...
Settlement Layer	(Ethereum) Blockchain		

Introduction – DLTs architectural components



Blockchains as a software service

Blockchains as a software architecture

- Software architecture – consists of software components
- Blockchain can be considered as a software component
- Allows us to understand architectural impacts on performance and attributes
- Attributes may include
  - Scalability
  - Security
  - Privacy
  - Sustainability etc
- Support decision making on adopting Blockchain components(e.g., which functionalities allocate in which components)
- Key decisions for blockchain components
- To use blockchain or other components
- Which parts of data/functionality to be placed on-chain and off-chain
- ledger and smart contracts can be used

Blockchains as a software architecture

Blockchains as software components, can provide:

- data storage,
- computation services,
- communication services,
- asset management and control functions.

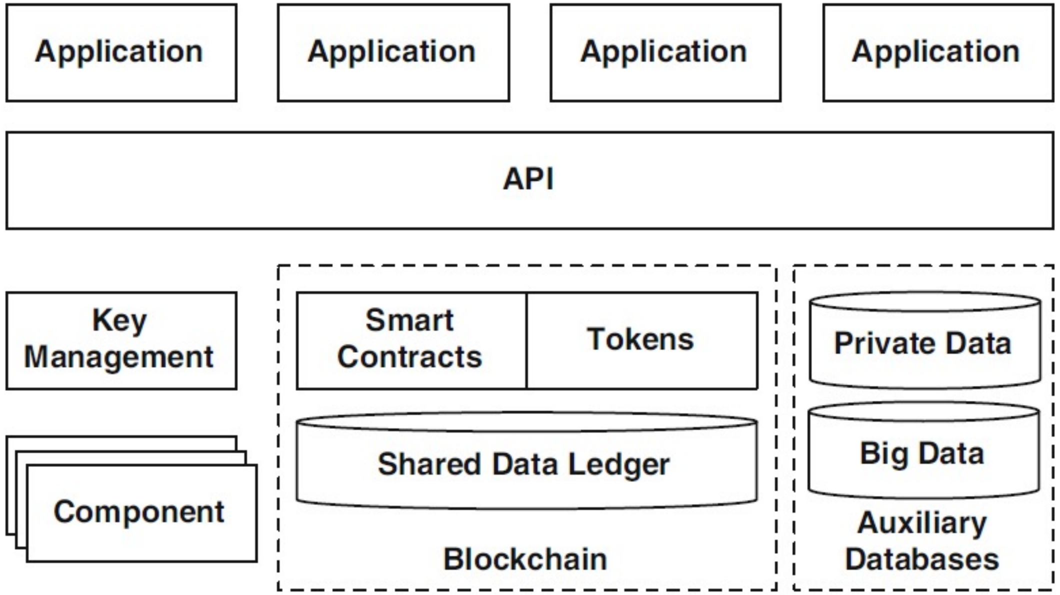
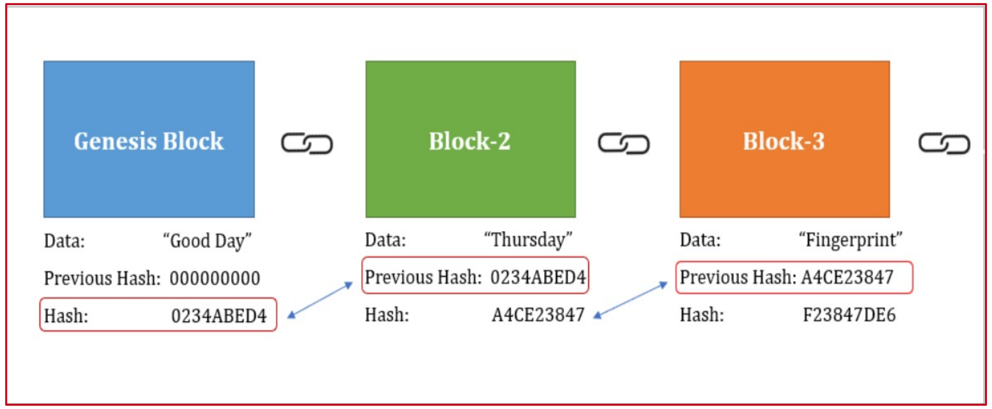


Fig. 5.1 Blockchain in a software architecture. This work is based on an earlier work: Xu et al. (2018) © ACM, 2018. https://doi.org/10.1145/doi. Included here by permission

Blockchains as a data store

- Transactions are recorded in Blocks
- Each Block contains a signature of the previous Block
- Linking them together in a chain
- Transactions are administrated
- In a distributed
- Totally decentralized way
- Rise of a new way to store and exchange information
- Transactions record data and transfer digital assets among participants
- Add data into:
  - transactions (e.g., OP_RETURN Bitcoin)
  - contract storage (e.g., Ethereum)



Blockchains as computational service

Bitcoin:

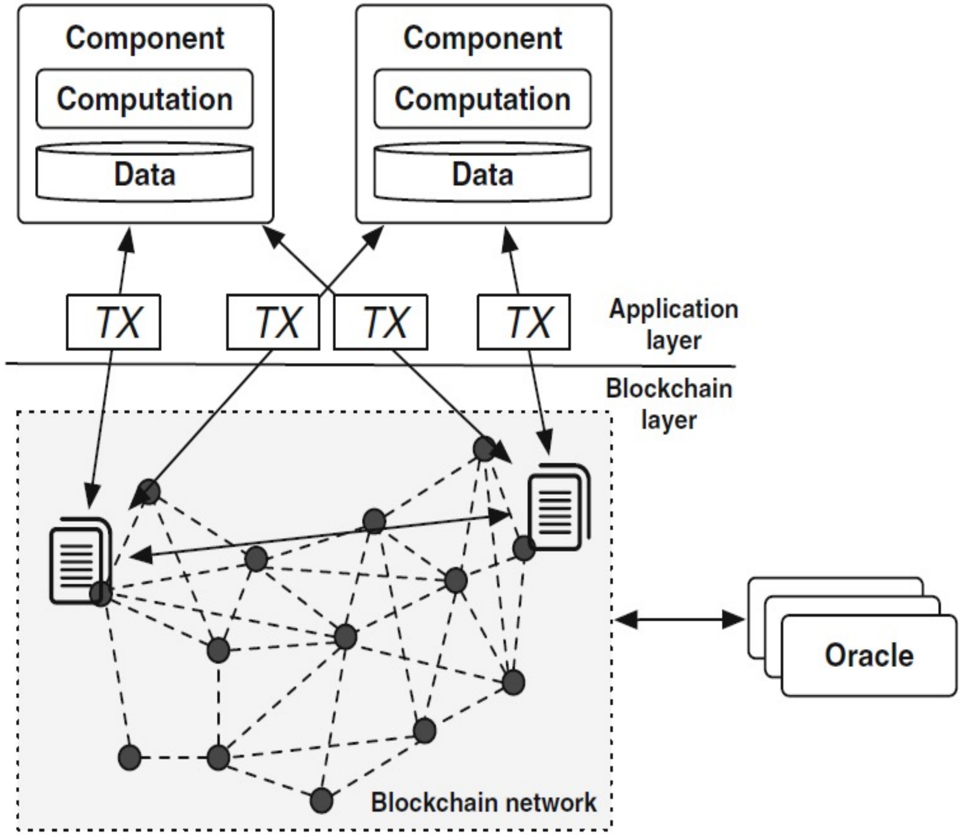
- limited native capability for programmable transactions.
- Simple smart contracts (do not support complex control flow).
- end users can build self-executing contracts through external services
- smart contract execution is performed by external oracles.

Smart contracts in Ethereum and other blockchains

- More advanced and powerful
- Increased functionality and performance
- Used by components connected to a blockchain to reach agreements and solve common problems with minimal trust.
- Smart contract example:
- Enable machine-to machine communication in an IoT-Blockchain environment.

Blockchains as a communication mechanism

- Software components communicate through other components (communication elements)
- Communication elements:
  - Transfer data
  - Coordinate computation among components
- Blockchains perform these operations
- They have differences
- Blockchain use APIs to access historical transactions
- Alternatives:
  - A component monitors updates from new blocks
  - These data can be store in local DB or
  - Pass it to other components (via APIs)



Blockchains and asset management and control

- Tokenization can be used for asset management and control
- Tokens: represent physical or digital assets
- Tokens:
  - Fungible tokens (interchangeable)
  - NFTs (non – interchangeable)
- 2nd blockchain generations (e.g., Ethereum) → more flexible asset management and control

- **Tokenization as a process** starts when an asset under custody is represented using a token.
- The control of this token aligns with the ownership of the corresponding asset.
- The reverse process can take place if the user redeems the token to recover the asset.
- By using smart contracts, some conditions can be implemented and associated with the transfer of ownership.
- ERC20 → standard for Ethereum-based fungible tokens.
- ERC721 & ERC1155 → NFTs
- ERC20, 721 and 1155 describe the functions and events that token smart contracts should implement. Newly proposed tokens should follow the respective standard.

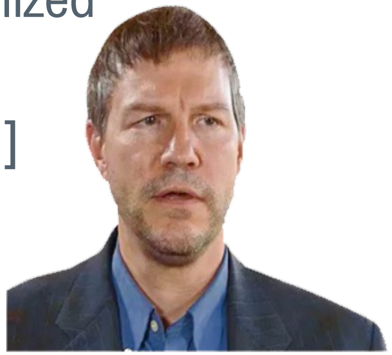
Smart Contracts

Smart contracts - definition

- Nick Szabo is a computer engineer and legal scholar. In 1996 Nick published a paper note on the concept of ***smart contracts*** [1].
- The idea was conceptualized in 2005, he conceptualized with a decentralized currency known as BitGold, a precursor to bitcoin.
- Smart contracts are self-executing, self-enforcing contracts

Definition

- "A smart contract is a self-executing contract with the terms of the agreement between buyer and seller being directly written into lines of code.
- The code and the agreements contained therein exist across a distributed, decentralized blockchain network.
- The code controls the execution, and transactions are trackable and irreversible." [2]



Nick Szabo

[1] Nick Szabo. Smart Contracts: Building Blocks for Digital Markets
[2] https://www.investopedia.com/terms/s/smart-contracts.asp

Smart contracts – key features

- self-verifying
- self-enforcing when the rules are met at all stages
- tamper-proof
- automate business processes
- ensure security
- No need for trusted intermediaries
- support multi-signature accounts to distribute funds (given that the terms of the contract are met)
- provide utility to other contracts (one contract can call another contract)

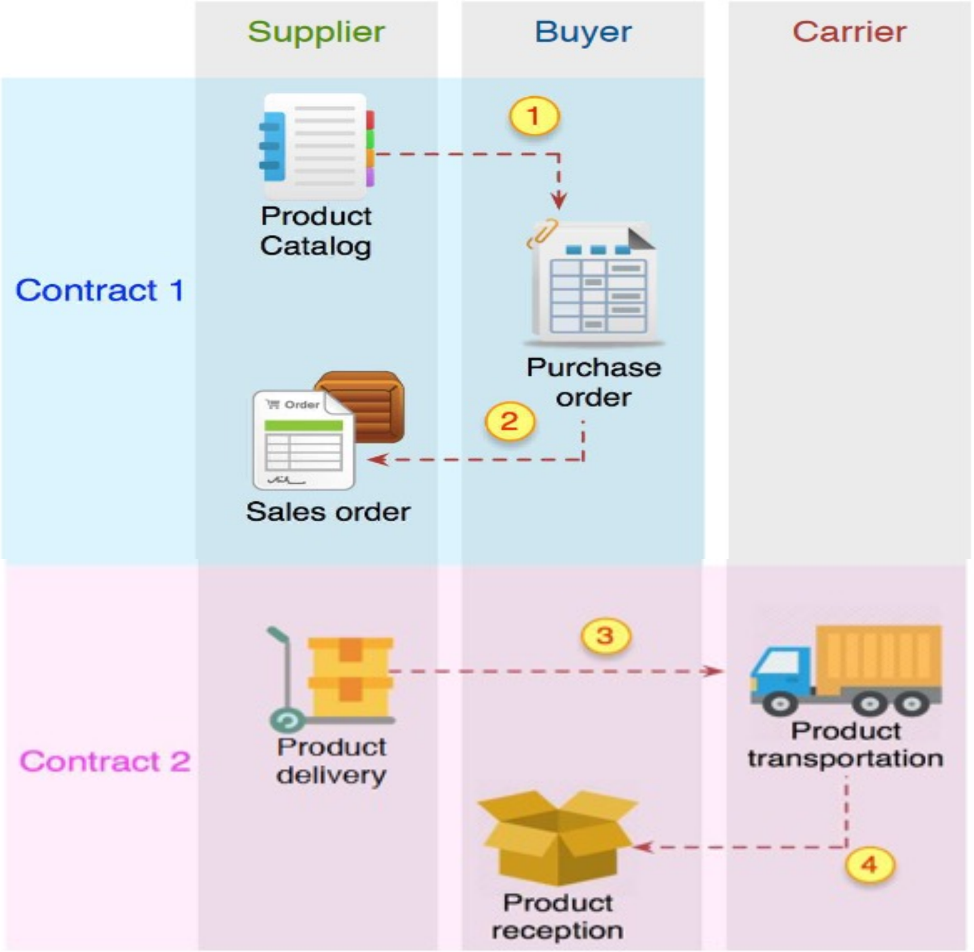
Smart contracts - benefits

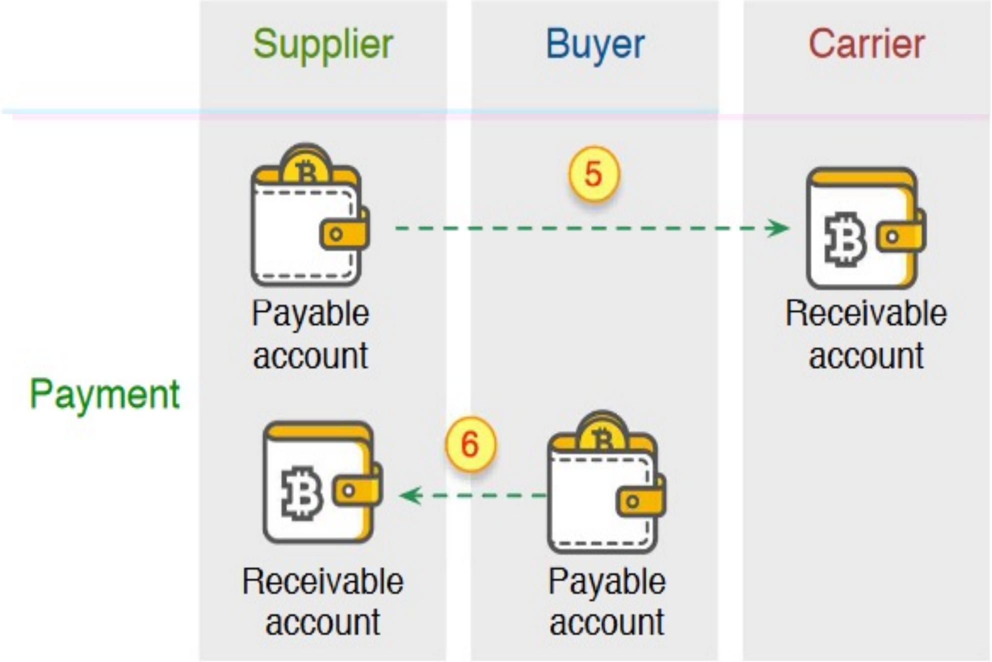
1. Accuracy
2. Clear communication
3. Cost savings
4. Efficiency
5. No disputes
6. Reduced risks
7. Security
8. Speed
9. Paper free
10. Transparency
11. Trust

Smart contract – popular platforms

- Agoric
- Algorand
- Ardor
- Avalanche
- Bitcoin
- Cardano
- Cosmos
- EOS
- Ergo
- Ethereum
- Fantom
- Hyperledger
- Matic Network
- Minter
- Neo
- Polkadot
- Qtum
- Ripple
- RSK
- Solana
- Stellar
- Telos
- Tezos
- TRON
- WAVES

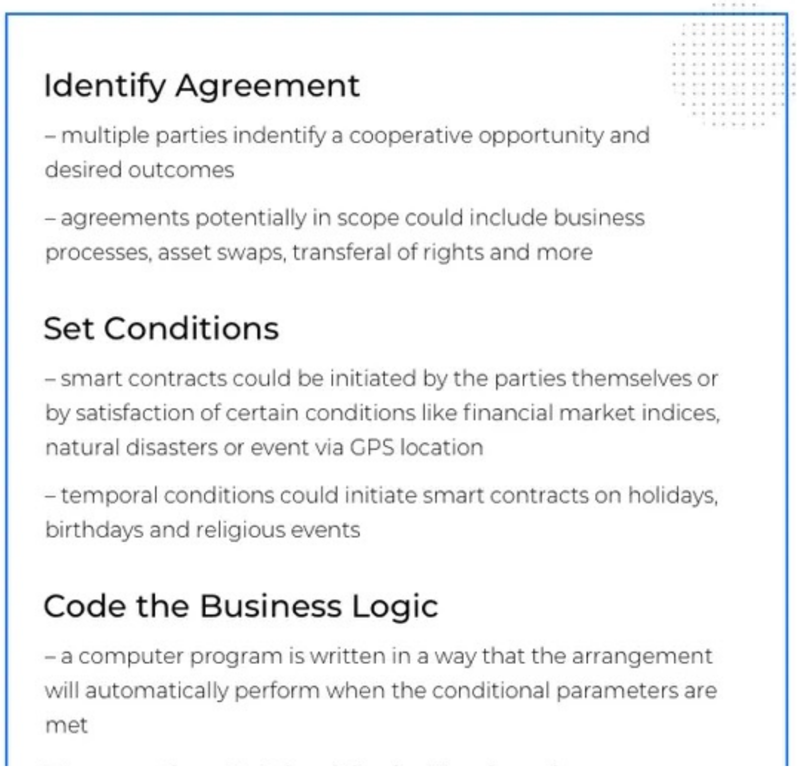
Smart contract - example





Source: https://arxiv.org/pdf/1912.10370.pdf

Smart contracts – in practice



Identify Agreement

– multiple parties indentify a cooperative opportunity and desired outcomes

– agreements potentially in scope could include business processes, asset swaps, transferal of rights and more

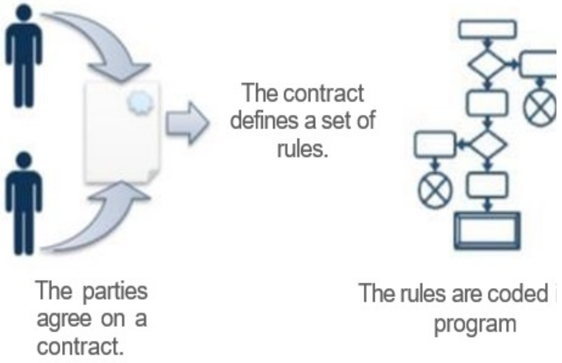
Set Conditions

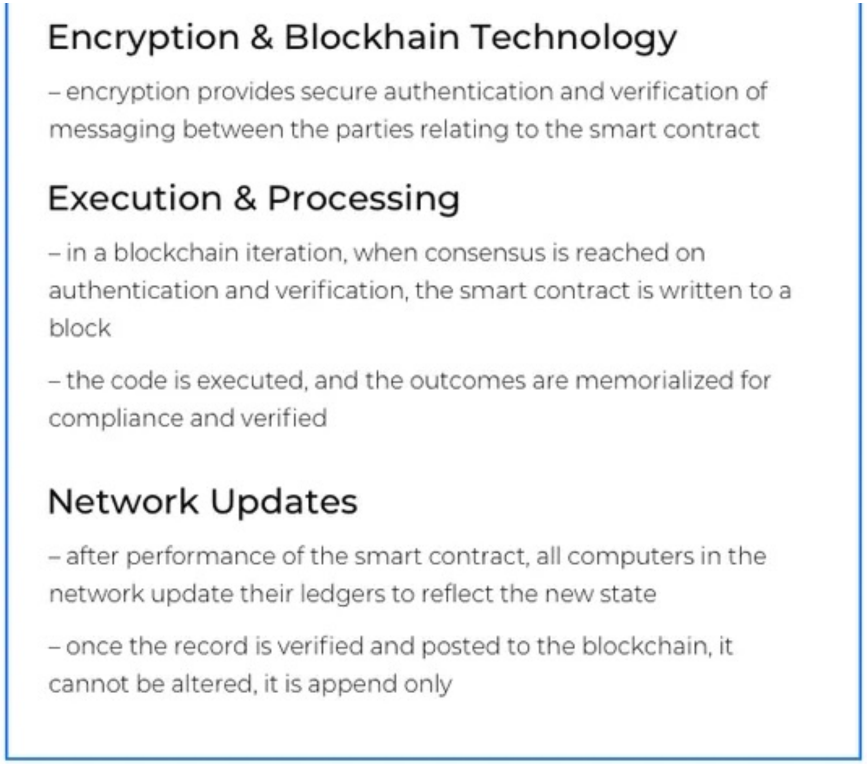
– smart contracts could be initiated by the parties themselves or by satisfaction of certain conditions like financial market indices, natural disasters or event via GPS location

– temporal conditions could initiate smart contracts on holidays, birthdays and religious events

Code the Business Logic

– a computer program is written in a way that the arrangement will automatically perform when the conditional parameters are met


The contract defines a set of rules.
The parties agree on a contract.
The rules are coded i
program



Encryption & Blockhain Technology

– encryption provides secure authentication and verification of messaging between the parties relating to the smart contract

Execution & Processing

– in a blockchain iteration, when consensus is reached on authentication and verification, the smart contract is written to a block

– the code is executed, and the outcomes are memorialized for compliance and verified

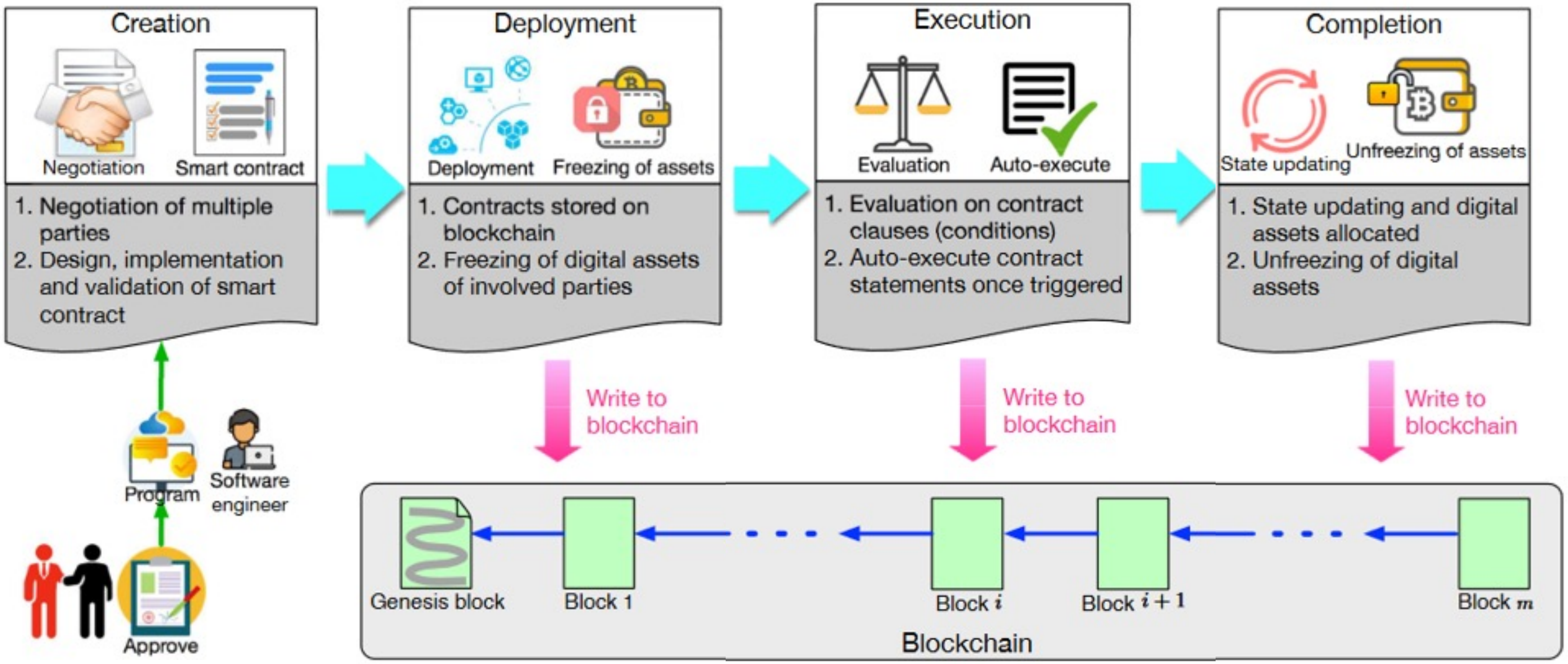
Network Updates

– after performance of the smart contract, all computers in the network update their ledgers to reflect the new state

– once the record is verified and posted to the blockchain, it cannot be altered, it is append only

Adapted from: the-blockchain.com

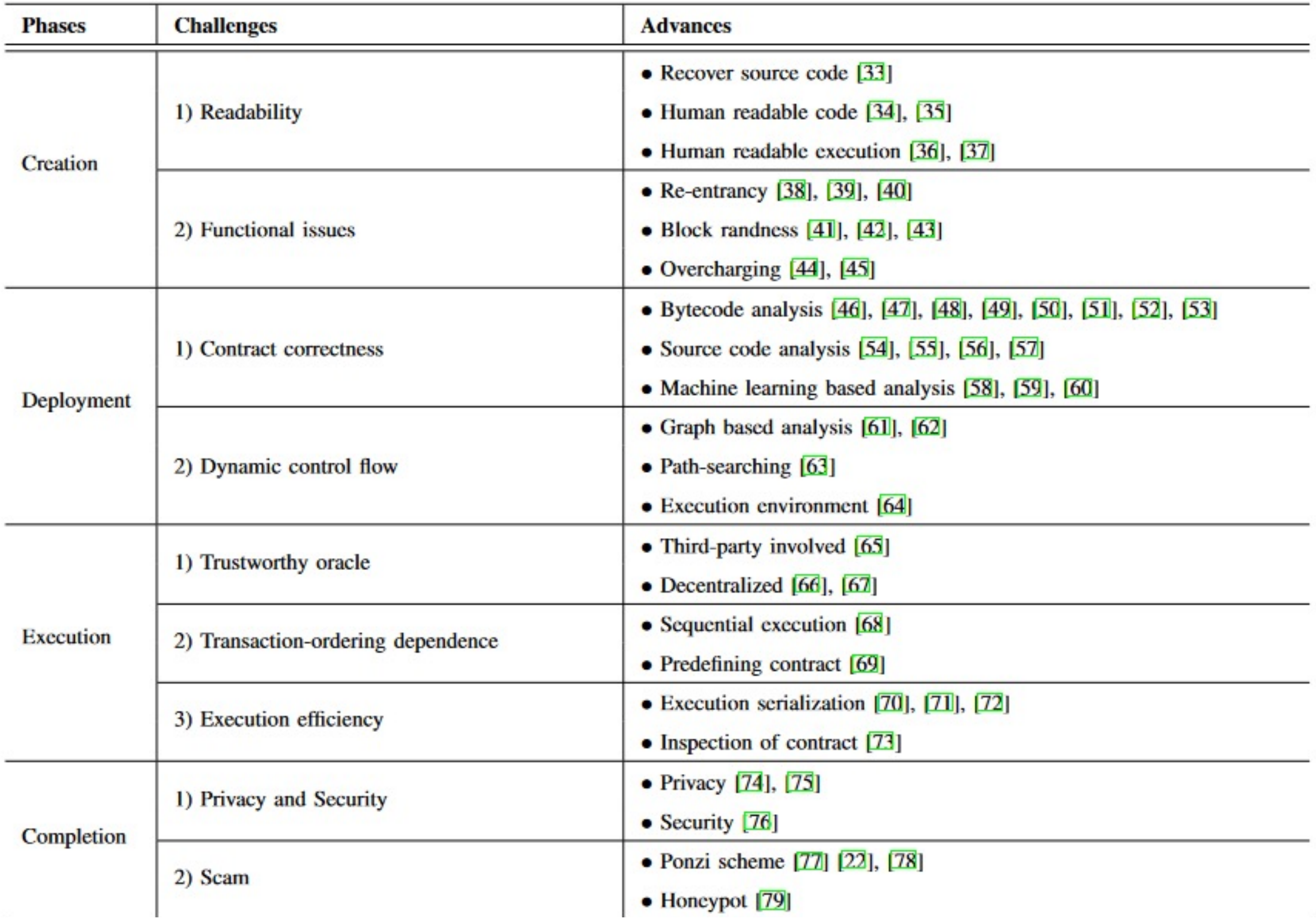
Smart contract - Lifecycle



Creation
Negotiation
Smart contract
1. Negotiation of multiple parties
2. Design, implementation and validation of smart contract
Deployment
Deployment
Freezing of assets
1. Contracts stored on blockchain
2. Freezing of digital assets of involved parties
Execution
Evaluation
Auto-execute
1. Evaluation on contract clauses (conditions)
2. Auto-execute contract statements once triggered
Completion
State updating
Unfreezing of assets
1. State updating and digital assets allocated
2. Unfreezing of digital assets
Program
Software engineer
Approve
Write to blockchain
Write to blockchain
Write to blockchain
Genesis block
Block 1
Block i
Block $i+1$
Block m
Blockchain

Source: https://arxiv.org/pdf/1912.10370.pdf

Smart contract - challenges

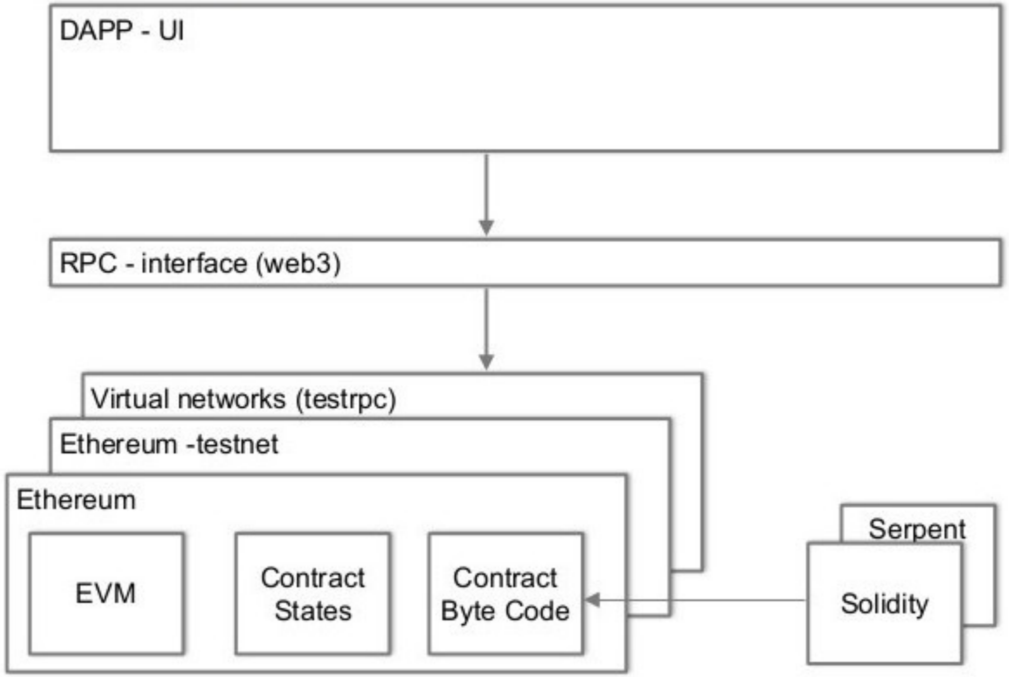


Phases	Challenges	Advances
Creation	1) Readability	• Recover source code [33] • Human readable code [34], [35] • Human readable execution [36], [37]
	2) Functional issues	• Re-entrancy [38], [39], [40] • Block randness [41], [42], [43] • Overcharging [44], [45]
Deployment	1) Contract correctness	• Bytecode analysis [46], [47], [48], [49], [50], [51], [52], [53] • Source code analysis [54], [55], [56], [57] • Machine learning based analysis [58], [59], [60]
	2) Dynamic control flow	• Graph based analysis [61], [62] • Path-searching [63] • Execution environment [64]
Execution	1) Trustworthy oracle	• Third-party involved [65] • Decentralized [66], [67]
	2) Transaction-ordering dependence	• Sequential execution [68] • Predefining contract [69]
	3) Execution efficiency	• Execution serialization [70], [71], [72] • Inspection of contract [73]
Completion	1) Privacy and Security	• Privacy [74], [75] • Security [76]
	2) Scam	• Ponzi scheme [77] [22], [78] • Honeypot [79]

Source: https://arxiv.org/pdf/1912.10370.pdf

Smart contracts and Decentralised Applications (Dapps)

- Applications that use smart contract(s)
- Provides a user-friendly interface to smart contracts
- Main components:
  - smart contract
  - files for web user interface front-end
  - files for web user interface back-end



..Ok, so what is a dApp?

A software project that is composed of one or more smart contracts and optionally interacts with a Web application. This combination of a Web front-end and a back-end that is fuelled by a decentralized peer-to-peer protocol (e.g., a Blockchain) results in a decentralized application, or simply a dApp.

What does it take for an application to qualify as a dApp?

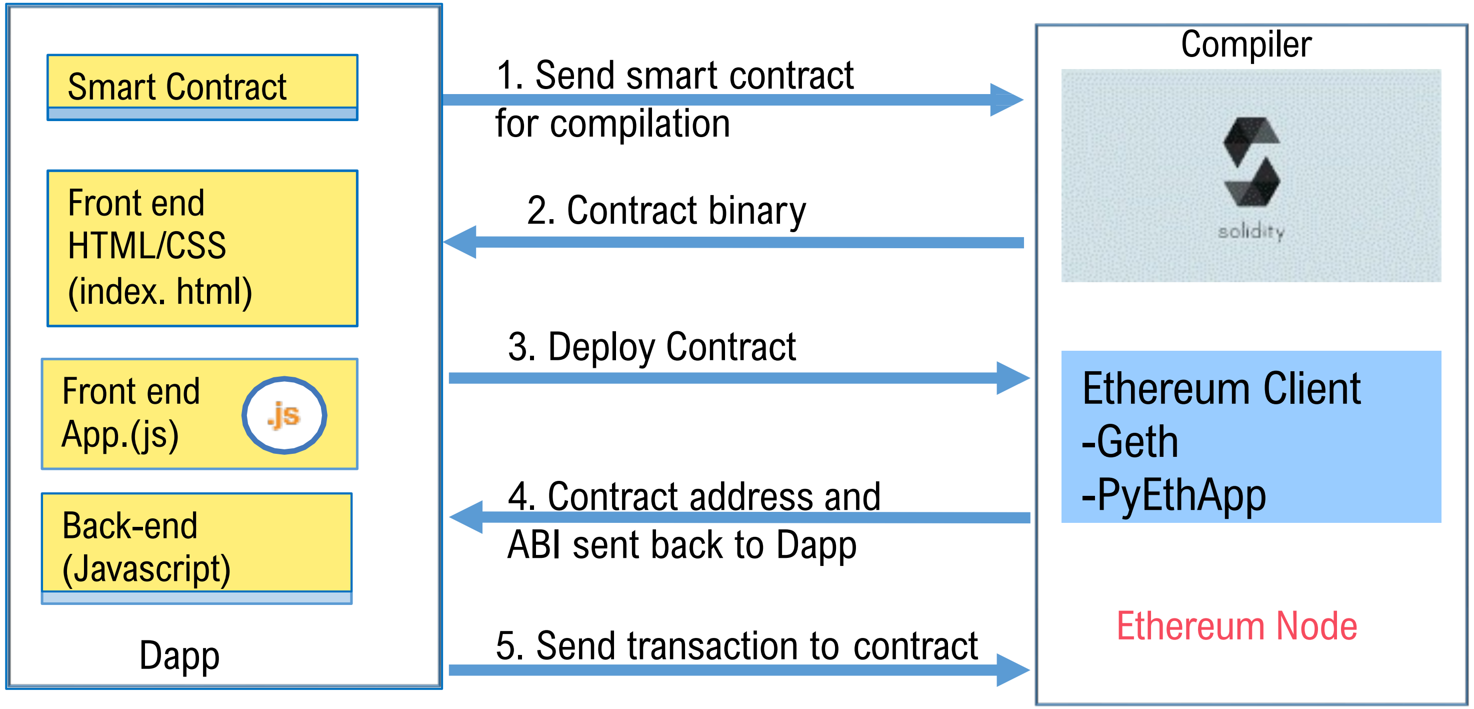
The following criteria characterize the degree on whether a software application is classified as a dApp:

- **Open-source:** Source code of the application needs to be open to the public. Any changes to the source code are proposed and applied based on consensus from the user base or ecosystem that supports the development of the application.
- **Internal Digital Currency:** The application should embed its own incentivization logic with the use of a native token. The token is used to incentivize usage, further development of the application and to encourage network participation.
- **Decentralized Consensus:** The application must be powered by a decentralized agreement mechanism. This will allow the application to reach agreement on the current state of the world.
- **Smart Contracts:** The application runs on a decentralized protocol that executes algorithmic code with the use of smart contracts. The smart contracts are executed and validated on every node upon persisting the deployment block. Once the smart contracts are deployed these are immutable.
- **No Central Point of Failure:** dApps should be able to run across a network of independent nodes without any disruption (even in the case where nodes experience failures). The application's data and records of operation must be stored in a public decentralized peer-to-peer network. In other words, must be fully decentralized with no central point of failure.

Smart contracts - Dapps creation

Main steps for creation of Dapps

- Develop smart contract in a high-level language
- Compile the contracts
- Deploy the contracts on a Blockchain network (e.g., for Ethereum we use Ethereum clients)
- Integrate smart contracts with web applications



Ethereum Virtual Machine (EVM)

EVM

- Runtime environment for smart contracts in Ethereum
- Completely isolated environment → code running inside the EVM has no access to
  - network,
  - filesystem
  - other processes.
- Smart contracts have limited access to other smart contracts
- Processes and keeps track of all transactions, blocks and smart contract results

Image source: https://blockgeeks.com/guides/ethereum-gas

EVM – accounts

Accounts:

- uniquely identified by their address (20 bytes)
- Two categories of accounts
    - External accounts (usually refer as users
    - Internal accounts (contracts)
- Users and contracts are treated equally by EVM
- Users and contracts have a balance in Wei (1 ETH = 10^{18} Wei (1 quintillion))
- Balance can be modified through transactions

EVM – accounts

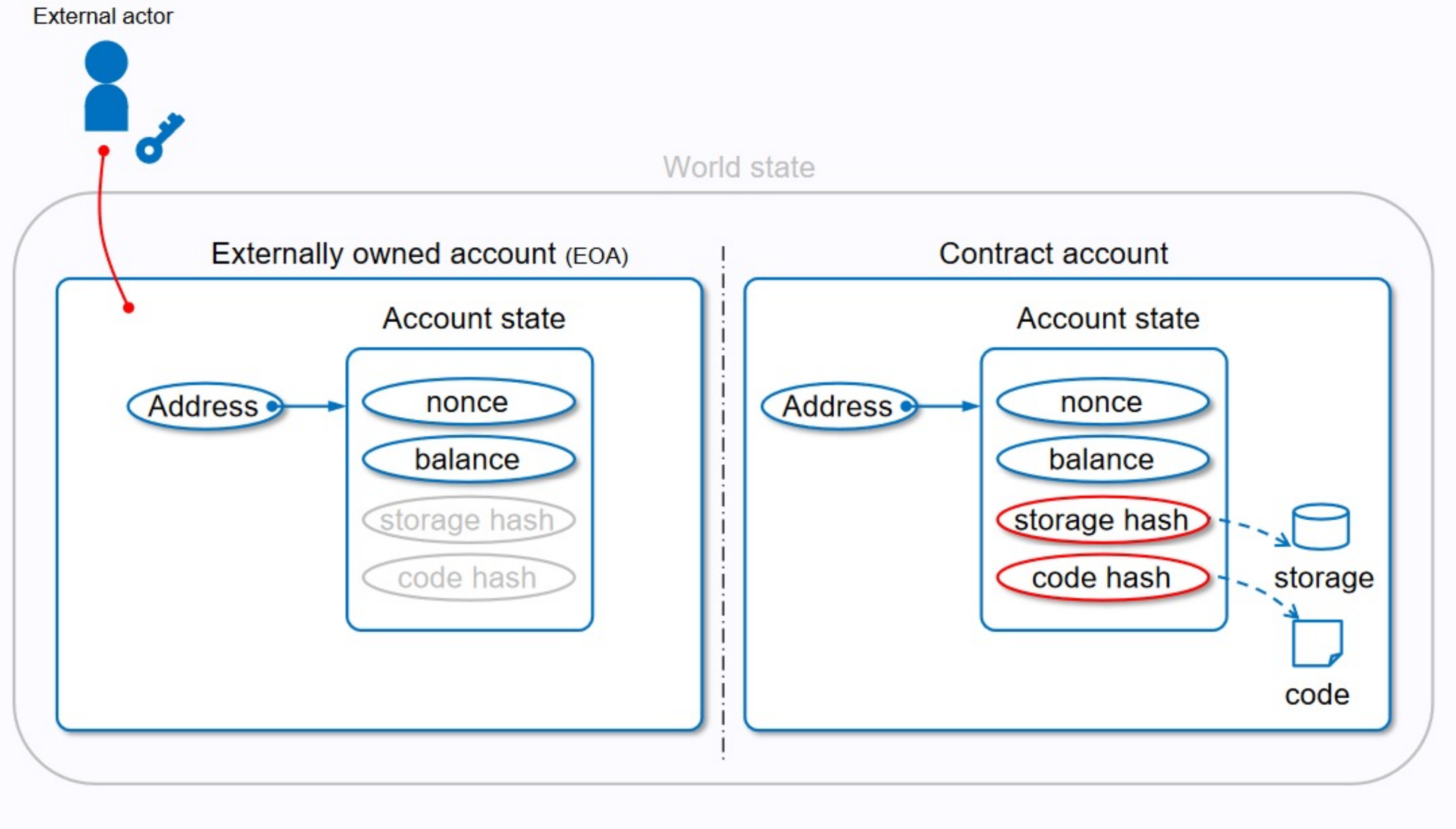
Externally-owned (User) account:

- Has an ether balance
- No associated code
- Send transactions (ether transfers or contract code trigger)
- Used to facilitate payments between users
- Controlled by public-private keys (e.g. humans)

Contract account:

- Has an ether balance
- Has associated code
- Code execution trigged by transactions or messages (calls) received from other contracts
- On execution it performs operations of arbitrary complexity
- controlled by code stored together with the account

EVM – accounts



EOA is controlled by a private key.
EOA cannot contain EVM code.

Contract contains EVM code.
Contract is controlled by EVM code.

https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

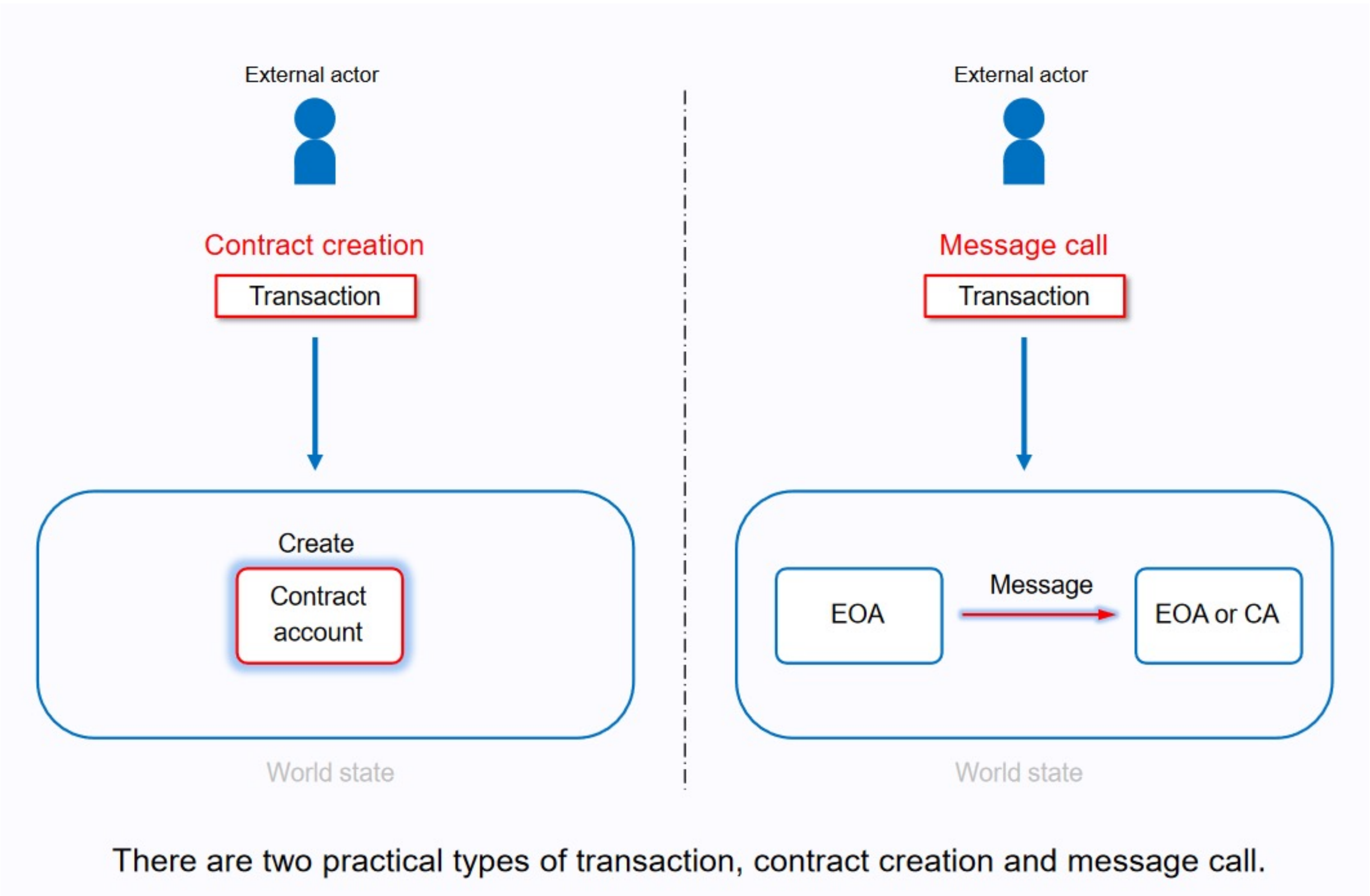
EVM – transactions

Transactions:

- Transactions are signed data packages
- In simple words: they are messages sent from one account to another
- Users can issue transactions
- Recorded on blockchain
- Transactions can
  - Transfer Ether to users and contracts
  - Call a contract
  - Create a contract
- Consists of:
  - payload (binary data)
  - Ether

- Case 1: The target account includes code:
  - The code is executed, and the payload is used as input
- Case 2: The target account is null (not set)
  - Step1: create a new contract
  - Step 2: define its address. The address is given by the sender account and its number of transactions sent
  - Step 3: the payload is executed
  - Step 4: the output is permanently stored as the code of the new contract

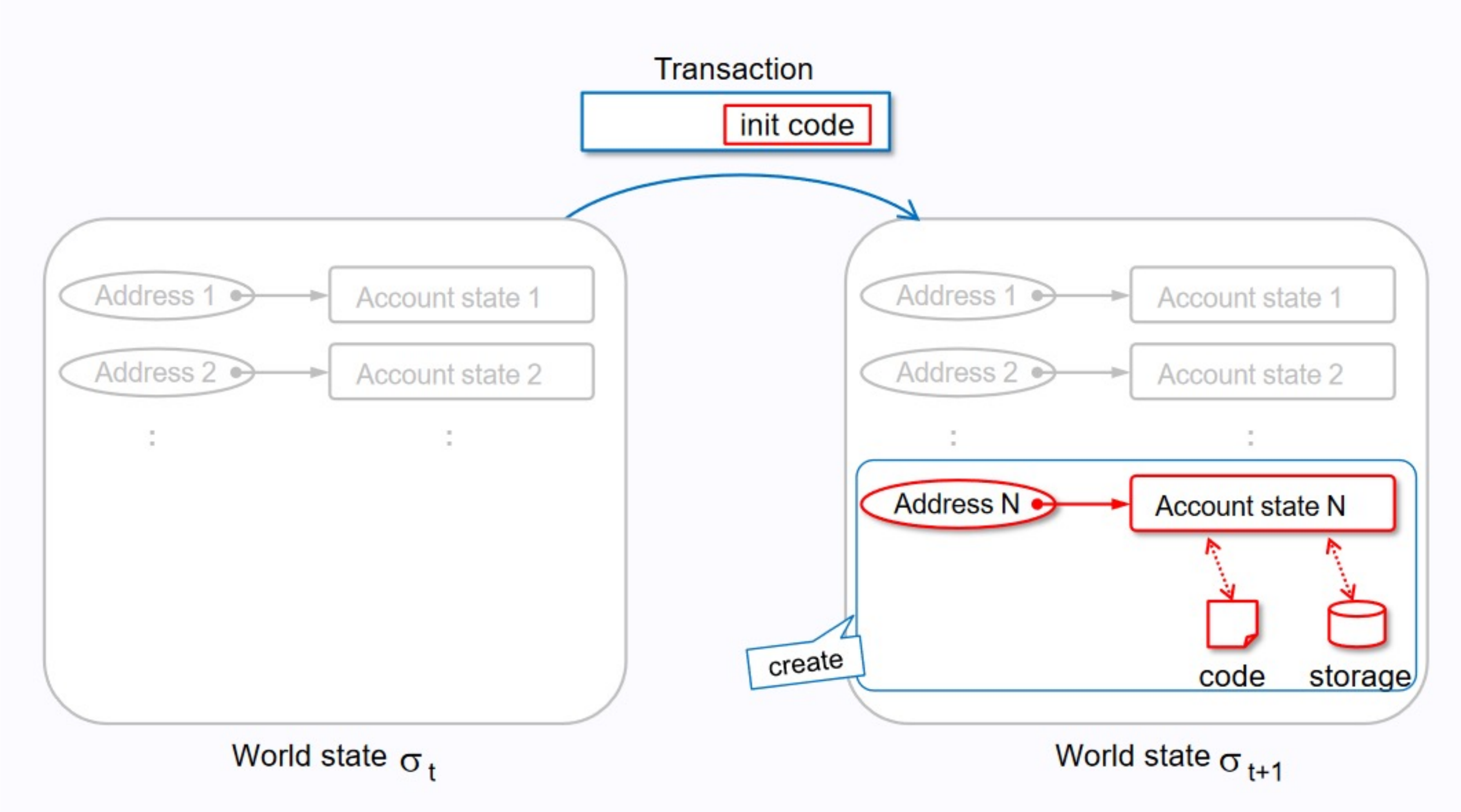
EVM – transactions



There are two practical types of transaction, contract creation and message call.

https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

EVM – transactions – contract creation



https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

EVM – contracts and messages

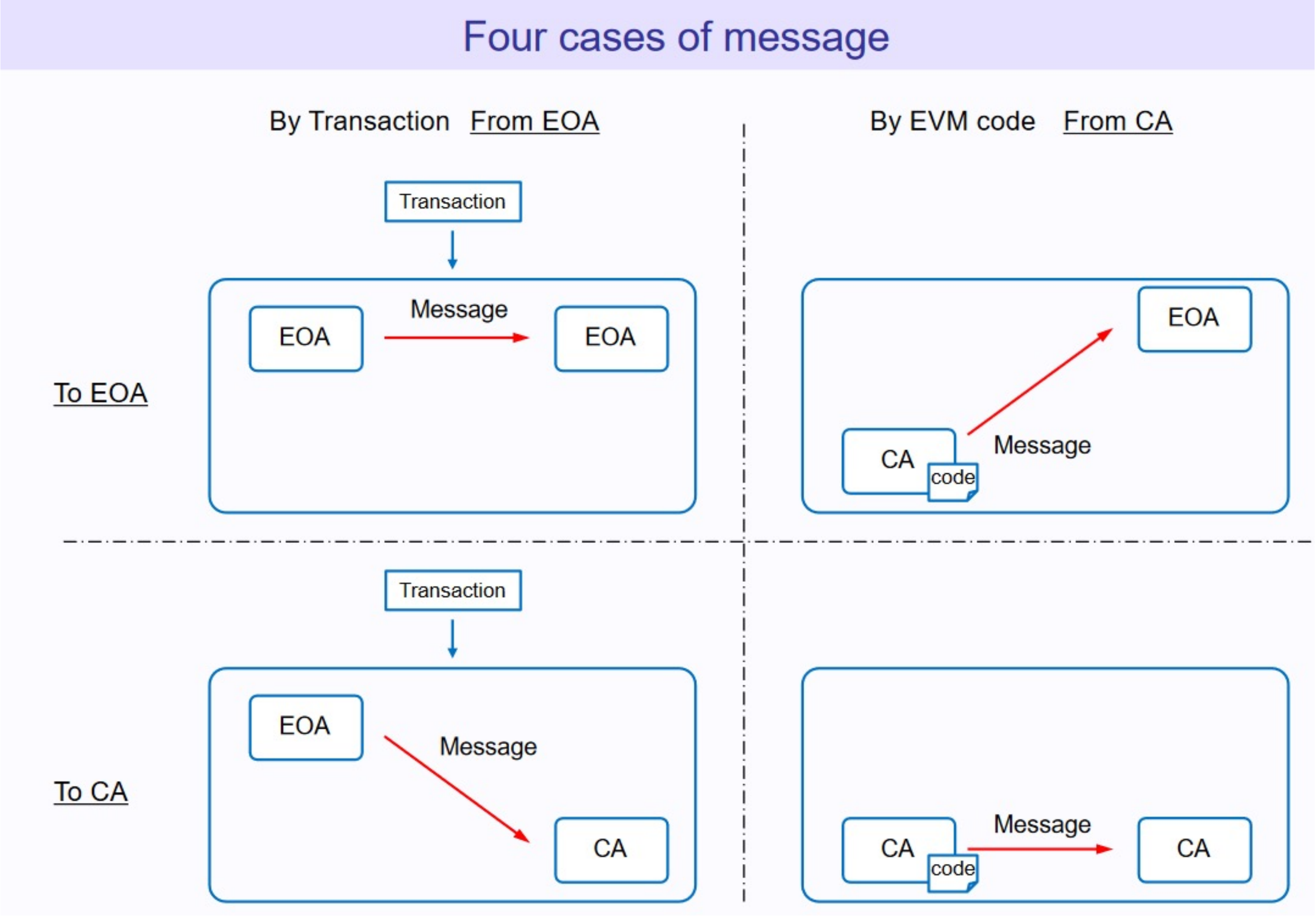
Contracts:

- Need to be triggered to become active
- Contracts activation
  - A user issues a transaction
  - A contract calls another contract

Messages:

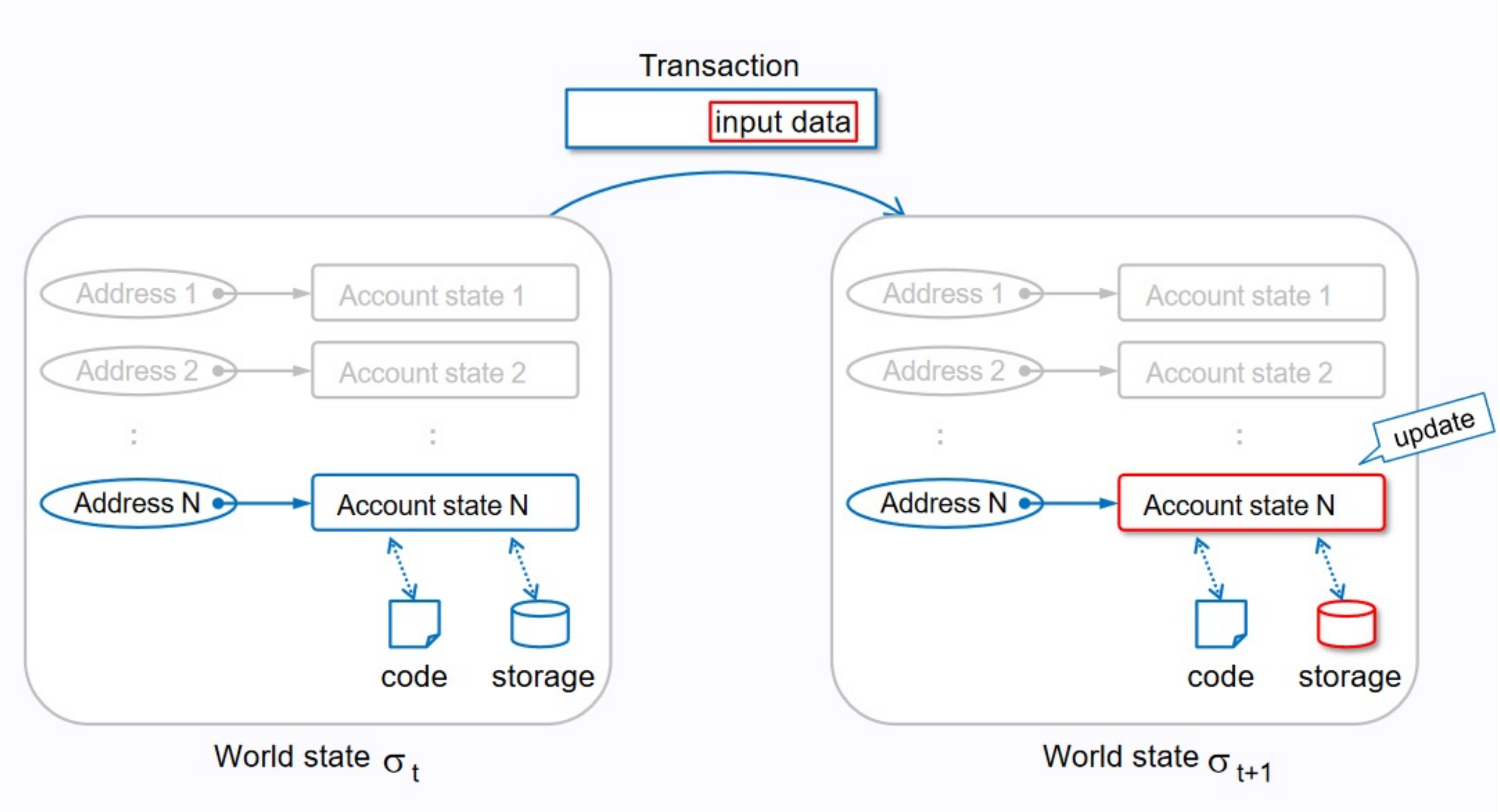
- Message is the call from one contract to another
- Not recorded on the blockchain
- Exist only in the execution environment (EVM)

EVM – message call



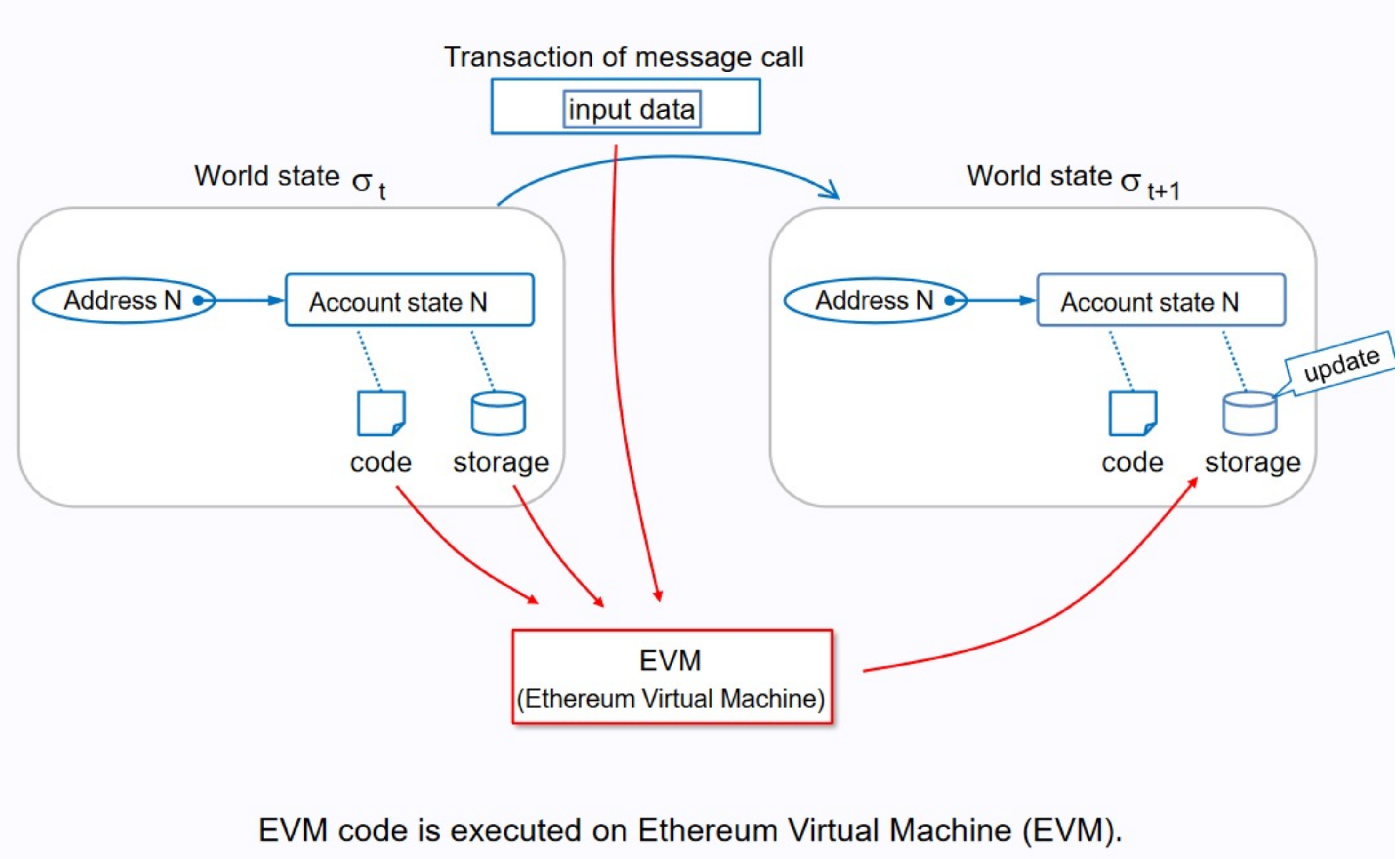
https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

EVM – message call



https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

EVM

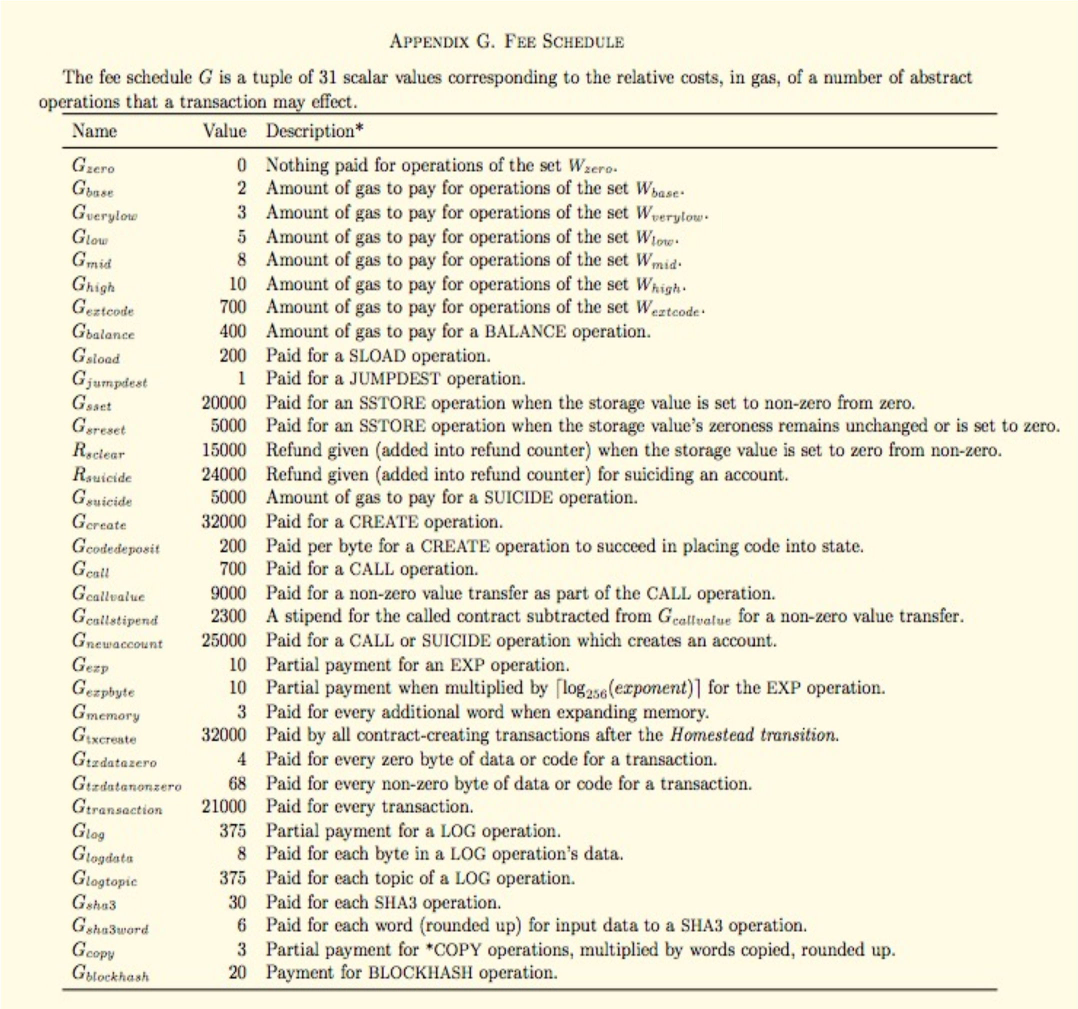


EVM code is executed on Ethereum Virtual Machine (EVM).

https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

EVM - gas

- Different kinds of Ethereum transactions exist, which add to its complexity
- "Gas" exists to address this complexity
- Imposed to award miners for the operations they are required to performed
- Different operations will have different gas costs
- Miners can't execute the moment the gas runs out
- Any gas left over, immediately refunded to the operation generator



Appendix G. Fee Schedule

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{suicide}$	24000	Refund given (added into refund counter) for suiciding an account.
$G_{suicide}$	5000	Amount of gas to pay for a SUICIDE operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SUICIDE operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	10	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the *Homestead transition*.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.

Image source: https://blockgeeks.com/guides/ethereum-gas

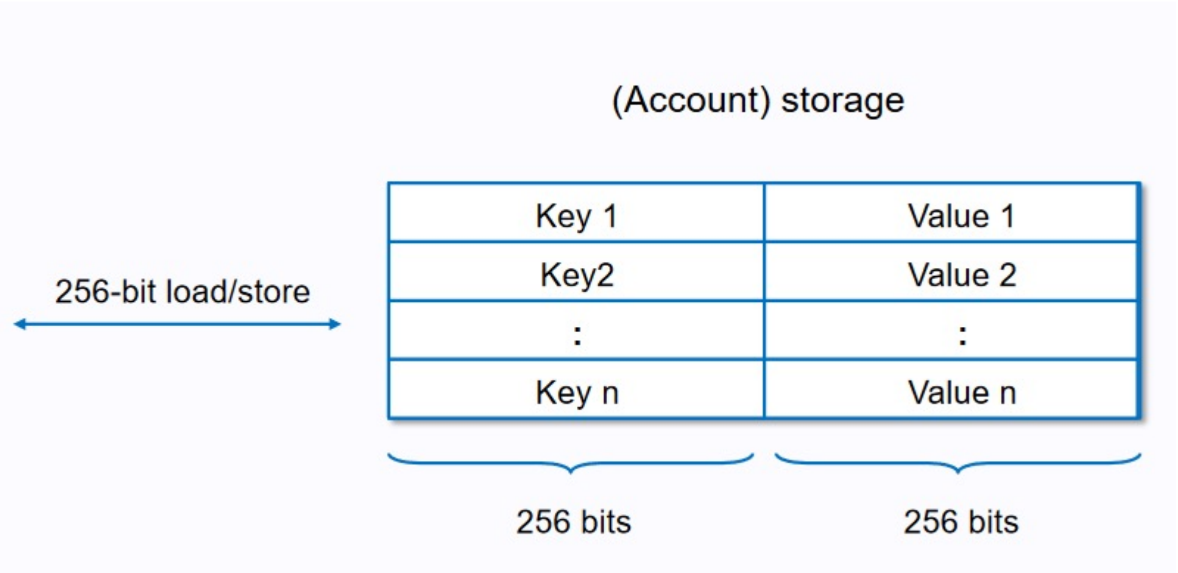
EVM – gas

- Gas: is the fee paid to miners for providing computational power to execute transactions.
- Examples
  - Move ETH between different accounts
  - Exchange ETH to other tokens
  - Mint an NFT
- Gas is used for security purposes too (e.g. prevents spamming)

- Gas price changes all times.
- Depends on
  - The busier the network the higher the fee it is
  - Not all transactions have the same fee. The simpler the transaction, the lower the fee
- London Hard Fork and EIP-1559 (August 2021) makes gas fees:
  - more transparent
  - Predictable
  - do not reduce gas fees

EVM – storage, memory and the stack

- EVM stores data in:
  - Storage
  - Memory
  - Stack



(Account) storage

Key 1	Value 1
Key2	Value 2
:	:
Key n	Value n

Storage is a key-value store

- Maps 256-bit words to 256-bit words
- Persistent among
  - function calls
  - transactions
- Each account has a storage area
- Contracts can only read and write to its own storage
- Costly to read and modify storage → minimize what is stored to what the contract needs to run.
- Data like derived calculations, aggregates etc should be stored outside of the contract.

https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

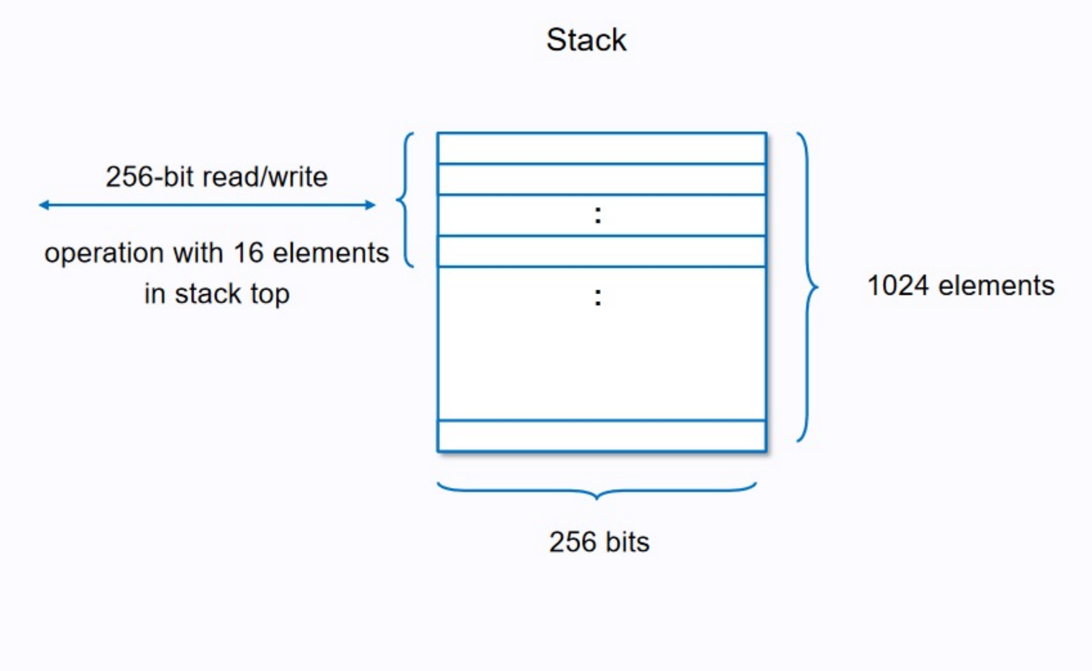
EVM – storage, memory and the stack

Stack:

- The EVM is a stack machine
- → All computations are performed on the **stack**.
- Maximum size: 1024 elements
- Access to the stack is limited to the top end
- Change the position of the elements (move up-down) to perform computations
- Stack elements can be moved to storage or memory to get deeper access to the stack

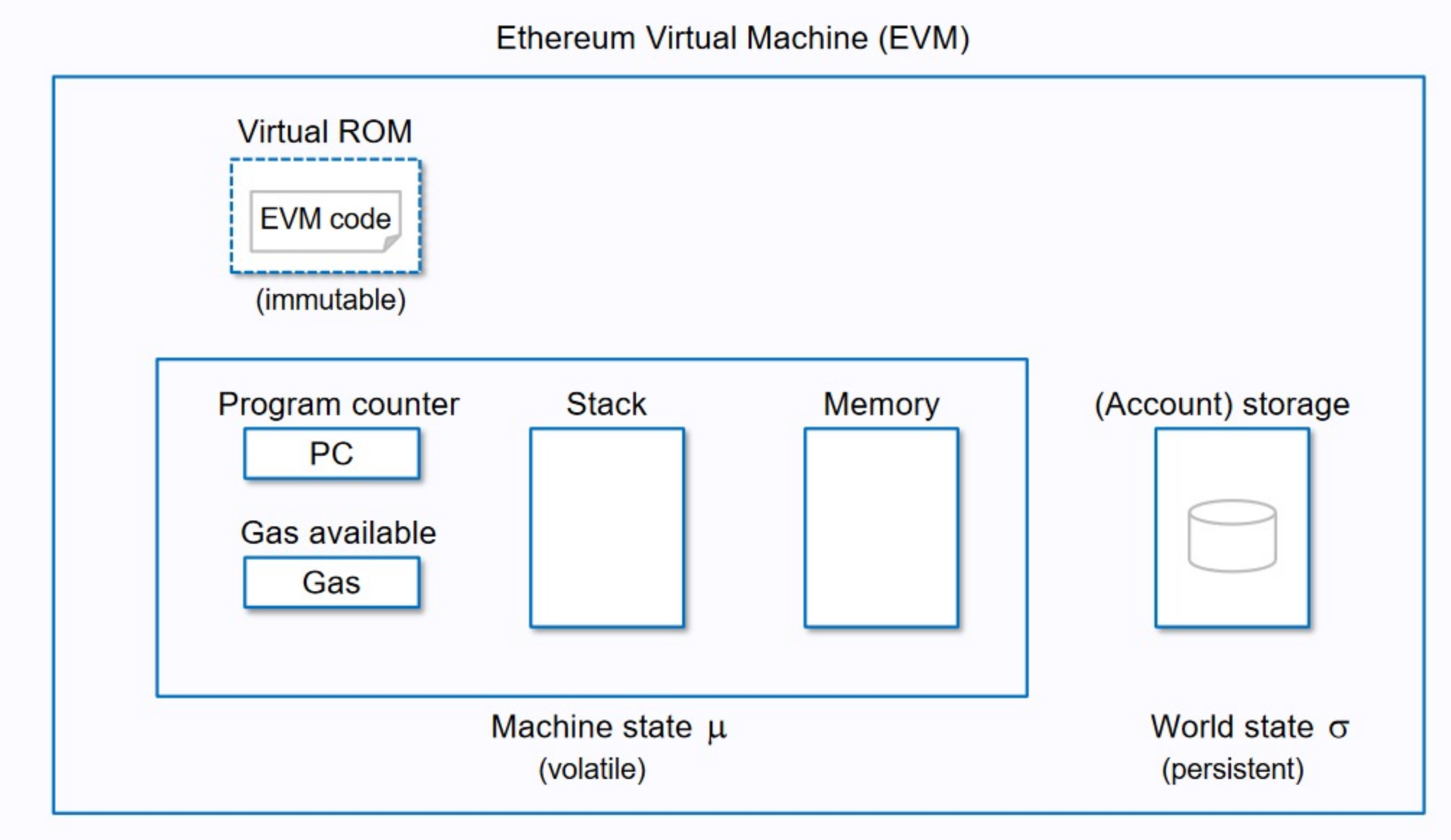
Memory:

- Provides a cleared instance for message calls
- The larger it grows the more costly it is



All operation are performed on the stack.

EVM – architecture



https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf

WebAssembly (WASM)

WebAssembly - background

Initiated by Moz://a research

Problems with existing browsers

Java script: Great language but did not designed to:

- be fast
- Start up a large application

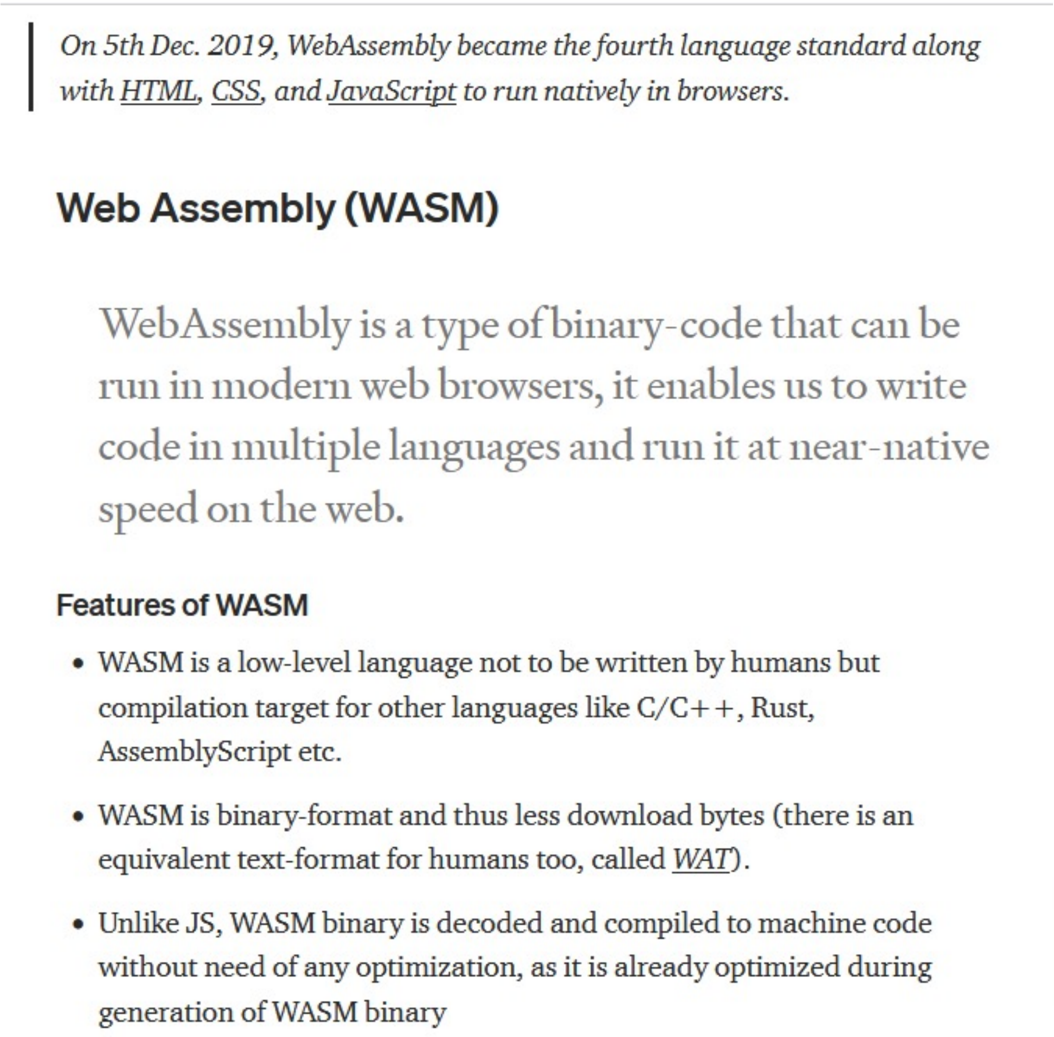
WASM is built as binary format

- Compact to download
- Very efficient to compile and execute
- Speed in 2 areas
  - Startup speed (e.g. 2+ times faster)
  - Allows huge applications to load quickly

Benefits

- Fast
- Efficient
- Portable
- Secure
- Increased throughput
- High performance execution without needing to use any plug-ins
- More control over memory management
- Smooth performance

WebAssembly



> *On 5th Dec. 2019, WebAssembly became the fourth language standard along with HTML, CSS, and JavaScript to run natively in browsers.*

Web Assembly (WASM)

WebAssembly is a type of binary-code that can be run in modern web browsers, it enables us to write code in multiple languages and run it at near-native speed on the web.

Features of WASM

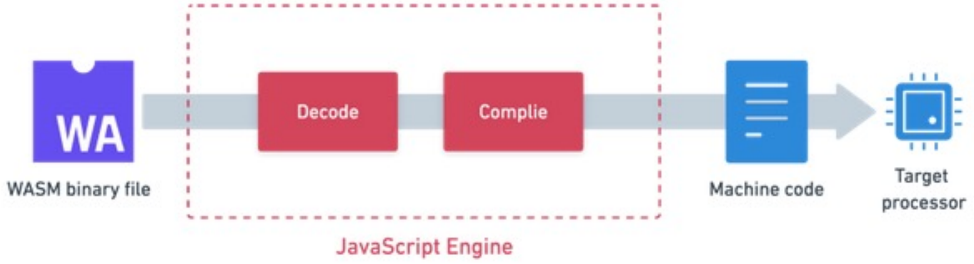
- WASM is a low-level language not to be written by humans but compilation target for other languages like C/C++, Rust, AssemblyScript etc.
- WASM is binary-format and thus less download bytes (there is an equivalent text-format for humans too, called *WAT*).
- Unlike JS, WASM binary is decoded and compiled to machine code without need of any optimization, as it is already optimized during generation of WASM binary

https://medium.com/front-end-weekly/webassembly-why-and-how-to-use-it-2a4f95c8148f

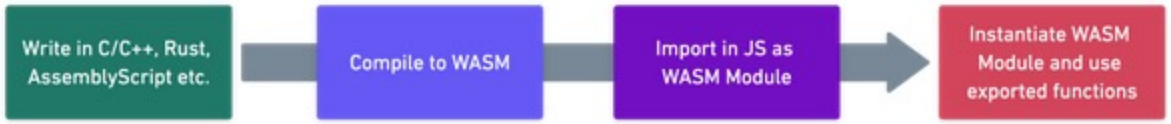
"**WebAssembly** is a binary instruction format for a stack-based virtual machine.

Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications."

https://webassembly.org/https://webassembly.org/

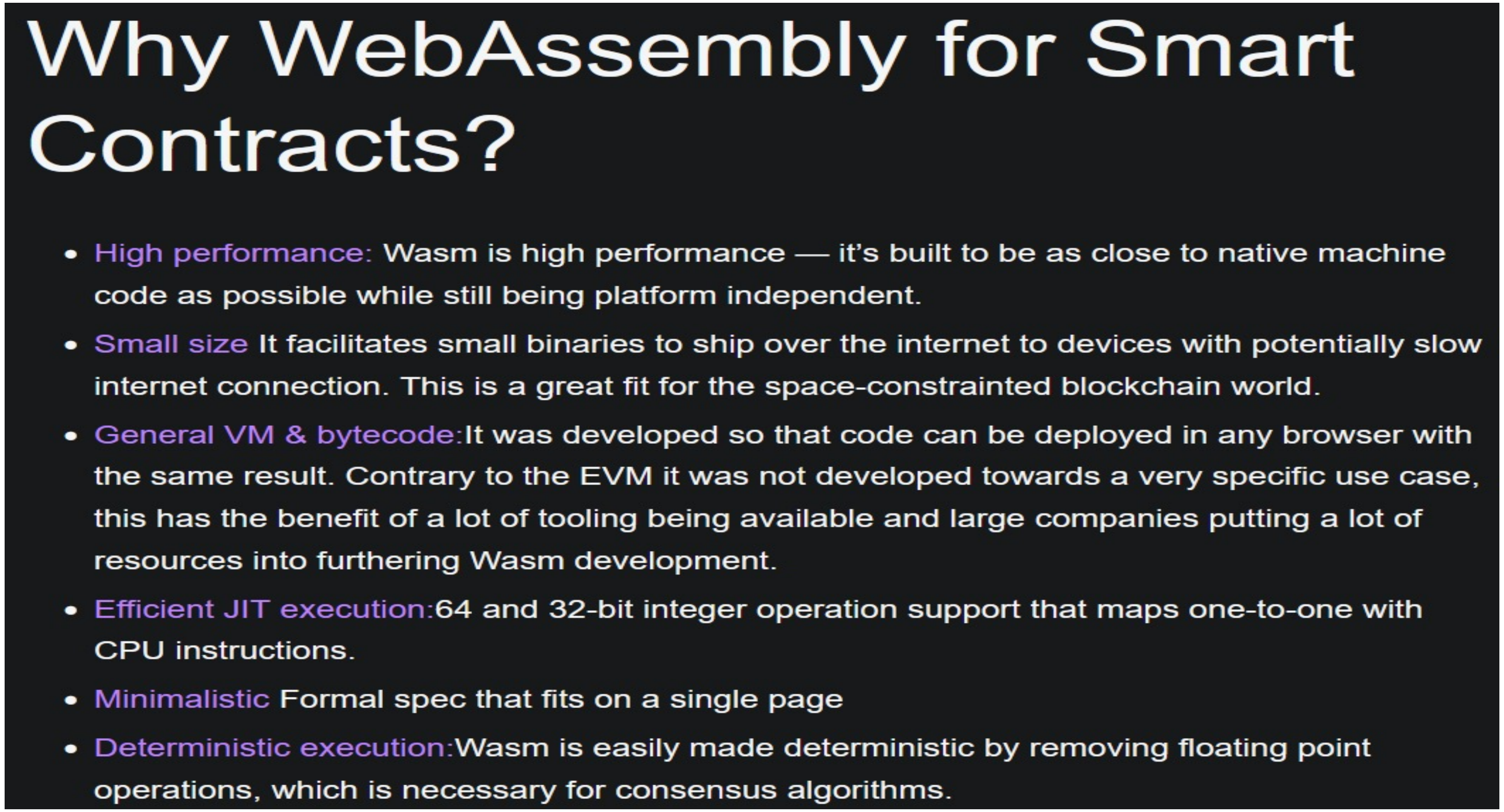


WebAssembly execution pipeline


Write in C/C++, Rust, AssemblyScript etc.
Compile to WASM
Import in JS as WASM Module
Instantiate WASM Module and use exported functions

Overview of generating and consuming WASM

WebAssembly and smart contracts

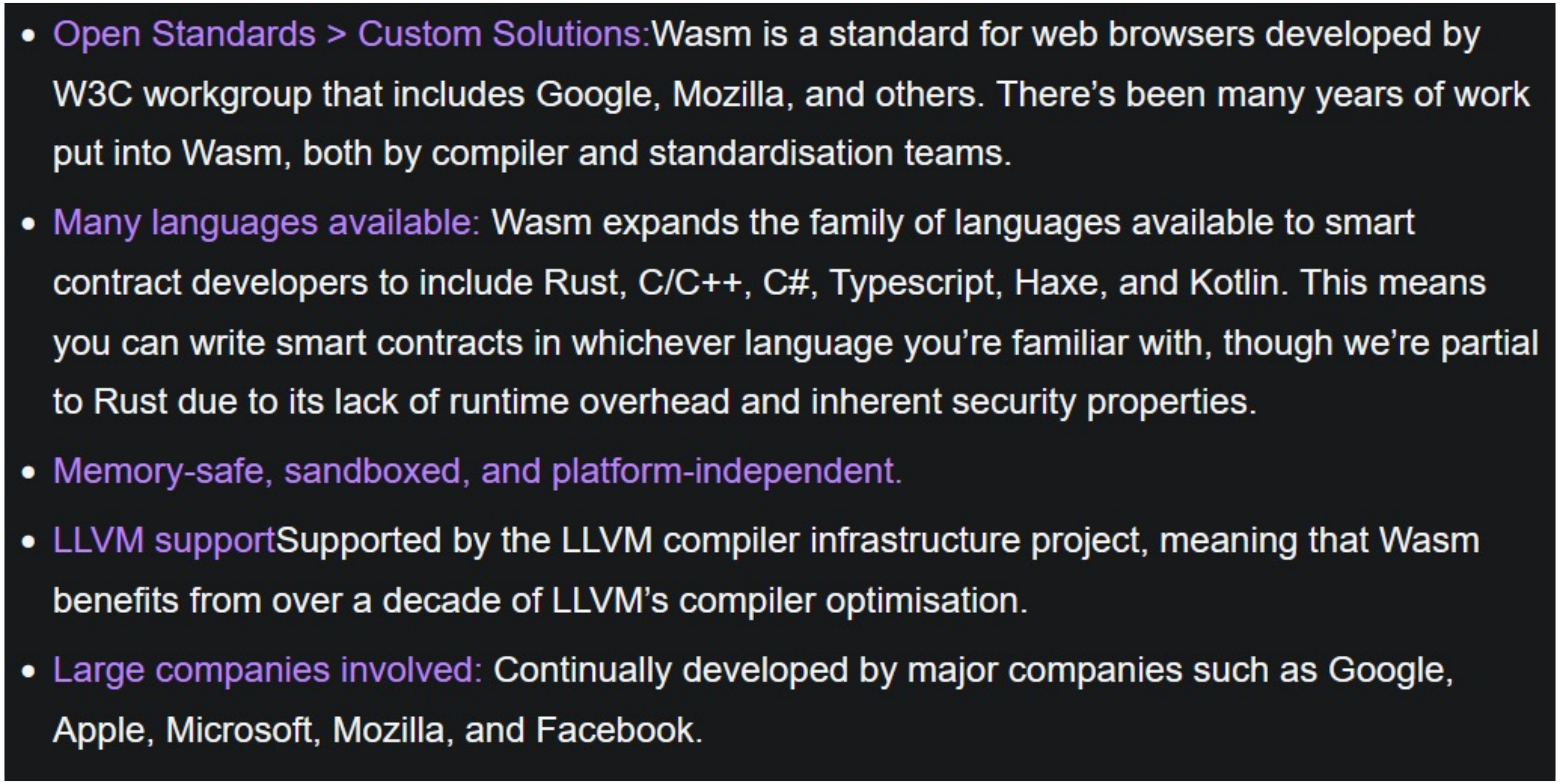


Why WebAssembly for Smart Contracts?

- High performance: Wasm is high performance — it's built to be as close to native machine code as possible while still being platform independent.
- Small size It facilitates small binaries to ship over the internet to devices with potentially slow internet connection. This is a great fit for the space-constrainted blockchain world.
- General VM & bytecode:It was developed so that code can be deployed in any browser with the same result. Contrary to the EVM it was not developed towards a very specific use case, this has the benefit of a lot of tooling being available and large companies putting a lot of resources into furthering Wasm development.
- Efficient JIT execution:64 and 32-bit integer operation support that maps one-to-one with CPU instructions.
- Minimalistic Formal spec that fits on a single page
- Deterministic execution:Wasm is easily made deterministic by removing floating point operations, which is necessary for consensus algorithms.

https://paritytech.github.io/ink-docs/why-webassembly-for-smart-contracts/

WebAssembly and smart contracts



- Open Standards > Custom Solutions:Wasm is a standard for web browsers developed by W3C workgroup that includes Google, Mozilla, and others. There's been many years of work put into Wasm, both by compiler and standardisation teams.
- Many languages available: Wasm expands the family of languages available to smart contract developers to include Rust, C/C++, C#, Typescript, Haxe, and Kotlin. This means you can write smart contracts in whichever language you're familiar with, though we're partial to Rust due to its lack of runtime overhead and inherent security properties.
- Memory-safe, sandboxed, and platform-independent.
- LLVM supportSupported by the LLVM compiler infrastructure project, meaning that Wasm benefits from over a decade of LLVM's compiler optimisation.
- Large companies involved: Continually developed by major companies such as Google, Apple, Microsoft, Mozilla, and Facebook.

https://paritytech.github.io/ink-docs/why-webassembly-for-smart-contracts/

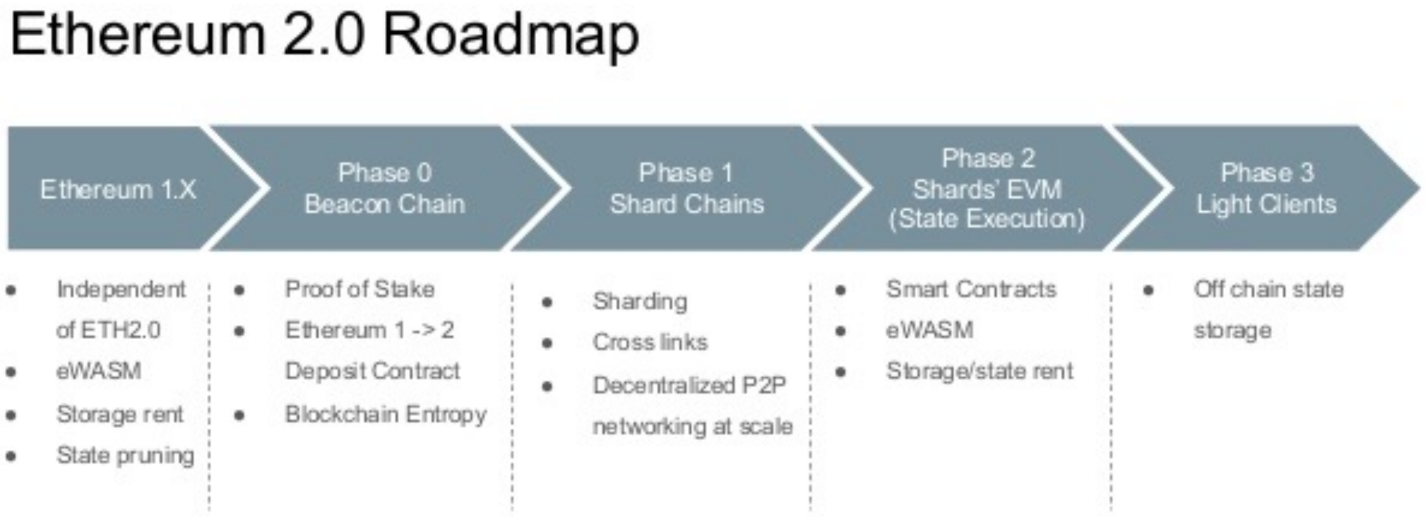
eWASM

Ethereum 2.0 and eWasm

- **Ethereum 2.0**
- A series of upgrades that will radically change the protocol
- → more scalable
- → more efficient
- Ethereum 2.0 upgrades include
  - Proof-of-stake with Casper Protocol,
  - Sharding,
  - Raiden (off-chain scaling solution, enabling near-instant, low-fee and scalable payments)
  - Plasma (separate blockchain anchored to *Ethereum* chain, and uses fraud proofs to arbitrate disputes)
  - Rollups

Ethereum 2.0 and eWasm

- These changes will be implemented in different Ethereum phases to ensure proper deployment and execution.
- Phase 0: Beacon chain to initiate a transition to proof-of-stake (POS).
- Phase 1: Initiate sharding.
- Phase 2: Transition from the Ethereum Virtual Machine (EVM) to eWASM Ethereum 2.0



Ethereum 2.0 Roadmap

Ethereum 1.X	Phase 0 Beacon Chain	Phase 1 Shard Chains	Phase 2 Shards' EVM (State Execution)	Phase 3 Light Clients
• Independent of ETH2.0 • eWASM • Storage rent • State pruning	• Proof of Stake • Ethereum 1 -> 2 Deposit Contract • Blockchain Entropy	• Sharding • Cross links • Decentralized P2P networking at scale	• Smart Contracts • eWASM • Storage/state rent	• Off chain state storage

https://www.numbrs.com/tech/2021/04/15/scaling-blockchain-systems-via-sharding-on-the-example-of-eth-2-0/

eWasm

eWasm = the Ethereum Wasm

eWasm = "restricted WASM subset" configures/modified for Ethereum

Important part of Ethereum 2.0 is the transition from EVM to eWasm

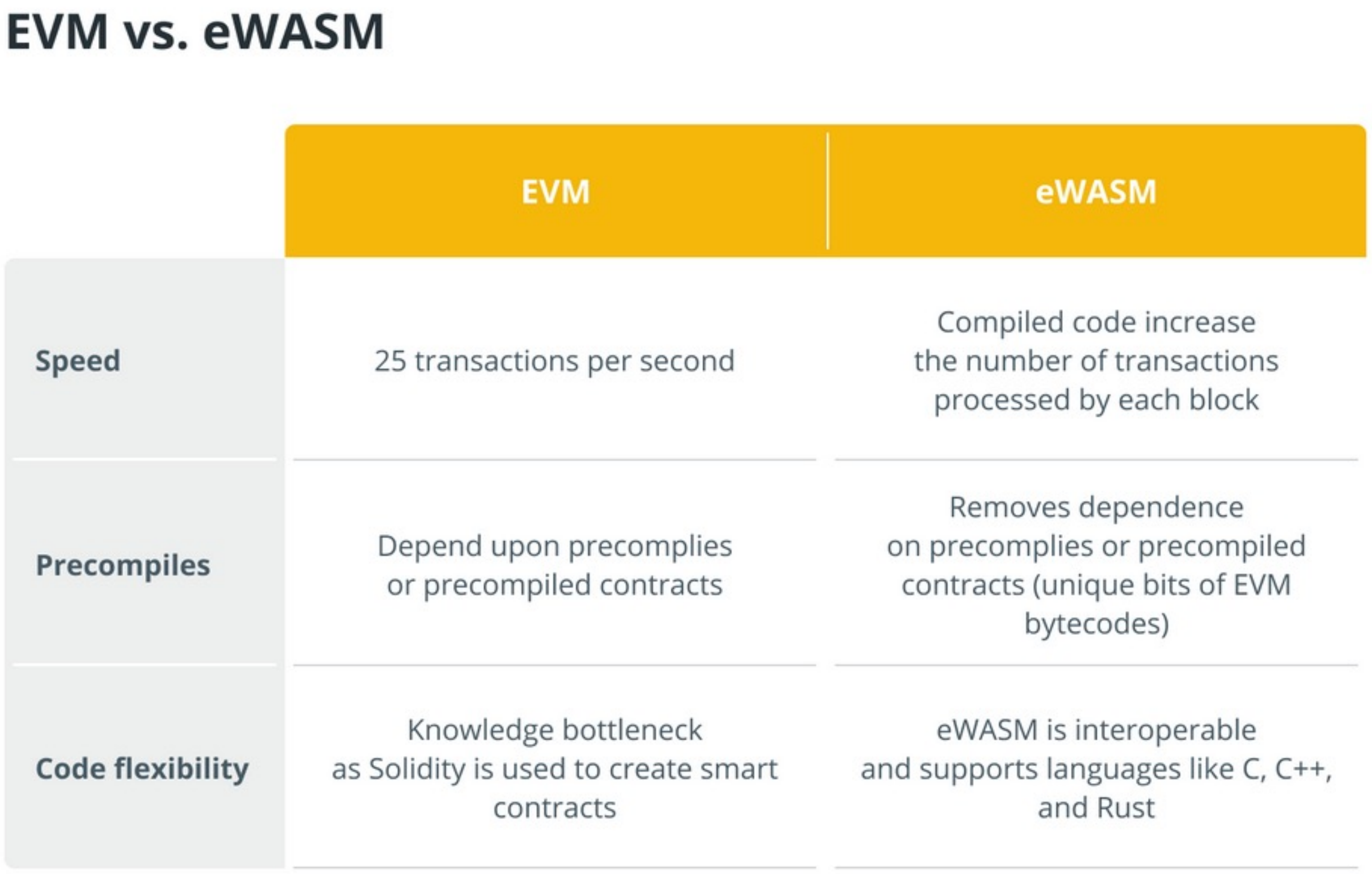
eWasm key features

- Still under development
- Near-native execution speed for smart contracts
- Develop smart contracts in C, C++, and Rust
- Access to a vast developer community and the toolchain surrounding WebAssembly

eWasm – main goals

- The main goals for the eWasm project are to offer:
- A specification of *ewasm contract* semantics and the *Ethereum interface*
- An *EVM transcompiler*, (e.g. as an ewasm contract)
- A *metering injector*, (e.g. as an ewasm contract)
- A VM implementation for executing ewasm contracts
- An ewasm backend in the Solidity compiler
- A library and instructions for writing contracts in Rust and C

eWasm vs EVM



EVM vs. eWASM

	EVM	eWASM
Speed	25 transactions per second	Compiled code increase the number of transactions processed by each block
Precompiles	Depend upon precomplies or precompiled contracts	Removes dependence on precomplies or precompiled contracts (unique bits of EVM bytecodes)
Code flexibility	Knowledge bottleneck as Solidity is used to create smart contracts	eWASM is interoperable and supports languages like C, C++, and Rust

https://cointelegraph.com/ethereum-for-beginners/ethereum-upgrades-a-beginners-guide-to-eth-2-0

Conclusions

Conclusions

- The architectural advancements of blockchains have paved the way to the evolution of the technology
- From a software engineering point of view, blockchain can be seen as software component that provides
  - data storage,
  - computation services,
  - communication services,
  - asset management and control functions.
- Smart contract engines are vital part of blockchain architectures
- Smart contracts are self-executing and self-enforcing contracts that have brought significant innovation to the ecosystem
- Smart contracts lifecycle consists of four stages: creation, deployment, execution and completion
- Ethereum Virtual Machine is a runtime environment that enables the execution of smart contracts in Ethereum
- WebAssembly is a fresh and very promising approach for running smart contracts
- eWasm is under development and it will be part of Ethereum 2.0. Ethereum works on the transition from EVM to eWasm

Glossary

Glossary

Sidechains: are blockchains that allow for digital assets from one blockchain to be used securely in a separate blockchain and subsequently returned to the original chain

Blockchain Interoperability: two or more blockchain systems can interact with other communicate and share value.

Multi-chain Frameworks: Blockchains can plug into a framework to become a part of the standardized ecosystem and transfer data and value between each other.

Atomic Swaps: to allow users to trade one cryptocurrency for another directly in a peer-to-peer transaction.

Merged Consensus: allow for two-way interoperability between chains through the use of a relay chain. Merged consensus can be quite powerful, but generally must be built into the chain from the ground up. Projects like Cosmos and ETH2.0 use merged consensus.

Glossary

Ewasm contract: a contract adhering to the ewasm specification

Ethereum environment interface (EEI): a set of methods available to ewasm contracts

metering injector: a transformation tool inserting metering code to an ewasm contract

EVM transcompiler: an EVM bytecode (the current Ethereum VM) to ewasm transcompiler.

Further Readings

Further Reading

- Xu, X., Weber, I., & Staples, M. (2019). Architecture for blockchain applications. Springer.
- Jin, Hai, Xiaohai Dai, and Jiang Xiao. "Towards a novel architecture for enabling interoperability amongst multiple blockchains." In 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 1203-1211. IEEE, 2018.
- Kannengießer, Niclas, Michelle Pfister, Malte Greulich, Sebastian Lins, and Ali Sunyaev. "Bridges Between Islands: Cross-Chain Technology for Distributed Ledger Technology." In 53rd Hawaii International Conference on System Sciences, vol. 2020. 2020.
- Buterin, Vitalik. "Chain interoperability." R3 Research Paper (2016).
- Wang, Hui, Yuanyuan Cen, and Xuefeng Li. "Blockchain router: a cross-chain communication protocol." In Proceedings of the 6th International Conference on Informatics, Environment, Energy and Applications, pp. 94-97. ACM, 2017.
- Johnson, Sandra, Peter Robinson, and John Brainard. "Sidechains and interoperability." arXiv preprint arXiv:1903.04077 (2019).
- Vitalik Buterin – Chain Interoperability
  - https://www.bubifans.com/ueditor/php/upload/file/20181015/1539602892605747.pdf

UNIVERSITY *of* NICOSIA

Questions?

Contact Us:

Twitter: @mscdigital
Instructor’s Email: christodoulou.kl@unic.ac.cy

Course Support:
Mark Wigmans - wigmans.m@unic.ac.cy
Marios Touloupos - touloupos.m@unic.ac.cy

IT & live session support: dl.it@unic.ac.cy