M. Gimenez-Aguilar, J. M. De Fuentes, L. González-Manzano and C. Camara, "Zephyrus: An Information Hiding Mechanism Leveraging Ethereum Data Fields," in *IEEE Access*, vol. 9, pp. 118553-118570, 2021.

DOI: 10.1109/ACCESS.2021.3106713

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 19, 2021, accepted August 11, 2021, date of publication August 23, 2021, date of current version September 1, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3106713

Zephyrus: An Information Hiding Mechanism Leveraging Ethereum Data Fields

MAR GIMENEZ-AGUILAR, JOSE M. DE FUENTES, LORENA GONZÁLEZ-MANZANO, AND CARMEN CAMARA

Computer Security Laboratory, Universidad Carlos III de Madrid, 28911 Leganes, Spain

Corresponding author: Jose M. De Fuentes (jfuentes@inf.uc3m.es)

This work was supported in part by the Spanish Ministry of Science and Innovation under Grant ODIO/COW(PID2019-111429RB-C21), in part by the Region of Madrid under Grant CYNAMON-CM(P2018/TCS-4566), in part by European Structural Funds European Social Fund (ESF) and Fondo Europeo de Desarrollo Regional (FEDER), in part by the Madrid Government (Comunidad de Madrid) under the Multiannual Agreement with Universidad Carlos III de Madrid (UC3M) in the line of "Fostering Young Doctors Research" under Grant CAVTIONS-CM-UC3M, in part by the context of the V PRICIT (Regional Programme of Research and Technological Innovation), and in part by the Excellence Program for University Researchers.

ABSTRACT Permanent availability makes blockchain technologies a suitable alternative for building a covert channel. Previous works have analysed its feasibility in a particular blockchain technology called Bitcoin. However, Ethereum cryptocurrency is gaining momentum as a means to build distributed apps. The novelty of this paper relies on the use of Ethereum to establish a covert channel considering all transaction fields and smart contracts. No previous work has explored this issue. Thus, a mechanism called *Zephyrus*, an information hiding mechanism based on steganography, is developed. Moreover, its capacity, cost and stealthiness are assessed both theoretically, and empirically through a prototype implementation that is publicly released. Disregarding the time taken to send the transaction to the blockchain, its retrieval and the mining time, experimental results show that, in the best case, 40 Kbits can be embedded in 0.57 s. for US$ 1.64, and retrieved in 2.8 s.

INDEX TERMS Ethereum, information hiding, steganography, blockchain.

I. INTRODUCTION

Blockchain is a well-known technology that allows the execution of transactions ensuring their integrity. In short, a blockchain is described as "an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way" [1].

This technology has been used in different fields such as health care [2], financial contracts [3] or digital rights management [4]. Among all possible uses of a blockchain, cryptocurrencies are quite well-known. A cryptocurrency is a digital asset that serves as a medium of exchange [5]. Bitcoin [6] and Ethereum are two of the main cryptocurrencies nowadays [7].

The extensive adoption of cryptocurrencies make them interesting to build covert channels (that is, a way to secretly send information) over a publicly available medium (that is, the list of transactions of the cryptocurrency). Since a great variety of use cases can be devised, in the following several examples are considered.

The associate editor coordinating the review of this manuscript and approving it for publication was Yassine Maleh.

Motivating use cases. There are three situations in which such a covert communication is interesting, although with different degrees of immediacy. On one hand, in the *panic button case* a threatened individual is willing to leave some secret material (e.g., account keys) to be released in case of emergency and thus, without immediacy in mind. On the other hand, in a *sabotage case* a malicious insider aims to immediately exfiltrate sensitive data without being detected. Also looking for this feature, in a *censorship case* an individual is willing to share information in a controlled and censored environment.

Blockchain provides a set of intrinsic features (availability and integrity [8]), which make it attractive for this purpose. Thus, any data inserted into the ledger will remain unaltered and readable by any party virtually anytime.

To hide a secret in such a setting, steganography is the art of concealing messages within a non-secret piece of data called cover [9]. It is a branch of cryptography used when discretion is a priority. Steganographic approaches can be generally divided into implicit and explicit ones. Implicit techniques rely upon the way in which the system is used [10]. For example, if the sending time is odd or even, it can be

understood by the receiver as 1 or 0, respectively. On the other hand, explicit approaches base on modifying the cover to embed the secret [11]. In this paper we focus on this latter, information-based approaches.

Several works have already applied steganography in Bitcoin for use cases like the examples above [12]–[15]. For example, it has been applied to counter censorship [16]. However, few efforts have been devoted to information hiding in Ethereum. To the best of authors' knowledge, only [17] leveraged a single data field, and [18] proposed the use of a Ethereum-related protocol. Thus, no previous work has focused on leveraging Ethereum for covert communications considering all its data fields. Most of them have never been used for steganographic purposes and this paper addresses this issue for the first time. Since this cryptocurrency holds significant differences against Bitcoin, it is necessary to characterize its suitability for this purpose. In particular, Ethereum is gaining momentum thanks to its support to distributed apps by means of smart contracts. They are pieces of software that can be executed without human intervention. Since they are stored in the blockchain, they remain permanently [19]. Moreover, the underlying data structures that are also stored in the blockchain are heavily different to those present in Bitcoin. Thus, the unique features of Ethereum motivate the need for proposing a tailored mechanism for covert communications.

In this paper the following contributions are achieved:

- Development of a steganographic system in Ethereum, called *Zephyrus*.[1] It considers all Ethereum transaction and smart contracts fields to hide a secret. The design of *Zephyrus* leverages a large amount of real-world Ethereum blockchain data to ensure the stealthiness of the secret. Indeed, the mechanism is assessed in terms of capacity, stealthiness and cost.
- An open-source proof of concept is released to foster further research. Moreover, it is used to assess the time taken for embedding and revealing a secret in a real-world Ethereum network.

The structure of the paper is the following. Section II provides the background to understand the proposal. Afterwards, the underlying model is described in Section III. Since the goal is to mimic existing transactions and contracts, Section IV focuses on a preliminary study of current Ethereum data. The proposed mechanism is introduced in Section V, whereas Section VI focuses on its evaluation. Section VII analyses the related work. Finally, Section VIII concludes the paper and points out future research directions.

II. BACKGROUND

In this Section, the main concepts needed to understand the proposal are presented. Thus, the basics on steganographic systems are introduced in Section II-A. Afterwards, the main notions of Ethereum are described in Section II-B.

[1]The name *Zephyrus* is based on one of the winds of the Greek mythology called *Anemoi*. *Anemoi* lives in the upper-air or *Aether*, which later derived in the English word *Ether*. Thus, *Zephyrus* lives in *Ether* but being invisible to the plain eye.

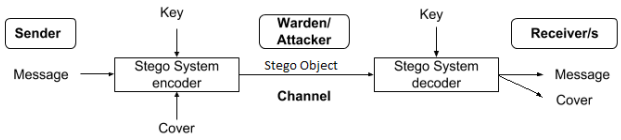

FIGURE 1. Entities in a steganographic system.

A. STEGANOGRAPHIC SYSTEMS

In a steganographic system different elements are involved, see Fig.1. A sender and one or more receivers share a key to hide or extract a secret message. This key is essential to provide confidentiality by means of encryption [20]. Basically, the sender has an element (e.g. code, image, text, etc.), called cover, in which a secret is hidden. The steganographic system receives the cover, the secret and the key and outputs a steganographic object. This object is sent to the receiver through the channel. In the destination, the receiver uses the steganographic object and the shared key to get the hidden message. Between both parties, an attacker or warden might be placed. It can be passive, thus eavesdropping the channel, or active, thus being able to tamper with the transmitted data [21].

In order to transmit information, different operations have to be carried out for hiding (also called embedding) and retrieving (usually called revealing) the secret at stake. In information-based steganography, one of the most well-known examples is the Least Significant Bit (LSB) [22]. In LSB, the cover's rightmost bit(s) are replaced by the secret. This technique should be applied while keeping the cover appearance to avoid raising suspicions.

B. ETHEREUM

Ethereum is an open-source decentralized platform that runs smart contracts. They are pieces of code executed by the nodes maintaining the network. In this way, censorship or code changes by third parties are avoided, thus enabling building distributed applications [23]. Apart from smart contracts, Ethereum enables sending funds among parties as any other cryptocurrency. Every interaction with this blockchain is carried out through transactions. To avoid false transactions, a distributed consensus algorithm is run by Ethereum nodes. This process involves a computational task called mining. In Ethereum, mining involves a trial-and-error process until the result of a cryptographic process (in particular, a hash function) meets a given condition [24].

In order to run smart contracts and maintain the blockchain, Ethereum comes with a decentralized virtual machine called EVM. It uses a Turing-complete language to be able to create sophisticated smart contracts [25].

In the following, the different data items of transactions and smart contracts are introduced.

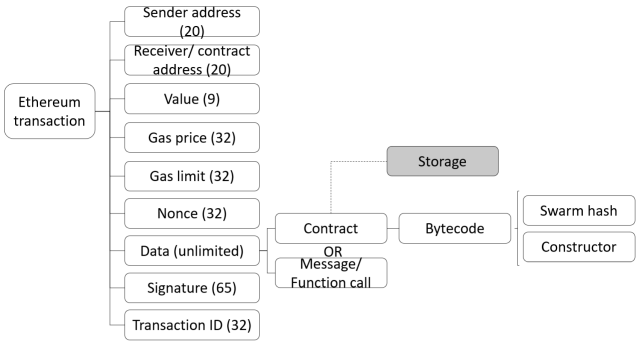

FIGURE 2. Ethereum transaction fields with their size in bytes. White boxes are stored immutably in the blockchain, while the greyed one is not.

1) ETHEREUM TRANSACTIONS

A transaction in Ethereum contains several fields of different size (Fig. 2). Each transaction is sent by a user, identified by a *Sender address* that results from the 20 last bytes of the hash of the user's public key. The receiver is also identified by an address. Note that it can be another user (using a *Receiver address*) or a smart contract. In the latter case, the *Contract address* comes from the hash of the sender address and the amount of transactions sent by that account. A particular case happens when the receiver address is null, that is when a contract is deployed.

On the other hand, the *Value* field includes the amount of funds at stake. Apart from the value itself, two fields (*Gas price* and *Gas limit*) express the costs that the sender assumes to include this transaction in the blockchain. In particular, *Gas price* defines the cost per operation and *Gas limit* sets the maximum incurred cost [26]. Such a cost depends on the information included in the data field. This can be of three types, namely a *text message*, a *contract* to be deployed (as explained in Section II-B2) or a *function call* to an existing contract. In the latter case, an encoded representation (called ABI [27]) of the called function name and arguments is included.

Last but not least, each transaction is identified by a serial number (*Nonce* field) and an *Identifier* which is the hash of the previous fields. The legitimacy of the transaction is shown by the sender's digital signature. The *signature* is formed by three values, namely V, r and s, which are the result of applying the ECDSA algorithm [28].

2) SMART CONTRACTS

A smart contract is a piece of code formed by functions to be executed by any Ethereum node through its EVM. It can be written in different languages, for example, Solidity, Serpent, LLL or Vyper. Those are high-level languages that contain functions, arguments and control flow instructions and operators. Since Solidity is the most widespread one, it is the one considered in this proposal.

When Solidity code is compiled, it is transformed into a hexadecimal string known as *bytecode*. It is formed by opcodes or low-level, human-readable instructions the EVM can execute. For example, JUMP instruction indicates the EVM to go to a particular part of the program and execute it. The order and placement of instructions is essential because they set the execution flow. The cost of each instruction depends on its type [25].

For the interest of this proposal, it is relevant to understand how the bytecode ends its execution. Before the last instruction (STOP or ASSERT), usually a JUMP is placed, which makes the code continue its execution from an address pointed out in that instruction. That address must contain a JUMPDEST instruction – otherwise the EVM throws an error. We will refer to this region as the *JUMP-JUMPDEST block*.

Apart from *bytecode*, metadata describing the contract at stake is produced by the compiler when Solidity is used. This information is intended to be published in an external repository to help in verifying the contract integrity. Thus, the *Swarm hash* field (which is the hash of the contract, including its file name) is included at the end of bytecode. It serves as a pointer to find the contract in a content-addressed storage outside the blockchain [29].

Smart contracts may optionally store their current state. Such *storage* can be initialized with a special function called *constructor*. Stored values can be updated by calling to the appropriate contract functions. Since storage leverages Ethereum nodes' memory space, these operations involve additional costs.

III. MODEL

The model consists of the description of the involved entities and attackers (Section III-A), goals at stake (Section III-B) and working assumptions (Section III-C).

A. ENTITIES AND ATTACKER MODEL

In this steganographic system two entities are identified, namely sender and receiver, communicating through a channel – the Ethereum blockchain. While the sender transmits information, the receiver is merely an observer of the blockchain.

With respect to the attacker, three different types are considered. A pair of them are assumed to be passive, inspecting blockchain contents using a block explorer (e.g., Etherscan [30]). However, while one of them is an eavesdropper (Basic Eavesdropper, BE), the other one might carry out syntactic checks on each transaction (Advanced Eavesdropper, AE). The third type of attacker (Interactive Attacker, IA) is active, being able to make transactions. Therefore, while BE and AE threaten the secret's confidentiality, IA aims to impact its integrity.

B. GOALS

The development of a steganographic mechanism should be designed to be resilient against any kind of suspicion. In this regard, the following goals are identified:

TABLE 1. Characterization per data field.

Field	Type	Number of different values	Min(# transactions, possible values)	Coverage (%)	Mean appearances	St. dev. appearances	Top 8 accum. freq.	Top 16 accum. freq.	Selected
Value	to user	5633180	8998787	62.60%	1.60	93.82	5.75%	8.11%	✓
	to function	165939	7943428	2.09%	47.87	18186.83	95.38%	96.01%	X
	contract	84	65346	0.13%	777.93	7069.38	99.87%	99.89%	X
Gas Price	to user	69032	8998787	0.77%	130.36	6928.91	48.68%	**65.20%**	✓
	to function	61139	7943428	0.77%	129.92	5294.73	39.96%	**57.98%**	✓
	contract	9007	65346	13.78%	7.26	117.42	40.12%	**57.13%**	✓
Gas Limit	to user	5713	8998787	0.06%	1575.14	57930.74	86.70%	92.24%	X
	to function	133700	7943428	1.68%	59.41	3966.78	11.30%	16.25%	X
	contract	6322	65346	9.67%	10.34	170.77	**52.48%**	72.39%	✓
Function arguments	uint256	3810193	11756671	32.41%	3.09	325.47	12.22%	16.21%	✓
	bytes32	1792159	2135859	83.91%	1.19	8.69	0.87%	1.09%	X
Constructor arguments	uint256	607	2328	26.07%	3.84	12.46	31.40%	42.23%	✓
Bytecode	PUSH1	53	256	20.70%	1754.60	5167.38	**98.09%**	99.47%	✓
	PUSH20	17	16275	0.10%	957.35	3937.99	99.94%	99.99%	X

- **Stealthiness**: embedded messages should be difficult to identify for an attacker.
- **Simplicity**: any user who is able to interact with the Ethereum blockchain should be able to use the mechanism.
- **Efficiency**: the mechanism should be efficient in terms of time and amount of sent information. It should allow sending a practical amount of information in an affordable amount of time.
- **Cost**: sending hidden information should be economically affordable for the sender.
- **Secret integrity**: hidden information's integrity should remain over time.

C. WORKING ASSUMPTIONS

The following assumptions are considered in this proposal:

- The receiver knows the following data items to get access to the secret:
  - The first transaction identifier. Sending a secret may involve several transactions, but knowing just the first identifier should be enough to retrieve the whole message.
  - Cryptographic materials, that is, the encryption key and a random number called *nonce*.
  - Fields in which the secret is embedded.
- Secrets are sent sequentially, thus avoiding sending simultaneous messages.
- The sender uses the same source or destination Ethereum address for a given secret.

IV. PRELIMINARY ETHEREUM DATA STUDY

Since *Zephyrus* aims to achieve stealthiness, transactions and contracts including secrets must mimic existing ones. For this purpose, we have analysed 16,942,215 transactions and 65,346 contracts. In order to reflect the evolution of transactions over time, we have considered one week every six months from 2017 to 2019. In particular, transactions are collected between March 24th and April 1st, and between September 24th and October 1st each year.

Once collected, transactions have been classified into three categories: sent to another blockchain user (8,998,787 transactions), to a function in a contract (7,943,428 transactions) and to deploy a contract (65,346 transactions). The proportion is in line with expectations, as transactions among Ethereum accounts are the most prevalent ones.

TABLE 2. Most common types of arguments and their coverage.

Field	Type	Coverage (%)
Function arguments	uint256	45.88%
	address	38.04%
	bytes32	8.33%
Constructor arguments	uint256	17.90%
	address	58.21%
	string	16.82%

TABLE 3. Fields with highest prevalence of patterns.

Field	Type	% of values ending in zero(s)
Value	To user	76.36%
Gas price	To user	95.09%
	To function	96.47%
	Contract deployment	85.28%
Gas limit	To user	97.09%
	To function	71.64%
	Contract deployment	75.14%
Function argument	uint256	67.84%
Constructor argument	uint256	82.95%

Since the secret is embedded in one or more transaction fields, or as part of the contract code, the analysis is carried out for each one independently. It must be noted that some fields are freely set by the user whereas others are the result of a cryptographic operation (e.g., hash function). Therefore, the techniques to characterize each field are different. In the former case the variability of each field is analysed by using statistical measures (Section IV-A). In the latter, entropy is studied because randomness is an essential cryptographic property (Section IV-B).

Note that the *Data* field has not been analysed for all transactions. Using this field to insert a secret would raise suspicions. However, function arguments and contract information (which are contained in this field for transactions related to contracts) have been characterized as they could potentially be used for covert communications.

TABLE 4. Analysis of patterns.

Field	Type	Length	Zeros	Number of different values	Min (# transactions, possible values)	Coverage (%)	Mean appearances	St. dev. appearances	Selected
Value	to user	18	10	594039	614569	96.66%	1.03	0.76	X
		18	12	109550	202425	54.12%	1.84	22.9	X
		17	10	384834	469609	81.95%	1.22	4.64	✓
		17	16	9	9	100.00%	22712.18	26543.6	X
Function arguments	uint256	22	18	8075	8100	99.69%	22.675	50.7	✓
		22	21	9	9	100.00%	19619.22	23869.9	X
		21	18	810	810	100.00%	250.9	442.44	X
		21	20	9	9	100.00%	23318.33	20027.3	X
Constructor arguments	uint256	10	2	103	148	69.59%	1.43	0.84	✓
		10	9	6	9	66.67%	22.66	44.36	X
		1	0	8	9	88.89%	24.75	31.36	X

A. VARIABILITY

Intuitively, a high variability of a given data field is beneficial for the sake of embedding secret information. Otherwise, if a given data field always has the same value, any alteration would easily be noticed. Thus, determining the variability of a field requires analysing the amount of different values, and their statistical distribution with respect to all potential values. Several metrics have been considered, namely the coverage of the value range, the mean and standard deviation of the amount of appearances per value, and the prevalence of the most frequent values for each field. Concerning coverage, it must be noted that the amount of collected transactions is usually smaller than the range size. Therefore, the minimum between these two factors will be considered. With respect to prevalences, the accumulated frequency for the 8 and 16 most frequent values is computed. Table 1 summarizes the analysis.

There are some fields that exhibit a suitable variability. For example, *Value* in transactions to another user, shows a reasonable degree of homogeneity. On the contrary, some fields (such as *Gas limit* to other users) are discarded as only two values account for the vast majority of cases.

For those fields which do not have such a variability but cannot be discarded either, the accumulated frequency of the top 8 and 16 elements has been studied. If that frequency is beyond 50%, a given set of values are frequent, so they could also be used to represent a secret. This happens, for example, for the *Gas price* field in bold in Table 1.

The analysis of function and constructor arguments requires special handling, as it is necessary to study each type of argument independently. For simplicity, the most common types are considered herein (see Table 2). In the case of *Function arguments*, they are ''uint256'', ''address'' and ''bytes32'', which together cover 92.25% of transactions (adding Function arguments percentages from Table 2). In the case of *Constructor arguments*, we focus on ''uint256'' and ''address'', which account for 76.11% of cases (adding both percentages of Constructor arguments from Table 2). The third most common type, ''string'', has not been considered as it is usually human-readable.

The analysis of the values contained in these argument types reveals one interesting feature. Particularly, some fields show a prevalent pattern a number ending with a variable amount of consecutive zeros. This is the case of uint256 type for *Function and Constructor arguments*, as well as the *Value* field. For the sake of illustration, 76.36% of transactions to users show this pattern in *Value* field (see Table 3). Table 4 shows the most prevalent patterns. For each one, patterns containing more non-zero digits are regarded as suitable, as they show nice coverage and homogeneity. The only exception is in the *Value* field, due to economic issues explained later (see Section V-B).

TABLE 5. Top 5 pairs of instructions after JUMPDEST.

Instructions	Contracts	Coverage (%)
[POP, POP]	22525	38.41
[PUSH1, SLOAD]	8967	15.29
[PUSH1, DUP1]	3353	5.72
[PUSH1, PUSH1]	2047	3.49
[PUSH1, SWAP1]	1731	2.95

Beyond patterns, the address argument is considered a crypto-related field. Concerning bytes32, given the low mean and standard deviation (see Table 1), as the lack of patterns, it is also studied as crypto-related (see Section IV-B).

Last but not least, the *Bytecode* field requires a tailored analysis. In particular, it is relevant to characterize the amount of instructions, their frequency and the value of their arguments, if any. In all cases, we focus on the JUMP-JUMPDEST block (recall Section II-B), as it is the region that can be altered with lower risks. Concerning the amount of instructions, 9 and 13 are selected as they are the biggest amounts among the most common ones (see Table 14 in the Appendix). POP and PUSH1 are the most common opcodes, covering 39.87% of the cases (see Table 13 in the Appendix). However, as there is room for more variable instructions, the 20 most used opcodes are chosen. Among them, just the variability of PUSH1 and PUSH20 is studied because the remaining opcodes do not use bytes as parameters, so they would offer very low capacity. PUSH20 is discarded due to its high cost and low variability. Finally, the 4 most common pairs of instructions after the JUMPDEST and before the JUMP are selected (see Tables 5 and 6). They cover 62.91% and 76.35% of the sample respectively.

TABLE 6. Top 5 pairs of instructions before the JUMP.

Instructions	Contracts	Coverage (%)
[POP, POP]	32009	54.58
[AND, DUP2]	9262	15.79
[SWAP1, SSTORE]	2779	4.74
[POP, SWAP1]	724	1.23
[SLOAD, DUP2]	364	0.62

B. ENTROPY

Entropy has been computed, combined and individually, in those fields that are the result of cryptographic operations and those meant to represent binary information. The calculus of the individual entropy involves computing Shannon entropy per value in each field, normalized from 0 to 1 [31]. By contrast, combined entropy is calculated concatenating all values per field and computing Shannon entropy. In this way, a high individual entropy ensures random value fields and a high combined entropy guarantees that value fields are different among transactions. Moreover, in both cases the mean and standard deviation are also computed.

Table 7 presents the results of the analysis. Concerning hashes, namely *Receiver address* and *Swarm hash*, we have computed entropy for all studied transactions. Furthermore, we have generated multiple hashes to compare their entropy. This allows us to reason about the possibility of generating hashes with the same or similar entropy to avoid suspicions. Their high entropy with low standard deviation support the uniqueness of the values and their possible use for embedding purposes.

V. PROPOSED MECHANISM

This Section presents *Zephyrus* by introducing both the embedding and revealing procedures of hidden messages for all transaction fields. Given the different nature of the fields at stake as well as their value distribution (recall Section IV), several embedding strategies are firstly proposed in Section V-A.

It must be noted that for the covert communication to take place, the mining procedure must be carried out [32]. However, it is out of the scope of this Section as it is the regular process for every Ethereum transaction. For the sake of brevity, the embedding procedure does not describe the potential retransmissions needed if a transaction is not included in the blockchain. The notation used in the remainder of this proposal is shown on Table 8.

A. EMBEDDING STRATEGIES

According to the previous analysis four different embedding strategies are identified. Table 9 summarizes the strategy applied for each data field. It must be noted that not all fields for all transaction types are considered. For example, not all types of *Function arguments* are selected. Similarly, the *Gas limit* field is used in transactions related to contracts.

On the one hand, crypto-related fields (such as *Receiver addresses*) and and those without patterns (bytes32 and address type) have been shown to have high entropies. Since the embedding mechanism will encrypt the secret (as explained later), the result exhibits high entropy as well. Indeed, this happens for the whole secret and for each individual fragment. Therefore, these fields are used in full (strategy S1).

On the other hand, strategy S2 is applied over those fields which count on acceptable variability, but in which a subset of *numval* values are prominently common. Such values are used for embedding purposes, though the amount of them depends on each field. This leads to a capacity given by Equation 1. For example, if $numval = 8$, 3 bits can be embedded.

$$Capacity_{S2}(bits) = \lfloor \log_2 numval \rfloor \tag{1}$$

Strategy S3 is applied in fields with acceptable variability and exhibiting some patterns in their values. In this case, the embedding operation uses these patterns to ensure that the result seems legitimate. Based on our observations, patterns are formed by a prefix and a suffix. Prefixes are formed by a set of digits ending in any number but 0. Suffixes are a sequence of z zeros. Therefore, for a value of total length l, the capacity of this strategy is given by Equation 2. For instance, for values of length 17 ending with 10 zeros, 22 bits can be embedded.

$$Capacity_{S3}(bits) = \left\lfloor \log_2(81 \times 10^{l-z-2}) \right\rfloor \tag{2}$$

Last but not least, a bytecode-specific strategy S4 is also proposed. As opposed to the previous ones, S4 does not consider the values of the data fields, but the set of instructions contained in the bytecode. Therefore, it provides with variable capacity, as it is explained in the following.

B. EMBEDDING PROCEDURE

The embedding process starts by preparing the secret to make it suitable for Ethereum transactions. Afterwards, data is hidden in fields according to their size and type. The capacity of each field per transaction (summarized in Table 10) is studied, as well as the applied embedding strategy selected according to last column of Table 1 and highlighted in bold in case of S2, and Table 4.

Note that embedding operations, regardless of the field, are limited by *LBB* and *LBGL*. *LBB* refers to the fact that the sender's balance should be bigger than the cost of sending the transaction (incluiding deploying a contract or calling functions). By contrast, *LBGL* refers to the maximum block gas limit, which depends on the network at stake – no transaction can surpass this limit [33], [34].

1) SECRET PREPARATION

The preparation process is depicted in Fig. 3. A key generation function is used to generate keys for the encryption processes (step 0). Firstly, the secret is symmetrically encrypted (step 1). In this process, the secret is adjusted by including encrypted control information. This is essential in the revealing process. In particular, the message length is necessary to

TABLE 7. Entropies per field.

Field	Subfield	Combined entropy	Mean of entropies	Standard deviation of entropies
Sender address	to user	1.00	1.00	0.00
	to function	1.00	1.00	0.00
	contract	1.00	1.00	0.00
Receiver address sample		1.00	0.99	0.01
Receiver address generated		1.00	1.00	0.01
Swarm hash		1.00	1.00	0.00
Swarm hash generated		1.00	1.00	0.00
Function arguments (filled)	address	1.00	0.99	0.01
	bytes32	1.00	0.98	0.10
Constructor arguments (filled)	address	1.00	1.00	0.00
r	to user	1.00	1.00	0.00
	to function	1.00	1.00	0.00
	contract	1.00	1.00	0.00
s	to user	1.00	1.00	0.00
	to function	1.00	1.00	0.00
	contract	1.00	1.00	0.00
Transaction id	to user	1.00	1.00	0.00
	to function	1.00	1.00	0.00
	contract	1.00	1.00	0.00
Public key	to user	1.00	1.00	0.00
	to function	1.00	1.00	0.00
	contract	1.00	1.00	0.00

TABLE 8. Notation. Cost and capacity-related symbols (left). Cost magnitudes (right).

Symbol	Description	Symbol	Description	Cost in gas (see [25])
T_{Nc}	Transaction nonce	C_{Tr}	Baseline transaction cost	21,000
$\|S_0\|$	Amount of bytes '0' in the embedded message	C_{Ct}	Contract creation cost	32,000
$\|S_{n0}\|$	Amount of bytes not '0' in the embedded message	C_{CtCd}	Cost per byte of contract code	200
$\|B_{Ct}\|$	Amount of bytes in contract code	C_{B0}	Cost per byte '0' of data or code	4
$\|B_{Ct0}\|$	Amount of bytes '0' in contract code	C_{Bn0}	Cost per byte not '0' of data or code	68
$\|B_{Ctn0}\|$	Amount of bytes not '0' in contract code	C_F	Cost per operation in a function	Variable
S	Secret to embed	C_{St}	Cost per contract storage	20,000
LBB	Limited By Balance			
$\|S\|$	Length of the secret			
$LBGL$	Limited By Gas Limit			
ML	Memory Limit			
ArL	Argument-related limit			
$\|IdF_0\|$	Amount of bytes '0' of function identifier			
$\|IdF_{n0}\|$	Amount of bytes not '0' of function identifier			
N_F	Number of operations in a function			
StL	Storage Limit			

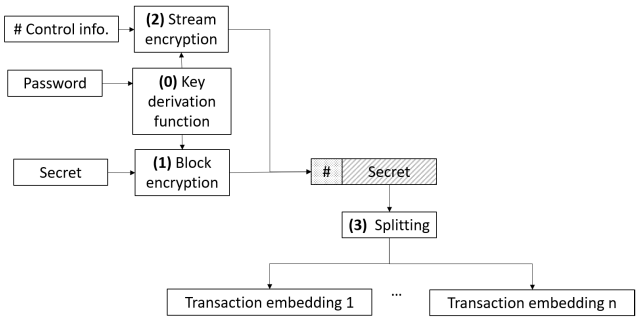

FIGURE 3. Secret preparation process.

distinguish between the secret itself and padding information. Moreover, additional data should be included for *Executable bytecode* (as explained in Section V-B2). Secondly, control data is also encrypted but with a stream cipher to keep the resulting size at a minimum (step 2). To randomize the output, the nonce from the last existing sender's transaction is also taken as input for this cipher. Finally, the secret is split if it exceeds the capacity of the transaction fields at stake (step 3).

2) DATA HIDING

For the sake of clarity, the description of the hiding process is divided into three main blocks, namely addresses, transaction information and smart contract data.

a: IN ADDRESSES

The three types of addresses (namely *Sender, Receiver* and *Contract* ones) can be modified in all cases, thus S1 strategy is applied. However, the required computational effort is dramatically different.

TABLE 9. Embedding strategy per selected field.

		S1 (full field)	S2 (top values)	S3 (pattern-based)	S4 (instruction encoding)
Addresses	Sender	x			
	Receiver	x			
	Contract	x			
Transaction info	Value			x	
	Gas limit (to contract only)		x		
	Gas price		x		
	Signature: r	x			
	Signature: s	x			
	Identifier	x			
	Sender public key	x			
	Function args: type uint256			x	
	Function args: type address	x			
	Function args: type bytes32	x			
Smart contract info	Swarm hash	x			
	Bytecode: PUSH1 values		x		
	Bytecode instructions				x
	Constructor args: type uint256			x	
	Constructor args: type address	x			

Recalling that the *Sender address* is the hash of a public key, the embedding process is limited by the computation of a valid inverse (i.e., private key) according to the cryptographic algorithm at stake (in particular, secp256k1 [25], [35]). Consequently, embedding data in this field involves a trial-and-error procedure.

A similar situation happens with *Contract addresses*. Since they are computed considering the number of transactions sent by the contract creator (recall Section II-B2), a trial-and-error process is carried out to find a suitable number.

On the contrary, the *Receiver address* is not under any restriction. Therefore, it can be modified at will.

b: IN TRANSACTION INFORMATION

Capacity and effort to do the embedding varies greatly among fields.

The *Value* field can be used considering its underlying patterns (strategy S3). However, it is limited by *LBB* as the secret is represented as the payment amount. In this case, only values of length 17 and 10 ending zeros will be considered as a trade-off between capacity and cost. Note that value to functions and contracts is discarded because the top 2 values (though for simplicity not presented in Table 1) represent more than 90% of the sample. Thus, it would allow a very small capacity.

On the other hand, the most prominent *Gas limit* and *Gas price* values (strategy S2) are considered for representing the secret. In this case, their use is bounded by *LBB*, and also by *LBGL* in *Gas limit*. These limitations depend on the sending account and the Ethereum blockchain, respectively. In the same line as *Value field* to functions and contracts, *Gas limit* to users is discarded because the top 2 values cover more than 62% of the sample.

As opposed to the previous field, signature values *r* and *s* and the *Sender public key* can be used in full (strategy S1). Furthermore, there is no technical limitation for the secret. However, a trial-and-error process must be followed to find the right cryptographic materials and produce a value that represents the fragment of the secret at stake.

c: IN SMART CONTRACTS

Depending on the field, a different embedding strategy is used, specially when bytecode is at stake.

Swarm hash field can be used in full (strategy S1) and with no limitations, since block scanners such as Etherscan do not currently check its value.

Function arguments appear within function calls or in a *Contract constructor*. Each function receives a different number of arguments and of varied types. In practice, the capacity is limited by *LBB*, *LBGL* and the technical limit for each argument type (called *ArL*). For instance, uint256 corresponds to 32 bytes and uint8 to 1 byte [36]. As it was stated in Section IV-A, only uint256, address and bytes32 types are used to embed information in *Function arguments* and uint256 and address types for *Constructors arguments*. In address and bytes32 types the whole capacity (S1) is used, while uint256 type follows a pattern (strategy S3).

With respect to the bytecode, there are two limitations in this regard – the code should look like a valid set of instructions and it has to be well-formed. In particular, two alternatives can be chosen – including instructions to represent the secret in an unreachable part of the code (called *Non-executable bytecode*) or in a reachable one (called *Executable bytecode*). Thus, *Non-executable bytecode* is placed between the JUMP-JUMPDEST block and the STOP/ASSERT instruction (recall Section II-B2). The code added in that region is never executed by any function of the contract. However, this should look like a legitimate JUMP-JUMPDEST block, so starting and ending instructions should follow the regular distribution.

The second way, *Executable bytecode*, involves including instructions in the JUMP-JUMPDEST block. It requires

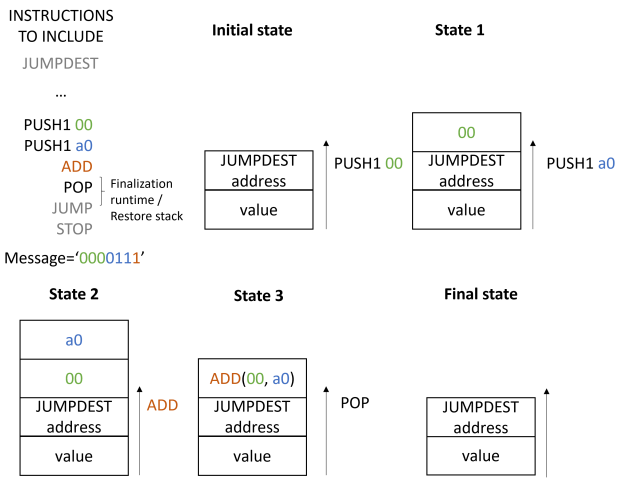

FIGURE 4. Embedding process in executable bytecode.

managing instructions carefully to keep the state of the stack and cause a failure. Therefore, the stack should be correctly restored.

In order to encode the secret, two strategies are followed. On the one hand, the choice of instructions (strategy S4) – the 20 most used opcodes (Table 13 in the Appendix) are divided in a couple of sets, one to represent 0 and another to represent 1. Thus, one opcode is chosen on a random weighted way. On the other hand, the argument of PUSH1 follows strategy S2.

In both cases, the capacity of the bytecode is limited in practice by EVM's total memory (*ML*), as well as *LBB* and *LBGL*. Moreover, the amount of instructions to be inserted is limited by the usual size of the JUMP-JUMPDEST block (9 and 13 instructions, recall Section IV-A). It must be noted that in the *Executable bytecode* case some instructions are needed to restore the stack. Therefore, they do not convey the secret themselves. As a result, in the *Executable bytecode* the capacity is limited by the number of secret-related opcodes applied. They are all instructions except for PUSH1, used to control the stack (if any), and the final POPs at stake. However, arguments of PUSH1 instructions are still used to embed information. Since the amount of secret-related instructions is not known in advance by the receiver, such information should be included as control data. Fig. 4 illustrates the process–colored instructions represent the secret, and the stack is properly managed to keep the execution of the bytecode. The secret message corresponds to "00000111", such that "000" is encoded with PUSH1 00, "011" with PUSH1 a0 and the last "1" with ADD operation. Then, after the initial state of the stack, PUSH1 00 is pushed to the stack (State 1), then PUSH1 a0 is pushed (State 2) and thirdly ADD (State 3). Finally, the stack should be restored by POPing all elements (Final state).

By contrast, in the *Non-executable bytecode*, all instructions are secret-related, which also includes PUSH1 arguments. No extra information is required as the message is inserted in the new JUMP-JUMPDEST block.

C. REVEALING MECHANISM

This process is analogous to the embedding one but in reverse order. Firstly, hidden data is extracted considering the field at stake. Secondly, control information is decrypted to delimit the message appropriately. Finally, the decryption is enforced.

However, one significant difference regarding the embedding procedure is that extraction does not require trial-and-error procedures. However, there is a performance overhead if the secret is to be revealed immediately (i.e., the sabotage case, recall Section I). In this situation, the receiver has to wait until all transactions containing the secret are included (after mining) in the blockchain.

VI. EVALUATION

The evaluation of the proposed mechanism is performed from a theoretical and a practical point of view. Firstly, the compliance of established goals is analysed (Section VI-A). Secondly, an experimental analysis has been carried out to determine the actual cost and time required to hide a secret per Ethereum transaction field (Section VI-B).

A. GOALS COMPLIANCE

Table 10 summarizes the analysis on the imposed goals per Ethereum field, whose compliance is discussed in the following sections.

1) STEALTHINESS

The type of attacker in terms of stealthiness to which each field is resistent is depicted in Table 10. Since the secret has been tailored to be disguised as normal values for each field (Section IV), almost all fields pass unnoticed to both BE and AE attackers, as there are no hints they might leverage on. For example, the *Swarm hash* has been proved to be random enough to be used in full and AE would need to have the original contract with the same file name to verify it, though there are situations with certain limitations (* is applied). In case of *Gas Limit* field, the study shows that it does not always match with the spent gas in the transaction and then, an attacker could have suspicions. Moreover, in *Executable bytecode*, the attacker should debug and understand that some of the instructions are really "dummy" code tailored as legitimate one but it is considered tedious and not really worthy. Nonetheless, there are a couple of exceptions in which just a single type of attacker applied. AE would notice some deviations from normality in *Non-executable bytecode*, as this code is never executed and could be more easily debugged.

2) SIMPLICITY

The proposed mechanism achieves simplicity as long as there is no special requirement to embed secret information in any of the fields. However, the computational effort varies among fields. Most of them, marked as $O(1)$, only require

TABLE 10. Goals assessment per Ethereum transaction field. (*) means conditional achievement.

	Field / Element	Subfield	Cost (gas)	Max capacity per transaction (bits)	Stealthiness (BE, AE, ALL)	Simplicity	Embedding computational cost	Secret integrity (IA)
Addresses	**Sender**		TransCost(1,0,0)	160	ALL	√	O(PoW)	√
	Receiver		TransCost(1,0,0)	160	ALL	√	O(1)	√
	Contract		TransCost(T_{Nc}-1,0,0) + ContCost(1, $\|B_{Ct}\|$, $\|B_{Ct0}\|$, $\|B_{Ctn0}\|$)	160	ALL	√	O(PoW)	√
Transaction info	**Value**	to user	C_{Tr} + S	22	ALL	√	O(1)	√
	Gas limit	to contract	TransCost(1,*,*) or ContCost(1, $\|B_{Ct}\|$, $\|B_{Ct0}\|$, $\|B_{Ctn0}\|$)	3	ALL*	√	O(1)	√
	Gas price	ALL		4	ALL	√	O(1)	√
	Nonce	-	-	0	-	-	-	-
	Signature: V	-	-	0	-	-	-	-
	Signature: r	ALL	TransCost(1,*,*) or ContCost(1, $\|B_{Ct}\|$, $\|B_{Ct0}\|$, $\|B_{Ctn0}\|$)	256	ALL	√	O(PoW)	√
	Signature: s	ALL		256	ALL	√	O(PoW)	√
	Identifier	ALL		256	ALL	√	O(PoW)	√
	Sender Public Key	ALL	TransCost(1,0,0)	512	ALL	√	O(PoW)	√
	Function args	address	TransCost(1,$\|IdF_0\|$+$\|S_0\|$, $\|IdF_{n0}\|$+$\|S_{n0}\|$) + C_F* N_F	160	ALL	√	O(1)	√
		bytes32		256	ALL	√	O(1)	√
		uint256		12	ALL	√	O(1)	√
Smart contract info (contained in **data** field)	**Swarm hash**		ContCost(1, $\|B_{Ct}\|$, $\|B_{Ct0}\|$, $\|B_{Ctn0}\|$)	256	ALL	√	O(1)	√
	Bytecode	Non-executable	ContCost(1, $\|B_{Ct}\|$, $\|B_{Ct0}\|$ + $\|S_0\|$, $\|B_{Ctn0}\|$ + $\|S_{n0}\|$)	46	AE	√	O(1)	√
		Executable		33	ALL*	√	O(1)	√
	Constructor args	address	ContCost(1, $\|B_{Ct}\|$, $\|B_{Ct0}\|$, $\|B_{Ctn0}\|$) + C_{St} * ($\|S\|$/32)	160	ALL	√	O(1)	√
		uint256		26	ALL	√	O(1)	√

one operation to hide information. Thus, the original contents of the field are replaced (partially or in full) by the secret. However, *Sender address*, *Contract address*, *Hash*, signature fields and *Public key* involve several repetitive operations until the right value is found. Since the required effort is analogous to solving proof-of-work computational puzzles [37], they are marked as *O*(*PoW*).

3) EFFICIENCY

Though time efficiency will be studied in Section VI-B3, efficiency in terms of the amount of sent information is studied herein. For this purpose, the size of the secret has to be higher than the data to be privately shared with the receiver beforehand – otherwise, the mechanism would not be needed. The data shared with the receiver is formed by 388 bits, namely transaction identifier (256 bits), encryption key (64 bits), nonce (64 bits) and fields to hide the secret (4 bits). Note that the use of functions and the constructor in smart contracts may require to know the ABI code but this is not necessary if such contracts are verified.

Efficiency of the amount of sent information, called Information Efficiency (IE), depends on the secret size $\|S\|$ and it is calculated following Equation 3.

$$IE = \frac{\|S\|}{388} \tag{3}$$

The system is efficient as long as IE $>$ 1. It must be noted that the individual capacity of each field per transaction would not meet this condition. However, *Zephyrus* enables using a series of transactions to hide a secret. In this way, as explained in Section VI-B2, in our experiments secrets range from 400 to 40,000 bits, thus leading to $1.90 < IE < 315.05$. Moreover, an analysis per field is shown in Table 11.

TABLE 11. Maximum secret size, cost and IE per field in our experiments.

Field	Max secret size (bits)	Additional cost (ether)	Cost \ Fee (ether)	Cost \ Fee (USD)	IE
Receiver address	40,760	-	0.005355	$ 1.64	105.05
Swarm hash	65,240	-	0.2631	$ 80.44	168.14
Gas Price	1,000	0.09	-	$ 27.51	2.58
Value	5,560	8.3137	-	$ 2,542	14.33
Gas limit	736	-	0.2575	$ 78.73	1.90
Function arguments	43,824	-	0.01073	$ 3.28	112.95
Constructor arguments	122,240	-	0.4490	$ 137.28	315.05
Non-executable bytecode	4,496	-	0.2215	$ 67.72	11.59

4) COST

Embedding information in each of the fields has an associated cost. It is related to the fees required for sending information to Ethereum's blockchain. Particularly, sending transactions or deploying contracts have an associated cost, which can be measured according to Ethereum's documentation [25]. These costs are described by Equations 4 and 5 for transactions and contracts, respectively. In both cases, they have a fixed cost per operation and a variable part depending on the amount of data at stake. Note that the notation is the one introduced in Table 8. Moreover, constant values are as

follows: a is the number of transactions or contracts required to send the message; b and c are the number of bytes 0 and 1 respectively; and d is the amount of bytes of the contract code.

$$TransCost(a, b, c) = a \times C_{Tr} + b \times C_{B0} + c \times C_{Bn0} \quad (4)$$

$$ContCost(a, b, c, d) = a \times (C_{Tr} + C_{Ct}) + b \times C_{B0} + c \times C_{Bn0} + d \times C_{CtCd} \quad (5)$$

Table 10 shows the cost per field leveraging these equations. *Sender* and *Receiver addresses* only need to send a transaction, and no additional payload is required. Optionally, some Ether could be included in the value to look like a natural transaction. Regarding *Contract addresses*, apart from deploying the contract, it is necessary to send transactions so as to make the nonce value lead to the required address value. Other fields involve a transaction in which the variable part increases with the size of the secret. In some of them, such part is increased with some inherent costs, such as the name of the function at stake in the case of *Function arguments*. It should be noted that in some cases (e.g.,*Gas limit* or *Signature* and *Hash* fields) the sender might decide using a transaction to another user or a to a function in a contract or deploy a contract for embedding information. Last but not least, most contract-related fields involve deploying a contract, with some additions like the cost of storing information. To illustrate this discussion, Section VI-B3 describes the real costs incurred by each of these fields in real transactions.

5) SECRET INTEGRITY

The immutability property of Ethereum ensures that the secret embedded in most fields can always be recovered. In particular, even if the IA attacker creates any transaction, the secret message is not affected. Nevertheless, the only exception is the use of the contract storage.

B. EXPERIMENTAL STUDY

A proof of concept has been implemented to measure the time taken for the proposed mechanism, as well as its associated costs. The implementation is described in Section VI-B1. The description of the experimental settings is presented in Section VI-B2. Afterwards, the obtained results are presented in Section VI-B3.

1) PROOF OF CONCEPT

Zephyrus has been implemented in an open-source software tool available in Gitlab.[2] Through a command-line interaction, the user will be asked to provide the input required depending on the field at stake.

From a technical viewpoint, the tool has been developed in Python 3.5. For encryption purposes, AES in Counter (CTR) mode is applied for the secret and ChaCha20 for the control data. Encryption keys are derived by means of the Password-Based Key Derivation Function 2 (PBKDF2) algorithm [38]. Sender and receiver/s can agree on an AES password and a nonce in a initial stage and increase this one per message transmission. ChaCha20 password is derived from the AES one and it changes per transaction to avoid patterns in encrypted information. Besides, the sender may send a message in different transactions and smart contracts to different receivers.

Regarding network connection options, *Zephyrus* is able to connect to a local node by interacting with the Go Ethereum Client (geth [39]), or to a Infura [40] node, so neither the sender or receiver/s need to have the blockchain synchronized, saving space and computational power.

In this current version of the implementation, all $O(1)$ (recall Table 10) methods have been implemented, except for *Gas price* to functions and contracts, as it is significantly cheaper, and *Executable bytecode*, as it allows embedding fewer information. Besides, only one field can be used for each secret.

2) EXPERIMENTAL SETTINGS

Experiments have been run in a AMD FX-8370 8-Core processor equipped with Debian 9 OS with 16 Gb. of RAM. Note that the mining process is not part of our system and *Zephyrus* would work in any computer with similar characteristics and once installed Python 3.5 and used libraries (described in the prototype implementation[2]). Concerning the blockchain, Ropsten [41] has been used. Addresses have been provided with enough funds to carry out all transactions and Infura nodes have been used to connect to the blockchain.

To ensure the validity of our results, each embedding and revealing operation has been carried out 5 times. Afterwards, the arithmetic mean has been computed.

Concerning applied elements, the secret is a random set of 400, 2,000, 4,000, 8,000, 24,000 and 40,000 bits. On the other hand, the cover is different depending on the field at stake. In case of regular transaction fields, a tailored transaction has been created. In contract-related fields, different smart contracts are at stake. For the *Swarm hash* field and the *Non-executable bytecode* one, the same contract has been used [42]. Regarding *Constructor arguments*, another contract with a constructor function has been applied [43]. Most common smart contracts in Ethereum use ERC-20 tokens, the most popular ERC-20 token by market capitalization [44] has been used to test *Function arguments*. For the *Gas limit* field the contract used is [45]. The gas limit for the rest of the fields has been set up according to a method available in Ethereum which estimates the necessary gas to complete the transaction [46].

Strategies and values analyzed in Section IV has been used and function ''approve'', selected from [44], is applied to test *Function arguments*. For the sake of a balance between computational cost and time, the experiment allows a maximum of 255 transactions per field.

[2]https://gitlab.com/MarGA2503/zephyrus.git

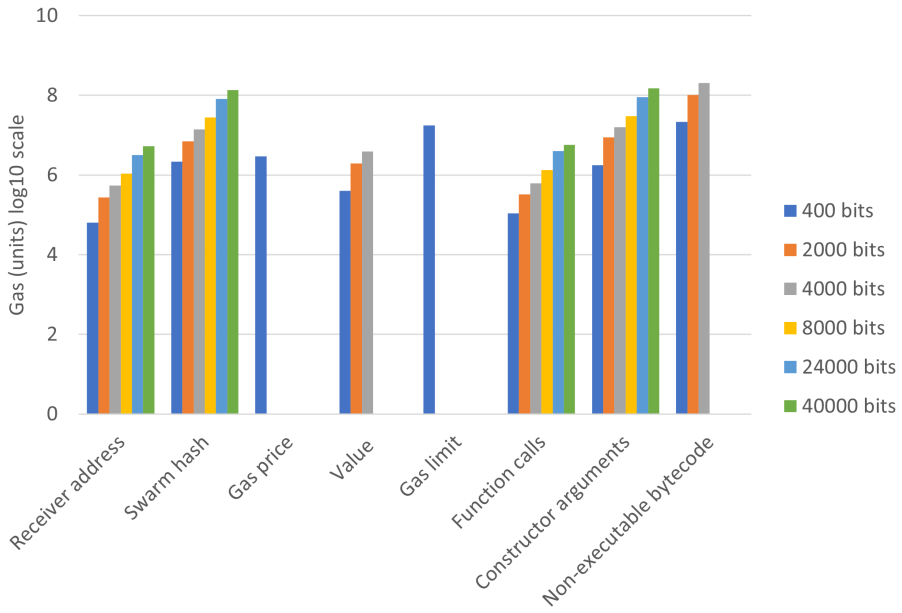

FIGURE 5. Gas cost per field (log scale).

The use existing contracts provides realism to our results – *Zephyrus* could be applied immediately leveraging the current Ethereum contents.

3) RESULTS

Concerning the actual costs incurred by *Zephyrus*, Fig. 5 shows the gas cost per field depending on the secret size in bits. Note that when a contract deployment is at stake the amount of gas is affected by the size of the contract. Similarly, the capacity of the constructor and function arguments field depends on the number and types of arguments. As expected, the cost increases with the secret size, but depending on the field more or less data can be embedded. In the case of *Gas price* and *Gas limit* 400 bits can be embedded, as they are fields with embedding restrictions. It costs 6.5 and 7.2 gas units respectively. Indeed, the best alternative from the cost point of view is the use of the *Receiver address* and *Function calls* – their cost is 6.7 gas units in both cases when embedding 40 Kbits and 4.8 and 5.0 for 400 bits.

Table 10 depicts the maximum capacity per individual transaction, identifying fields in which up to 256 and 512 bits can be embedded. Moreover, Table 11 shows the maximum capacity of each field and the actual cost in USD, along with their IE ratio for all carried out transactions (255 in this experiment). For this purpose, the average price [47] of 1 Ether in 2020 (1 Ether = $ 305.76) has been considered, taking the cheapest gas price (1 Gwei) [48]. Note that embedding into *Value* and *Gas price* fields involves an additional cost.

The most efficient field, regarding stealthiness and cost is to embed a message in *Function arguments* allowing up to 43,824 bits for $ 3.28. However, inserting data in the *Receiver address* also provides great results. In relation to the quantity of embedded data, *Constructor arguments* method is the best with the tested contract. The most expensive one, *Value*, allows 5,560 bits for around $ 2,542, as real Ether is transferred. Besides, in terms of IE, results show that the system is efficient even using a single field in all cases. Nevertheless, significant differences exist between them like *Gas price* or *Non-executable bytecode*.

With respect to the time taken by *Zephyrus* and linked to the efficiency goal, it can be divided into three main parts, namely embedding time, network management time and revealing time. Note that network management time corresponds to sending the transaction to the blockchain, its retrieval and the mining time. Although such management is out of the scope of *Zephyrus*, it will be unavoidably required for its usage in the real world.

The embedding (E) and revealing (R) time for every field depending on the secret's size is presented in Fig. 6. Embedding involves the encryption of the secret and the preparation of required transactions. Conversely, revealing requires extracting the secret and decrypting it afterwards. The time of encryption and decryption is quite similar for all fields, 4.5×10^{-3} *s* on average for both operations. As expected, the size of the message directly affects the time of embedding and revealing but not to a great extent. However, the secret's size slightly affects the time spent in the revealing, as more transactions are managed and more checking operations are required. Though the time differs between embedding and revealing, it is minimum considering the applied scale (max. 65 *s*). This is in line with expectations, as both operations are similar but applied in reverse order. Also noticeable is the fact that including data in the *Value* is the toughest operation because of the amount of required numbers according to

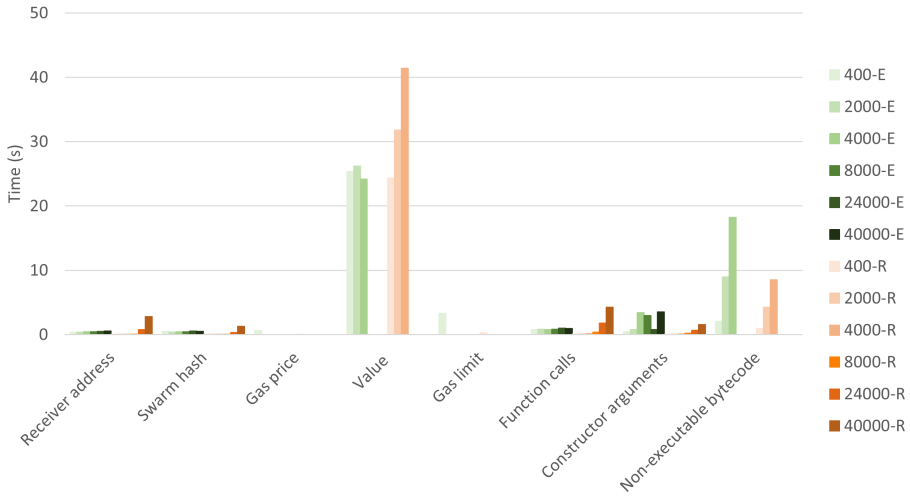

FIGURE 6. Embedding and revealing time per field.

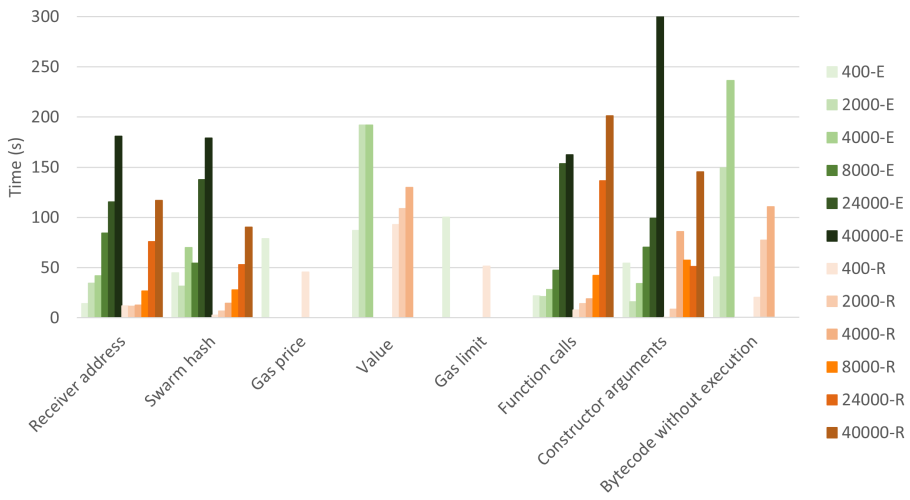

FIGURE 7. Network management time per field.

the selected pattern (recall Section IV-A), it takes 24.2 *s* for embedding and 41.4 *s* for revealing 4 Kbits, which is the worst case. As a trade-off between efficiency and stealthiness, the best choice is the use of *Function calls*, as they allow embedding 40 Kbits in 0.93 *s* and retrieving them in 4.3 *s*. On the contrary, just looking for time restrictions, *Receiver address* provides meaningful results as embedding and revealing time is quite reduced even for 40 Kbits – 0.57 *s* and 2.8 *s* respectively.

For illustration purposes, though it depends on the network status, a complete overview of the steganographic process is analysed including network management time. Fig. 7 presents the time of embedding (E) and revealing (R) data. E includes the mining time and that of sending data to the network, and R includes the retrieving time from the network. When contracts are not involved, for instance, mining time for *Receiver address* and 40 Kbits takes around 50 *s* on average, whereas the time to connect and send the information to Infura is around 130 *s*, 180.1 *s* for the whole embedding process. However, in those cases involving a contract deployment, for example the *Swarm hash*, mining times are usually higher, 84 *s*, and 94 *s* to send the transaction to the network, leading to a total embedding time of 178.1 *s*. Regarding revealing time, it depends on the amount of transactions at stake and how they are mined (i.e., same or different block and distance between them). For instance, transactions with smaller sizes mined in the same blocks take less time to be retrieved than larger transactions in different blocks. In the worst case, the revealing time of *Function calls* is 201.1 *s* for 40 Kbits. In the best case, it is 90.4 *s* for *Swarm hash* and the same amount of transmitted data.

This analysis shows the feasibility of using *Zephyrus* for building covert channels. In sum, on average, the embedding and revealing procedures take 8.07 *s*, while network management takes 154.23 *s* and thus, 162.3 *s* (2.71 min) in total.

VII. RELATED WORK

Several works combine the concepts of steganography and blockchain. For instance, in [49] secrets are embedded in images which are later shared through a blockchain and a file system. On the other hand, in [50] each transaction is divided in two parts which are hidden in videos. However, here we focus on the use of the blockchain itself for steganographic purposes – the secret is directly hidden in blockchain data. In this regard, recent results show that it has not been detected in Bitcoin [51], although most proposals focus on this cryptocurrency. This is the case of Ken Shirriff's blog [12], which presents some basic steganographic techniques. They use encoding (e.g. hexadecimal, base64, etc.) to hide different messages, texts or files but any of them is really sophisticated. In this way, decoding is the only process required to access secret messages. One example is the use of the receiver address to store data, namely the inclusion of an image of Nelson Mandela and a tribute text. Many transactions were generated to store all information, in hexadecimal, into receiver addresses. Each transaction can contain 20 bytes of data. The use of an arbitrary field of 100 bytes or more, in the coinbase block (e.g. the initial block of the chain) has been also applied to hide data, namely a political sentence or some prayer names. Another example pointed out is the concealment of data in the hash of the public key script (P2PKH), used to verify performed transactions. Dan Kaminsky used this method to embed a tribute to the cryptographer Len Sassaman. It is also common to replace keys in a multi-signature transaction, in these case the 1-3 type. A final example is the use of Nulldata transactions, in which the OP_RETURN (Null data transaction) field is applied for invalid transactions. This technique has been used to store lyrics of Rick Astley. However, though OP_RETURN can be used once per transaction, its use should be limited to not raise suspicions.

Also with the focus on Bitcoin, [52] presents different fields to hide messages without the use of encryption. Data is included in the timestamp (nLockTime) and in the sequence number, but a combination of multi-signature (1-12) inputs and outputs, transaction amount and Nulldata transactions were finally used. In the case of signatures, the secret message is embedded when a valid signature is computed; and in case of the transaction amount, the budget is split in multiple transactions based on a combinatorial composition.

In relation to this cryptocurrency, A. Sward *et al.* [13] review the different existing data insertion methods like including information in the public key in a Pay to Public Key (P2PK) transaction, or in the hash of the public key in a Pay to Script Hash (P2SH), both methods using the ScriptPubKey of a transaction. Regarding the ScriptSig, P2SH script could also be used, either inserting data on the Reedem part of the script or in the Data Input part.

R. Matzutt *et al.* [14] analyse the impact of inserting content on Bitcoin, explaining different methods and naming some of the existing tools that are able to perform this action.

On the other hand, R. Recabarren at al. [16], propose Thitonious, an anti-censorship Bitcoin tool, using the scriptSig of a P2SH multisignature transaction by inserting the message on the 28 most significant bytes. Thitonious allows users to access free-altruistic content published in clear text, or pay for on-demand content. In this case, the information is encrypted.

M.D. Sleiman *et al.* [53] propose inserting text in the transaction amount of Bitcoin by using an arithmetic encoding, which provides an space of eight characters (seven plus termination symbol in an ideal case) that should be lower-case English characters, spaces or periods. Multiple transactions can be used to insert larger messages.

Tian *et al.* [54] use the OP_RETURN and Private key in Bitcoin transactions. The private key (32 bytes) is used to embed the message while the OP_RETURN is used in order to change the labels between messages which are generated in a dynamic way and are stadistically indistinguishable from normal transactions. Data is encrypted before the embedding process.

Fionov [55] reviews briefly the existing covert channel in Bitcoin. Futhermore, he proposes a method based on permutations of transaction outputs, inputs and values (payments), whose number affects capacity. The secret message is encrypted. However, just one transaction is used to justify the number of inputs and outputs in this study and further analysis is highlighted as a necessity.

Torki *et al.* [56] propose a pair of algorithms to embed data in blockchains. One of them has high embedding capacity as secret data is embedded in transactions' data and the other one has medium capacity embedding data in sender addresses. Though both algorithms seem to be general, they are directly related to Bitcoin transactions.

D. Frkat *et al.* [57] propose ChainChannels, a scheme to send hidden information to bots within ECDSA signatures. The sender introduces the message in the random number used to generate the signature and the receiver needs to know the signature private key to retrieve the message. Besides, Bitcoin network is used for evaluation purposes.

By contrast, Ethereum is used by Basuki *et al.* [58], working with image steganography. Instructions for recovering the secret within the image are included in the timestamp of a smart-contract, allowing 29 bits of capacity. Then, the image is stored in a web server and clear text data is stored in the blockchain for the secret recovery.

Gao *et al.* [59] use kleptographic algorithms in order to identify which transactions have secret information. Even though different fields of Bitcoin and Ethereum blockchains are mentioned, most of them are not studied. According to a proof of concept, just Ethereum OP_RETURN and data

TABLE 12. Related work summary.

Reference	Cryptocurrency	Method	Equivalent in Ethereum	Max capacity (bits)	Stealthiness (BE, AE, ALL)	Simplicity	Embedding computational cost	Secret integrity (IA)	Comparision of embedded data with normal content in blockchain
[12]	Bitcoin	Receiver address(in hex)	Receiver address	160	BE*	√	O(1)	√	X
	Bitcoin	Block coinbase	Block Extra Data	Up to 800	BE*	-	O(1)	√	X
	Bitcoin	ScriptPubKey:paytopubkeyhash	Smart contract**	160	BE*	√	O(1)	√	X
	Bitcoin	ScriptPubKey:paytoscripthash-multisig	Smart contract**	1560	BE*	√	O(1)	√	X
	Bitcoin	Null data transaction	Transaction data	320	BE*	√	O(1)	√	X
[52]	Bitcoin	ScriptSig	Smart contract**	8	ALL*	√	O(PoW)	√	X
	Bitcoin	ScriptPubKey:paytoscripthash-multisig	Smart contract**	2766	ALL*	√	O(PoW)	√	X
	Bitcoin	Transaction amount	Transaction value	64	ALL*	√	O(1)	√	X
	Bitcoin	nLockTime	-	32	ALL*	√	O(1)	√	X
	Bitcoin	Sequence number	-	32	ALL*	√	O(1)	√	X
[13]	Bitcoin	ScriptPubKey:paytopublickey	Smart contract**	520/264	BE,AE	√	O(1)	√	X
	Bitcoin	ScriptPubKey:paytoscripthash	Smart contract**	160	BE,AE	√	O(1)	√	X
	Bitcoin	ScriptSign(Redeem script):paytoscripthash	Smart contract**	4136	BE,AE	√	O(1)	√	X
	Bitcoin	ScriptSign(Data input):DataDropwithoutSignature	Smart contract**	13040	BE,AE	√	O(1)	√	X
	Bitcoin	ScriptSign(Data input):DataDropwithSignature	Smart contract**	12232	BE,AE	√	O(1)	√	X
	Bitcoin	ScriptSign(Data input):DataHashwithoutSignature	Smart contract**	12480	BE,AE	√	O(1)	√	X
	Bitcoin	ScriptSign(Data input):DataHashwithSignature	Transaction dataSmart contract**	11688	BE,AE	√	O(1)	√	X
[16]	Bitcoin	ScriptSign:multisignature	Smart contract**	448	BE,AE/ALL	√	O(PoW)	√	X
[53]	Bitcoin	Transaction amount	Transaction value	64	BE,AE	√	O(PoW)	√	X
[54]	Bitcoin	Sender private key	Sender private key	256	ALL	√	O(1)	√	√
[55]	Bitcoin	Transaction inputs + outputs + value	Transaction sender address + receiver + value	N/A	BE/ALL (depends of numbers per field)	√	O(1)	√	X
[56]	Bitcoin	Receiver address + ScriptPubKey:paytopublickeyhash + ScriptSign(Redeemscript):paytoscripthash	Transaction receiver address + Smart contract**	81.9	ALL	√	$O(2^{m_i})$	√	X
[58]	Ethereum	Timestamp in smart-contract	Smart-contract	29	BE	√	O(1)	√	X
[17]	Ethereum	Value field	Value field	30	AE	√	O(1)	√	√
[60]	Any	Receiver address	Receiver address	1	ALL	√	O(1)	√	X
[57]	ECDSA-based	Nonce of the signature	Sender address/Sender public key	256	ALL	√	O(1)	√	X
[61]	CryptoNote-based-based	Signature	Sender address/Sender public key	504	ALL	√	O(1)	√	X
[62]	Any	Sender address	Sender address	Blockchain depending	ALL	-	O(1)	√	X
[59]	Bitcoin	Null data transaction	Transaction data	320	BE*	√	O(1)	√	X
	Ethereum	Transaction data	Transaction data	320	BE*	√	O(1)	√	X
Zephyrus	Ethereum	Sender address Receiver address Contract address Transaction data Signature R Signature S Transaction identifier Sender Public Key Gas price Transaction value Gas Limit Function calls Swarm hash Constructor arguments Executable bytecode Non-executable bytecode	-	Several values (cf. Table 11))	AE/ALL	√	O(1)/ O(PoW)	√	√

** simplified version of smart contracts

* simple hidden technique

field are used for steganographic purposes with 80 bytes of capacity, embedding encrypted data.

Ethereum is also used in Liu *et al.* [17]. They only use the Value field. They propose three different ways of including

information with a maximum of 1, 30 and 15 bits per transaction.

Some other proposals are applicable to a different range of blockhains and cryptocurrencies. J. Partala [60] suggests a method for securely embedding covert messages into a general blockchain. The sender generates payments and the secret message is embedded, bit by bit, in the LSB of each receiver address. Then, the sender and the receiver have to order and collect bits accordingly.

N. Alsalami *et al.* [61] propose the use of CryptoNote framework, applied in cryptocurrencies like Monero, by embedding a message in the ring signature's random numbers.

Finally, Xu *et al.* [62] embed secret information in the blockchain using the sender address of preselected transactions according to a certain key. Selected transactions are arranged in a certain way in order to carry the secret message. The amount of data that this method is able to transmit depends on the quantity of transactions that can fit in a block and the number of different senders. However, the sender of the secret message should mine the block and the receiver needs the key to retrieve it.

A. SUMMARY OF RELATED WORK

Table 12 presents a summary of this analysis describing the applied cryptocurrency, the blockchain element in which data is hidden, the equivalent element in Ethereum, the maximum capacity of secret data per transaction, the stealthiness, the simplicity, the embedding computational cost and the secret integrity of each technique. Results show that only three proposals study the use of steganography in Ethereum. Moreover, as opposed to this paper, none of these works considers all data fields. Furthermore, [54] and [17] are the only ones studying the proposed mechanism in relation to the blockchain actual content, Bitcoin and Ethereum in particular.

In the case of [17], authors focus on the *Value field* by characterizing its entropy and length. However, their approach does not consider the frequent patterns appearing in this field, as we have discovered in our study. Moreover, some of their proposed schemes require the message starts by 1, which may be of interest for an attacker.

Note that there are elements marked as equivalent to smart contracts, but they correspond to an extremely simplified version of them. Capacity is expressed in terms of a single input and output in a Bitcoin transaction, though Bitcoin transactions may have multiple one. In Bitcoin, the maximum size of inputs is of 1,650 bytes, each element in the stack can have a maximum size of 520 bytes, whereas that of the whole transaction should not exceed 100,000 bytes. Ethereum, on the other hand, works per transaction. It means that each action in the chain requires a different transaction. However, the maximum capacity per single input/output in Bitcoin is comparable with *Zephyrus* in many cases.

On the other hand, [57], [60]–[62] could be used in Ethereum too. In the case of [53], Bitcoin could be replaced by Ether, but it should be noticed that this method only allows English text messages, whereas *Zephyrus* can transmit any binary information.

On the other hand, stealthiness is the most remarkable issue, except for [54]–[57], [59]–[62] and [16] when paying for content, all techniques offer a limited protection against BE. Approaches in [52], [58] embed data in clear text and [12] applies encoding. *Zephyrus* allows users the exchange of encrypted (or clear) messages without any retrieving cost, unlike Thitonious which is a commercial service for demand and encryted content. [54] uses dynamic labels in order to hide the communication, *Zephyrus* could also change the sender address for each transaction using the control and pre-shared information.

In terms of simplicity, almost all techniques can be used by regular users and just 'Block coinbase' [12] and [62] require to be a miner. Similarly, the computation cost of embedding is also analogous in most approaches (O(1)), except for [52] in which 'ScriptSig' and 'ScriptPubKey:scripthash-multisig' need to compute a valid key to make coins reedemable; [16] requires generating valid-looking quadratic residues; and [61] uses a non-lineal message retrieval process. Finally, secret integrity is achieved in all cases, thus being resistant against IA.

VIII. CONCLUSION AND FUTURE WORK

Ethereum permanent availability is an appealling feature to build covert communications on top of it. No previous work has considered all its fields for this purpose. In this paper, *Zephyrus*, a mechanism to hide information in Ethereum transactions has been proposed. An open-source implementation has been released to foster further research in this area. Our results show that some information can be concealed in most transaction fields while remaining stealthy if some limits are observed. Moreover, cost and time incurred have been characterized, supporting the real-world suitability of this proposal.

Future work will go towards the identification of the optimal fields to embed information considering time, cost and stealthiness restrictions all together. Furthermore, control structures can be adapted to efficiently support multi-field usage. Open and interactive channel communications can also be implemented, by letting *Zephyrus* scan the network and automatically retrieve content that fullfil certain characteristics. Lastly, adaptive steganographic techniques will be considered to improve capacity considering the existing Ethereum contents.

APPENDIX

The 20 most common opcodes and their frequency is depicted in Table 13.

TABLE 13. 20 most common opcodes in smart contracts.

Opcode	Frequency(%)
POP	23.38%
PUSH1	16.49%
SWAP1	7.10%
AND	6.13%
DUP2	4.48%
SWAP2	4.32%
SWAP3	3.86%
DUP1	3.69%
SLOAD	3.66%
EXP	2.90%
PUSH20	2.89%
SUB	2.71%
MSTORE	1.86%
ADD	1.49%
MLOAD	1.49%
DUP4	1.18%
SSTORE	1.15%
SWAP4	0.99%
NOT	0.93%
OR	0.90%

The top 5 number and quantity of instruction in the JUMP-JUMPDEST block is presented in Table 14.

TABLE 14. Top 5 number of instructions in the JUMP-JUMPDEST block.

# Instructions	Contracts
2	12234
9	7830
0	6863
1	5249
13	3974

REFERENCES

[1] M. Iansiti and K. R. Lakhani, "The truth about blockchain," *Harvard Bus. Rev.*, vol. 95, no. 1, pp. 118–127, 2017.

[2] V. Dhillon, D. Metcalf, and M. Hooper, "Blockchain in healthcare," in *Blockchain Enabled Applications*. Berkeley, CA, USA: Apress, 2021, pp. 201–220.

[3] S. F Fahmy, "Blockchain and its uses," in *Arab Academy for Science and Technology and Maritime Transport*. Cairo, Egypt: Sheraton, 2018.

[4] Z. Ma, M. Jiang, H. Gao, and Z. Wang, "Blockchain for digital rights management," *Future Gener. Comput. Syst.*, vol. 89, pp. 746–764, Dec. 2018.

[5] U. W. Chohan, "Cryptocurrencies: A brief thematic review," Tech. Rep., 2017.

[6] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2008, p. 21260.

[7] R. Bagshaw. *Top 10 Cryptocurrencies by Market Capitalisation*. Accessed: Apr. 2021. [Online]. Available: https://coinrivet.com/top-10-cryptocurrencies-by-market-capitalisation/%

[8] E. Kane. (Mar. 11, 2017). *Is Blockchain a General Purpose Technology?* [Online]. Available: https://ssrn.com/abstract=2932585

[9] S. Katzenbeisser and F. Petitcolas, *Information Hiding Techniques for Steganography and Digital Watermarking*. Norwood, MA, USA: Artech House, 2000.

[10] P. Cuff and L. Zhao, "Coordination using implicit communication," 2011, *arXiv:1108.3652*. [Online]. Available: http://arxiv.org/abs/1108.3652

[11] A. Abuadbba, I. Khalil, and M. Atiquzzaman, "Robust privacy preservation and authenticity of the collected data in cognitive radio network—Walsh–hadamard based steganographic approach," *Pervas. Mobile Comput.*, vol. 22, pp. 58–70, Sep. 2015.

[12] K. Shirriff. *Hidden Surprises in the Bitcoin Blockchain and how They are Stored*. Accessed: Apr. 2021. [Online]. Available: http://www.righto.com/2014/02/ascii-bernanke-wikileaks-photographs.ht%ml#ref6

[13] A. Sward, I. Vecna, and F. Stonedahl, "Data insertion in bitcoin's blockchain," *Ledger*, vol. 3, pp. 1–19, Apr. 2018.

[14] R. Matzutt, J. Hiller, M. Henze, J. H. Ziegeldorf, D. Müllmann, O. Hohlfeld, and K. Wehrle, "A quantitative analysis of the impact of arbitrary blockchain content on bitcoin," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2018, pp. 420–438.

[15] F. Ding, "Broadcasting steganography in the blockchain," in *Proc. 18th Int. Workshop Digit. Forensics Watermarking, (IWDW)*, vol. 12022, Chengdu, China: Springer, Nov. 2019, p. 256.

[16] R. Recabarren and B. Carbunar, "Tithonus: A bitcoin based censorship resilient system," *Proc. Privacy Enhancing Technol.*, vol. 2019, no. 1, pp. 68–86, Jan. 2019.

[17] S. Liu, Z. Fang, F. Gao, B. Koussainov, Z. Zhang, J. Liu, and L. Zhu, "Whispers on ethereum: Blockchain-based covert data embedding schemes," in *Proc. 2nd ACM Int. Symp. Blockchain Secure Crit. Infrastruct.*, Oct. 2020, pp. 171–179.

[18] L. Zhang, Z. Zhang, Z. Jin, Y. Su, and Z. Wang, "An approach of covert communication based on the Ethereum whisper protocol in blockchain," *Int. J. Intell. Syst.*, vol. 36, no. 2, pp. 962–996, Feb. 2021.

[19] draglet. *Smart Contracts Explained*. Accessed: Apr. 2021. [Online]. Available: https://www.draglet.com/blockchain-services/smart-contracts/

[20] A. Westfeld and A. Pfitzmann, "Attacks on steganographic systems," in *Proc. Int. Workshop Inf. Hiding*. Berlin, Germany: Springer, Sep. 1999, pp. 61–76.

[21] R. J. Anderson and F. A. P. Petitcolas, "On the limits of steganography," *IEEE J. Sel. Areas Commun.*, vol. 16, no. 4, pp. 474–481, May 1998.

[22] C.-C. Chang, J.-Y. Hsiao, and C.-S. Chan, "Finding optimal least-significant-bit substitution in image hiding by dynamic programming strategy," *Pattern Recognit.*, vol. 36, no. 7, pp. 1583–1595, Jul. 2003.

[23] Ethereum Foundation. *Ethereum. Blockchain Application Platform*. Accessed: Apr. 2021. [Online]. Available: https://www.ethereum.org/

[24] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *Proc. IEEE Int. Congr. Big Data (BigData Congr.)*, Jun. 2017, pp. 557–564.

[25] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Apr. 2014.

[26] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," White Paper, 2014, vol. 3, no. 37.

[27] *Abi Spec*. Accessed: Apr. 2021. [Online]. Available: https://solidity.readthedocs.io/en/develop/abi-spec.html Last

[28] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm," *J. Inf. Secur.*, vol. 1, no. 1, pp. 36–63, 2001.

[29] *Solidity Metadata Specification*. Accessed: Apr. 2021. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.10/metadata.html

[30] *Etherscan*. Accessed: Apr. 2021. [Online]. Available: https://etherscan.io/

[31] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, Jul./Oct. 1948.

[32] *Mining in Ethereum*. Accessed: Apr. 2021. [Online]. Available: https://github.com/ethereum/wiki/wiki/Mining

[33] Ethereum. (Jan. 2019). *Ethereum Wiki*. Accessed: Apr. 2021. [Online]. Available: https://github.com/ethereum/wiki/wiki/Design-Rationale#gas-and-fees

[34] H. Moriya. (May 2018). *How to Get Ethereum Block Gas Limit*. Accessed: Apr. 2021. [Online]. Available: https://medium.com/@piyopiyo/how-to-get-ethereum-block-gas-limit-eba2c8%f32ce

[35] V. Kobel. (2017). *Generating a Usable Ethereum Wallet and its Corresponding Keys*. Accessed: Apr. 2021. [Online]. Available: https://kobl.one/blog/create-full-ethereum-keypair-and-address/

[36] *Solidity Types*. Accessed: Apr. 2021. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.10/types.html

[37] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 3–16.

[38] B. Kaliski, *Password-Based Cryptography Specification*, document RFC 2898, 2000.

[39] *Geth*. Accessed: Apr. 2021. [Online]. Available: https://geth.ethereum.org/

[40] *Infura*. Accessed: Apr. 2021. [Online]. Available: https://infura.io/

[41] *Ropsten*. Accessed: Apr. 2021. [Online]. Available: https://github.com/ethereum/ropsten

[42] *Salt Contract*. Accessed: Apr. 2021. [Online]. Available: https://etherscan.io/address/0x4156d3342d5c385a87d264f90653733592000581%

[43] *UniswapMakerBroker Contract*. Accessed: May 2020. [Online]. Available: https://etherscan.io/address/0xa35f3acb4d6c43e6f9a1c2d8c136ad4be725152f%#code

[44] *BNB Token Contract*. Accessed: Apr. 2021. [Online]. Available: https://etherscan.io/address/0xB8c77482e45F1F44dE1745F52C74426C631bDD52%

[45] *UniversalDeployer Contract*. Accessed: Apr. 2021. [Online]. Available: https://etherscan.io/address/0x252f1c9aee12a65ac113e4b6c4660a4c2f572b06%#code
[46] *Estimate Gas*. Accessed: Apr. 2021. [Online]. Available: https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_estimategas
[47] J. Rudden. (May 2020). *Ethereum Price Monthly 2016–2020*. [Online]. Available: https://www.statista.com/statistics/806453/price-of-ethereum/
[48] *ETH Gas Station*. Accessed: Apr. 2021. [Online]. Available: https://ethgasstation.info/
[49] W. She *et al.*, "A double steganography model combining blockchain and interplanetary file system," *Peer-to-Peer Netw. Appl.*, vol. 14, pp. 3029–3042, 2021.
[50] S. Liu, Y. Liu, C. Feng, H. Zhao, and Y. Huang, "Blockchain privacy data protection method based on HEVC video steganography," in *Proc. 3rd Int. Conf. Smart BlockChain (SmartBlock)*, Oct. 2020, pp. 1–6.
[51] A. A. Giron, J. E. Martina, and R. Custódio, "Bitcoin blockchain steganographic analysis," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.* Cham, Switzerland: Springer, Oct. 2020, pp. 41–57.
[52] S. K. Okupski, "(Ab) using bitcoin for anti-censorship tool," M.S. thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2014.
[53] M. D. Sleiman, A. P. Lauf, and R. Yampolskiy, "Bitcoin message: Data insertion on a proof-of-work cryptocurrency system," in *Proc. Int. Conf. Cyberworlds (CW)*, Oct. 2015, pp. 332–336.
[54] J. Tian, G. Gou, C. Liu, Y. Chen, G. Xiong, and Z. Li, "Dlchain: A covert channel over blockchain based on dynamic labels," *Information and Communications Security*, in J. Zhou, X. Luo, Q. Shen, and Z. Xu, Eds. Cham, Switzerland: Springer, 2020, pp. 814–830.
[55] A. Fionov, "Exploring covert channels in bitcoin transactions," in *Proc. Int. Multi-Conf. Eng., Comput. Inf. Sci. (SIBIRCON)*, Oct. 2019, pp. 0059–0064.
[56] O. Torki, M. Ashouri-Talouki, and M. Mahdavi, "Blockchain for steganography: Advantages, new algorithms and open challenges," 2021, *arXiv:2101.03103*. [Online]. Available: http://arxiv.org/abs/2101.03103
[57] D. Frkat, R. Annessi, and T. Zseby, "ChainChannels: Private botnet communication over public blockchains," in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber, Phys. Social Comput. (CPSCom) IEEE Smart Data (SmartData)*, Jul. 2018, pp. 1244–1252.
[58] A. I. Basuki and D. Rosiyadi, "Joint transaction-image steganography for high capacity covert communication," in *Proc. Int. Conf. Comput., Control, Informat. Appl. (IC3INA)*, Oct. 2019, pp. 41–46.
[59] F. Gao, L. Zhu, K. Gai, C. Zhang, and S. Liu, "Achieving a covert channel over an open blockchain network," *IEEE Netw.*, vol. 34, no. 2, pp. 6–13, Mar. 2020.
[60] J. Partala, "Provably secure covert communication on blockchain," *Cryptography*, vol. 2, no. 3, p. 18, Aug. 2018.
[61] N. Alsalami and B. Zhang, "Uncontrolled randomness in blockchains: Covert bulletin board for illicit activity," in *Proc. IEEE/ACM 28th Int. Symp. Qual. Service (IWQoS)*, Jun. 2020, pp. 1–10.
[62] F. Ding, "Broadcasting steganography in the blockchain," in *Proc. 18th Int. Workshop, Digit. Forensics Watermarking (IWDW)*, vol. 12022. Chengdu, China: Springer, Mar. 2020, p. 256.

MAR GIMENEZ-AGUILAR received the M.Sc. degree in cybersecurity from the University Carlos III of Madrid, Spain, where she is currently pursuing the Ph.D. degree with the Computer Security Laboratory. Her research interests include cybersecurity, specially steganography and cryptography, and blockchain. At the moment, she is focused studying different aspects of cybersecurity in relation with blockchain technologies.

JOSE M. DE FUENTES received the Ph.D. degree in computer science from the University Carlos III of Madrid, Spain. He is currently an Associate Professor with the Department of Computer Science and Engineering, University Carlos III of Madrid. He is also a Computer Scientist Engineer with the University Carlos III of Madrid. He has published several articles in international conferences and journals. He is participating in several national research and development projects. His main research interests include cybersecurity as well as security and privacy in the Internet of Things and *ad-hoc* networks.

LORENA GONZÁLEZ-MANZANO received the Ph.D. degree in computer science from the University Carlos III of Madrid, Spain, with a focus on security and privacy in social networks. She is currently an Associate Professor with the Computer Security Laboratory, University Carlos III of Madrid. She is also a Computer Scientist Engineer with the University Carlos III of Madrid. She has published several papers in national and international conferences and journals. She is involved in national research and development projects. Her research interests include the Internet of Things, cloud computing security, and cybersecurity.

CARMEN CAMARA received the M.Sc. degree in biomedical engineering from the Technical University of Madrid and the Ph.D. degree in computer science and technology, with specialization in artificial intelligence from the University Carlos III of Madrid, Spain. She is currently an Assistant Professor with the Computer Security Laboratory, University Carlos III of Madrid. She is also focused on designing secure solutions for implantable medical devices. Her research interests include applied cryptography and biometrics.