

(Online Appendix) Understanding Smart Contracts: Hype or Hope?

Elizaveta Zinovyeva

Raphael C. G. Reule

Blockchain Research Center, Humboldt-Universität zu Berlin, Germany.

International Research Training Group 1792, Humboldt-Universität zu Berlin, Germany.

elizaveta.zinovyeva[at]wiwi.hu-berlin.de irtg1792.wiwi[at]wiwi.hu-berlin.de

Wolfgang Karl Härdle

Blockchain Research Center, Humboldt-Universität zu Berlin, Germany.

Wang Yanan Institute for Studies in Economics, Xiamen University, China.

Sim Kee Boon Institute for Financial Economics, Singapore Management University, Singapore.

Faculty of Mathematics and Physics, Charles University, Czech Republic.

National Chiao Tung University, Taiwan.

haerdle[at]wiwi.hu-berlin.de

June 22, 2021

1 Appendix

1.1 List of cryptocurrencies used in this research

Abbrev.	CC	Website
BTC (XBT)	Bitcoin	bitcoin.com, bitcoin.org
ETC	Ethereum Classic	ethereumclassic.github.io
ETH	Ethereum	ethereum.org
LEO	UNUS SED LEO	bitfinex.com (iFinex ecosystem)
USDC	USD Coin	centre.io/usdc

1.2 List of abbreviations

Terminus	Abbrev.
Blockchain	BC
Cryptocurrency	CC
Smart Contract (general terminus)	SC
Verified Smart Contract (Source Code public)	VSC
decentralized Apps	DApps
State of the DApps	SDA
Externally Owned Account	EOA
Contract Account	CA
Transaction (BC recorded)	TX
Call/Message (“internal TX”)	MSG

1.3 Topics in the literature research

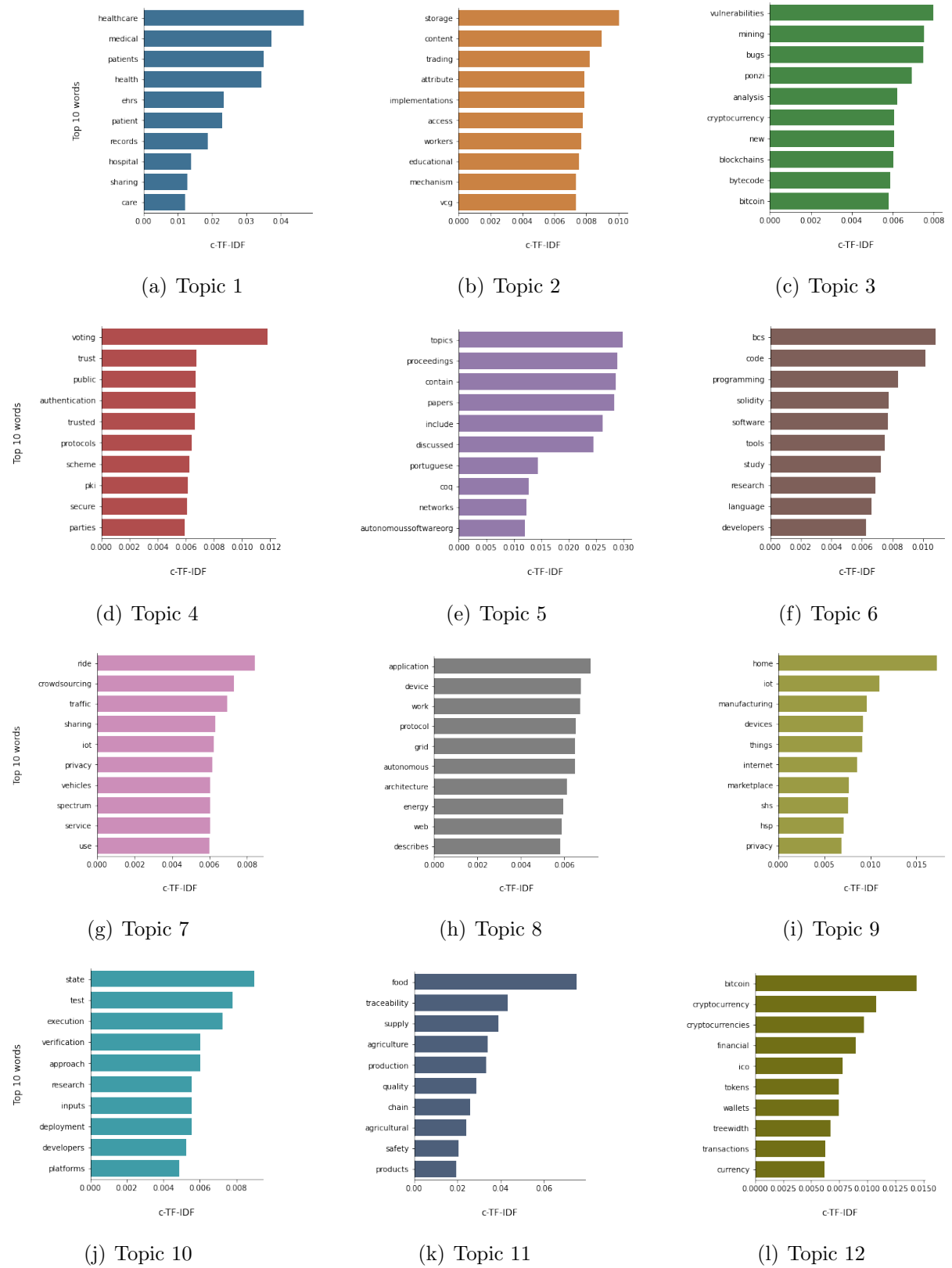



Figure 1: Top 10 the most important words per topic identified in the existing SC research (Part 1) 

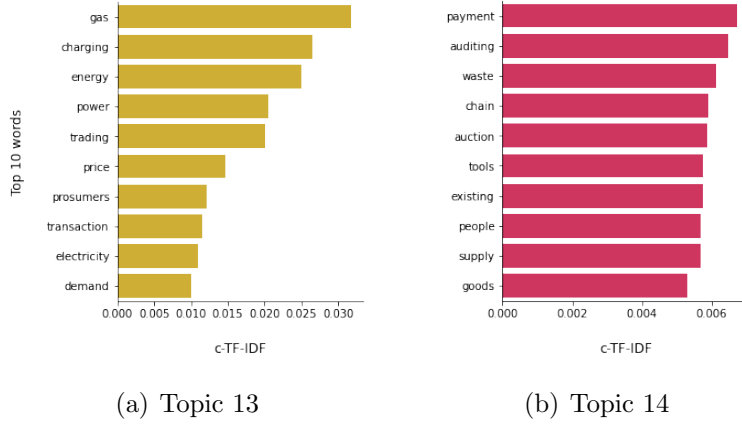


Figure 2: Top 10 the most important words per topic identified in the existing SC research (Part 2) [🔗](#)

1.4 ETH value denominations

Further Units and Globally Available Variables used in Solidity can be accessed through [Ethereum - Read the Docs](#).

Unit	Wei Value	Wei
Wei	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
Microether (szabo)	1e12 wei	1,000,000,000,000
Milliether (finney)	1e15 wei	1,000,000,000,000,000
Ether	1e18 wei	1,000,000,000,000,000,000

1.5 Elliptic Curve Digital Signature Algorithm

Asymmetric cryptography is used to create accounts in ETH in three steps: Firstly, the private key associated with an EOA is randomly generated as SHA256 output and could look like this: b032ac4de581a6f65c41889f2c90b3a629dc80667bf7167611f8b5575744f818 - a random 256 bit/32 bytes big and 64 hex character long output. Secondly, the public key is derived from the private key via the Elliptic Curve Digital Signature Algorithm (secp256k1 ECDSA) and could then look like this: fc2921c35715210aeb0f2fffeaad94d906aaf2feda8a71e52c5d0a0da8ada4a44099e8ea65e3ec214b6686189255bba2373bd2ee6b05520dfd4dc571b682ccc3 - a 512 bits/64 bytes big and 128 hex character long output. The public key is therefore subsequently calculated from the private key, but as we are dealing with a *trapdoor function*, this is not possible vice versa being an irreversible calculation. A private key is therefore kept non-public, whereas the public key

is used to derive, in a third step - via Keccak-256 hashing of that public key, which results in a bytestring of length 32, from which the first 12 bytes are removed and result in a bytestring of length 20 - the individual ETH *address*, that could look like 0x9255bba2373BD2Ee6B05520DfD4dC571B682b349 - a 160 bits/20 bytes big and 40 hex character long output (leaving out the 0x prefix). A public key can hence be used to determine, that the given signature of the information at hand is genuine, that means created with the respective public key and address of the interactor, without requiring the private key to be divulged (see further appendix 1.6 and Buchanan, 2020).

1.6 Zero-Knowledge Proofs

The standard notion of a mathematical proof can be related to the definition of an non-deterministic polynomial (NP) time complexity class of decision problems. That is, to prove that a statement is true one provides a sequence of symbols on a piece of paper, and a verifier checks that they represent a valid proof. Usually, though, some additional knowledge, other than the sole fact that a statement is true, is gained as a byproduct of the proof. Zero-knowledge proofs were introduced as a way to circumvent that, i.e., to convey no additional knowledge beyond proving the validity of an assertion. Goldwasser, Micali, and Rackoff (1989) first described zero-knowledge proofs as interactive proof systems.

As the term itself suggests, interactive zero-knowledge proofs require some interaction between a prover and a verifier. Intuitively, a proof system is considered zero-knowledge if whatever the verifier can compute, while interacting with the prover, it can compute by itself without going through the protocol (Goldreich and Oden, 1994). Formally they can be defined as follows, as per Boaz and Sanjeev (2009).

Given an interactive proof system, or an interactive protocol (P, V) , where P and V can be seen as interactive probabilistic polynomial-time (PPT) Turing machines symbolizing a Prover and a Verifier, for a formal NP-language $L \subset \{0, 1\}^*$ and an input x , the *output* of V on x at the end of interaction between P and V can be written as

$$out_V[P(x), V(x)].$$

(P, V) is called a zero-knowledge protocol for L if the following three conditions hold:

Completeness:

$$\forall x \in L, u \in \{0, 1\}^*, \quad Pr[out_V[P(x, u), V(x)]] \geq 2/3,$$

where u is a certificate for the fact that $x \in L$. In other words the prover can convince

the verifier of $x \in L$ if both follow the protocol properly.

Soundness: If $x \notin L$, then

$$\forall P^*, u \in \{0, 1\}^*, \quad \Pr[out_V[P^*(x, u), V(x)]] \leq 1/3.$$

I.e., the prover cannot fool the verifier, except with small probability.

Perfect Zero-Knowledge: For every strategy V^* there exists an expected PPT simulator S^* such that

$$\forall x \in L, u \in \{0, 1\}^*, \quad out_{V^*}[P(x, u), V^*(x)] \equiv S^*(x).$$

The last condition prevents the verifier from learning anything new from the interaction, even if she does not follow the protocol but rather uses some other strategy V^* . Otherwise she could have learned the same thing by just running the simulator S^* on the publicly known input x . S^* is called the simulator for V^* , as it simulates the outcome of V^* 's interaction with the prover without any access to such an interaction.

1.7 t -SNE (t -Distributed Stochastic Neighbor Embedding)

t -SNE is a non-linear technique for dimension reduction and data visualisation. it allows to preserve the local structure and is proposed by v. d. Maaten and Hinton (2008). It aims to design an embedding of high-dimensional input to low-dimensional map while preserving much of a significant structure. On the algorithm 1 below, the reader can find the pseudocode of the t -SNE computation.

$$X = \{x_1, x_2, \dots, x_n\} \rightarrow Y = \{y_1, y_2, \dots, y_n\}$$

x_i is the i^{th} object in high-dimensional space.

y_i is the i^{th} object in low-dimensional space.

Algorithm 1 t -SNE Pseudocode

Data: data set $\chi = \{x_1, x_2, \dots, x_n\}$

cost function parameters: perplexity $Perp$,

optimization parameters: number of iterations T , learning rate η , momentum $\alpha(t)$

Result: low-dimensional data representation $Y^{(T)} = \{y_1, y_2, \dots, y_n\}$

begin

 compute pairwise affinities $p_{j|i}$ with perplexity $Perp$ (using Equation 1) set $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$ sample initial solution $Y^{(0)} = \{y_1, y_2, \dots, y_n\}$ from $N(0, 10^{-4}I)$

for $t = 1$ **to** T **do**

 compute low-dimensional affinities q_{ij} (using Equation 2))

 compute gradient $\frac{\delta C}{\delta Y}$ (using Equation 3))

 set $Y^{(t)} = Y^{(t-1)} + \eta \frac{\delta C}{\delta Y} + \alpha(t)(Y^{(t-1)} - Y^{(t-2)})$

end

end

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|x_i - x_k\|^2 / 2\sigma_i^2)} \quad (1)$$

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_i - y_l\|^2)^{-1}} \quad (2)$$

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1} \quad (3)$$

1.8 UMAP vs. t -SNE

The dimensionality reduction algorithm that we are using here is the Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP) (McInnes, Healy, and Melville, 2018) and is like t -SNE – a neighbor graph algorithm. The mathematical foundations of the UMAP rely on Laplacian Eigenmaps and are very extensive. The important differences between t -SNE and the UMAP are the following. The UMAP aims to better preserve more of the global structure while requiring less computational time. As compared to the t -SNE, the UMAP relies not only on the the Kullback-Leibler divergence measure, but on the cross-entropy.

$$C_{UMAP} = \sum_{i \neq j} \left\{ v_{ij} \log \left(\frac{v_{ij}}{w_{ij}} \right) + (1 - v_{ij}) \log \left(\frac{1 - v_{ij}}{1 - w_{ij}} \right) \right\} \quad (4)$$

where v_{ij} are the pair-wise similarities in the high dimensional space and w_{ij} - in the low-dimensional. The optimization problem used in the UMAP is the stochastic gradient descent instead of gradient descent used in the t -SNE, which speeds up the computations and decreases the required memory resources. Moreover, UMAP does not require the

distance Euclidean.

1.9 Example Code

Adding to the source code review and possible use cases, we are presenting a simple *Hello World*-esque SC as provided by the Ethereum Github with explanatory adaptations to outline some of the technical complexity that has a grave impact on every adjacent structure. We have observed many non-technical outlets – especially Blogs and the such – discussing a theme, that they have apparently never seen as code itself – consequently, we will also keep this to a very brief overview to introduce the structures in a very simple example. Each comment starts with `/**` and ends with `/*`.

```
/** Version control for the compiler, see above section 6, as each Solidity version may have  
    ↪ different commands that can be coded – higher versions will obviously improve  
    ↪ efficiency of the coding and lead to better code controlling. */  
pragma solidity >=0.4.22 <0.6.0;  
  
/** "Mortal" is the name of this SC. */  
contract Mortal {  
    /** Defines the variable "owner" of the type "address". */  
    address owner;  
  
    /** The "constructor" is executed at initialization and sets the owner of the SC, i.e., it is  
        ↪ executed once when the CA/SC is first deployed. Similar to other class-based  
        ↪ programming languages, it initializes state variables to specified values. "msg.  
        ↪ sender" refers to the address where the CA is being created from, i.e., here in the  
        ↪ constructor setting the "owner" to the address of the SC creator. SCs depend on  
        ↪ external TX/MSG to trigger its functions, whereas "msg" is a global variable  
        ↪ that includes relevant data on the given interaction, such as the address of the  
        ↪ sender and the value included in the interaction. This is assured by the "public"  
        ↪ function, which can be called from within the CA/SC or externally via MSG's,  
        ↪ like here getting the address of the interactor. "private" functions are not callable  
        ↪ and can only reached by the SC itself – a particular source for grave errors, as  
        ↪ you can not change the SC once deployed. */  
    constructor() public { owner = msg.sender; }  
  
    /** Another important function, and source for grave errors if missing, follows and  
        ↪ represents a mean to recover funds stored on the CA. Alternatively, calling "  
        ↪ selfdestruct(address)" sends all of the SCs current balance to address specified.  
        ↪ Remember, that once deployed the SC can not be changed unlike non-BC-  
        ↪ based software. The only way to modify an SC is to deploy a corrected one –
```



```

    ↪ best after deactivating and recovering all funds in the problematic one.
    ↪ Interestingly, "selfdestruct" consumed "negative Gas", as it frees up BC/EVM
    ↪ space by clearing all of the CA/SCs's data./*
function kill() public { if (msg.sender == owner) selfdestruct(msg.sender); }
}

/** After "Mortal", "Greeter" is another SC presented to visualize, that CA/SCs can "inherit
    ↪ " characteristics of CA/SCs enabling SCs to be written shorter and clearer. By
    ↪ declaring that "Greeter is Mortal", "Greeter" inherits all characteristics of "Mortal"
    ↪ and keeps the "Greeter" code herewith crisp and clear to to point, where is has
    ↪ individual functions to be executed. In this example, the inherited characteristic of "
    ↪ Mortal" gives, as defined beforehand in "Mortal", that "Greeter" can be deactivated
    ↪ with all locked funds being recovered. /*
contract Greeter is Mortal {

    /** Defines the variable "greeting" of the type "string", i.e., a sequence of characters. /*
    string greeting;

    /** This is defined as beforehand in "Mortal", whereas in this case the underscore in "\
    ↪ _greeting" is a style used to differentiate between function arguments and global
    ↪ variables. There is no semantic difference between "greeting" and "_greeting",
    ↪ whereas the latter one is defined as such not to shadow the first one. Here, the
    ↪ underscore differentiates between the global variable "greeting" and the
    ↪ corresponding function parameter. Strings can be stored in both "storage" and "
    ↪ memory" depending on the type of variable and usage. "memory" lifetime is
    ↪ limited to a function MSG and is meant to be used to temporarily store variables
    ↪ and respective values. Values stored in "memory" do not persist on the network (
    ↪ EVM & BC) after the interaction has been completed. /*

    constructor(string memory _greeting) public {
        greeting = _greeting;
    }

    /** Main function of the SC that returns the greeting once "greet" function is MSG'ed
    ↪ /*
    function greet() public view returns (string memory) {
        return greeting;
    }
}

```

References

- Boaz, B. and A. Sanjeev. 2009. *Computational Complexity: A Modern Approach*. Princeton University: Cambridge University Press, 1st edition ed.
- Buchanan, W. 2020. “Ethereum Address Generation, Asecuritysite.” URL <https://asecuritysite.com/encryption/ethadd>. Online; accessed 10 January 2021.
- Goldreich, O. and Y. Oden. 1994. “Definitions and properties of zero-knowledge proof systems.” *Journal of Cryptology* 7:1–32.
- Goldwasser, S., S. Micali, and C. Rackoff. 1989. “The Knowledge Complexity of Interactive Proof Systems.” *SIAM Journal on Computing* 18:168–298.
- McInnes, L., J. Healy, and J. Melville. 2018. “Umap: Uniform manifold approximation and projection for dimension reduction.” *arXiv preprint arXiv:1802.03426* .
- v. d. Maaten, L. and G. Hinton. 2008. “Visualizing data using t-SNE.” *Journal of machine learning research* 9 (Nov):2579–2605.