R3C3: Cryptographically secure Censorship Resistant Rendezvous using Cryptocurrencies

Mohsen Minaei* Pedro Moreno-Sanchez* Aniket Kate

Purdue University
{mohsen, pmorenos, aniket}@purdue.edu
Revision: May 15, 2018

Abstract

Cryptocurrencies and blockchains are set to play a major role in the financial and supply-chain systems. Their presence and acceptance across different geopolitical corridors, including in repressive regimes, have been one of their striking features. In this work, we leverage this popularity for bootstrapping censorship resistant (CR) communication. We formalize the notion of stego-bootstrapping scheme and formally describe the security notions of the scheme in terms of rareness and security against chosen-covertext attacks. We present R3C3[1], a Cryptographically secure Censorship-Resistant Rendezvous using Cryptocurrencies. R3C3 allows a censored user to interact with a decoder entity outside the censored region, through blockchain transactions as rendezvous, to obtain bootstrapping information such as a CR proxy and its public key. Unlike the usual bootstrapping approaches (e.g., emailing) with heuristic security if any, R3C3 employs public-key steganography over blockchain transactions to ensure cryptographic security, while the blockchain transaction costs may deter the entry-point harvesting attacks. We develop bootstrapping rendezvous over Bitcoin, Zcash, Monero and Ethereum as well as the typical mining process, and analyze their effectivity in terms of cryptocurrency network volume and introduced monetary cost. With its highly cryptographic structure, Zcash is an outright winner for normal users with 1168 byte bandwidth per transaction costing only 0.03 USD as the fee, while mining pool managers have a free, extremely high bandwidth rendezvous when they mine a block.

1 Introduction

One of the most ubiquitous and challenging problems faced by the Internet today is the restrictions imposed on its free use. Repressive and totalitarian governments continue professing Internet censorship to their citizens. Censors employ techniques [67] ranging from IP address filtering to deep-packet inspection to block disfavored Internet content [29, 80]. Censored users are thereby prevented from not only accessing information on the Internet in a totally free manner but also from expressing their views freely. Several circumvention systems have been proposed and deployed over the last decade [8, 19, 23, 27, 33, 41, 48, 52, 54, 56, 59, 65, 73, 78, 79]; nevertheless, censorship still remains a challenge to be fully resolved.

Today Bitcoin [58] and other cryptocurrencies are observing a fast growth and worldwide adoption. Importantly, adoption is prevalent even in countries like China with large-scale censorship, and completely censoring Bitcoin may not be in the best interest of most countries [61]. The same also holds true for other cryptocurrencies focussed on different aspects such as complex smart contracts as in Ethereum [34] or privacy-preserving coin transfers as in Zcash [66] and Monero [68].

Therefore, the availability of the cryptocurrencies across different geopolitical corridors makes them a suitable fully distributed rendezvous to post steganographic messages. In fact, censored users can leverage the highly cryptographic transaction structure to encode censored data while maintaining undetectability. In this work, we thoroughly study for the first time the feasibility of using the different available cryptocurrencies as a censorship circumvention rendezvous.

Firstly, we conceptualize the notion of *stego-bootstrapping scheme*, a two-way handshake between a censored user and an uncensored entity (i.e., decoder) that allows the decoder to transmit bootstrapping credentials of

*Both authors contributed equally and are considered co-first authors.
[1]An extra R for additional Resistance. We acknowledge the motivation from the name DP5 in [31].

an entry point for a censorship-circumvention protocol (e.g., Tor Bridge) to the censored user in the presence of the censor (Section 2). We then formally describe the security properties for a stego-bootstrapping scheme in terms of *rareness* and *security against chosen-covertext attacks*. Intuitively, we say that a stego-bootstrapping scheme achieves rareness if it does not decode a valid message from a regular transaction (i.e., not carrying steganographic data). Moreover, we say that a stego-bootstrapping scheme is secure against chosen-covertext attacks if the adversary, given a covertext, cannot tell better than guessing whether it encodes a message of his choice (Section 2).

Secondly, we contribute R3C3, our instantiation of the stego-bootstrapping scheme. R3C3 uses cryptocurrencies as rendezvous and reuses functionality already available in them, making R3C3 seamlessly deployable with the major cryptocurrencies available today. In fact, we describe how R3C3 works using Bitcoin, Zcash, Monero, and Ethereum as rendezvous. Additionally, we describe how to leverage the block creation as the rendezvous, a feature available by definition in every cryptocurrency (Section 3 and 4).

Thirdly, we carry out a comparative study of the different rendezvous by thoroughly evaluating their tradeoffs in terms of available bandwidth, monetary costs and percentage of cover transactions. Zcash is an outright winner for normal users with 1168 byte bandwidth per transaction costing only 0.04 USD as the fee. When using a block as rendezvous, miners have an extremely high bandwidth at no cost. The associated cost to our solutions raises the bar for the censor to perform harvesting attacks [54]. Currently, censors retrieve entry points at no cost (e.g., sending an email), making it easy to enumerate and block them (Section 5 and 6).

Finally, we have implemented R3C3 using Zcash as rendezvous, demonstrating the feasibility of our solution in practice. Moreover, the performance evaluation of our prototype shows the feasibility for an uncensored user to monitor simultaneously all the cryptocurrency blockchains looking for encoded data in real time, even with her commodity equipment. Our sample execution in the Zcash test-net demonstrates that R3C3 is compatible with current cryptocurrencies and ready to be deployed in practice (Section 7).

2 Problem Statement

In this work, we consider the problem of bootstrapping communication into an uncensored area. We refer to this problem as *stego-bootstrapping* and illustrate it in Figure 1. A *censored user* sitting within a *censored area* wants to receive the credentials (IP address, port number, and public key) of a censorship-resistance protocol entry point (e.g., Tor bridge). For that, the censored user first sends a short *forward* bootstrapping message to the uncensored area. A *decoder* is sitting in the uncensored area waiting to decode a message from the censored user and reply it with a *backward* message that contains the information required to connect to an entry point of an anonymous communication protocol (e.g., a Tor bridge) or a proxy. The communication between censored user and decoder is hindered by the *censor*, an entity that decides what messages enter or exit the censored area.

Apart from the natural challenge of having the censor inspecting all messages sent by the censored user, the censor can also run the protocol impersonating a censored user, and this leads to two main problems. First, the censor learns the identity of the decoder and can easily stop the messages addressed directly to it. Second, the censor reproducing the protocol steps honestly appears as an honest censored user to the decoder. The censor thereby receives the identity of a bootstrapping node that can be blacklisted. Repeating these steps several times, the censor can perform a *harvesting attack* [54] and obtain all available entry points.

Therefore, we require a solution that allows the censored user to send bootstrapping messages to the decoder without directly addressing them to it. Additionally, this solution must impose an extra cost for the censor to carry out a harvesting attack.

In the rest of this section, we formalize this problem and describe the considered threat model and the goals.

2.1 Stego-Bootstrapping Scheme

The stego-bootstrapping problem inherently requires a *two-way handshake*, a *challenge* from the censored user to the decoder and the corresponding *response* from the decoder to the censored user. The two-way handshake can be considered as two *independent* "one-way handshakes" and each defined in terms of a public-key stegosystem [30], with a single setup, encoding and decoding algorithms. This approach, however, requires the decoder and censored user knowing each other's public keys in advance. In practice, instead, the censored user knows the public key of the decoder, but the decoder does not know the public key of the censored user.

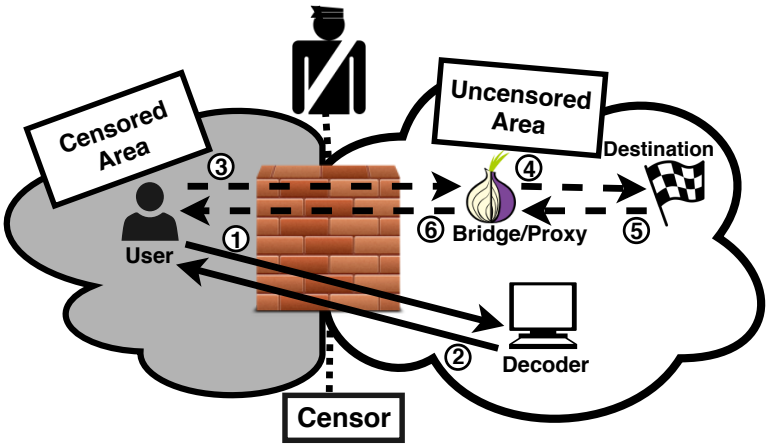


Figure 1: Censorship circumvention bootstrapping problem. Censored user sends a covertext to the decoder, who replies with another covertext including proxy's details. Then, the censored user can access censored information through the proxy. We, focus on the bootstrapping process (solid arrows).

We consider the two-way handshake as a whole and only require that the censored user knows in advance the public key of the decoder. Therefore, our definition of stego-bootstrapping scheme contains two pairs of encoding and decoding algorithms, the first for the challenge operations and the second for the response operations.

In the following, we formally define the notion of *stego-bootstrapping scheme* as well as its properties.

Notation. Let λ represent the security parameter. Let $\epsilon_1(\lambda)$ and $\epsilon_2(\lambda)$ be two negligible functions. Let $\mathcal{M}_c$ and $\mathcal{M}_r$ be a set of challenge and response messages respectively. Let $\mathcal{C}_c$ and $\mathcal{C}_r$ be a set of challenge and response covertexts respectively. Finally, let $\mathcal{T}$ be a set of tags.

Definition 1 (Stego-Bootstrapping Scheme)**.** *The stego-bootstrapping scheme is a tuple of algorithms (*$\mathsf{SBSetup}$*,* $\mathsf{SBEncode}_f$*,* $\mathsf{SBDecode}_f$*,* $\mathsf{SBEncode}_b$*,* $\mathsf{SBDecode}_b$*) defined as described below:*

- $vk_d, sk_d, \tau \leftarrow \mathsf{SBSetup}(\lambda)$. *On input the security parameter* λ*, output a public key* vk_d*, a private key* sk_d *and a tag* $\tau \in \mathcal{T}$*.*
- $\{(cc, k), \perp\} \leftarrow \mathsf{SBEncode}_f(vk_d, cm, \tau)$. *On input a public key* vk_d*, a challenge message* $cm \in \mathcal{M}_c$ *and a tag* $\tau \in \mathcal{T}$*, output either a tuple consisting of a challenge covertext* $cc \in \mathcal{C}_c$ *and a symmetric key* k*; or the special symbol* $\perp$ *indicating an error in the encoding process.*
- $\{(cm, k'), \perp\} \leftarrow \mathsf{SBDecode}_f(sk_d, cc, \tau)$. *On input a a private key* sk_d*, a challenge covertext* $cc \in \mathcal{C}_c$ *and a tag* $\tau \in \mathcal{T}$*, output either a tuple consisting of a challenge message* $cm \in \mathcal{M}_c$ *and a symmetric key* k'*; or the special symbol* $\perp$ *indicating an error in the decoding process.*
- $\{rc, \perp\} \leftarrow \mathsf{SBEncode}_b(sk_d, k', rm)$. *On input the private key* sk_d*, a symmetric key* k' *and a response message* $rm \in \mathcal{M}_r$*, output either a response covertext* $rc \in \mathcal{C}_r$*; or the special symbol* $\perp$ *indicating an error in the encoding process.*
- $\{rm, \perp\} \leftarrow \mathsf{SBDecode}_b(vk_d, k, rc)$. *On input the public key* vk_d*, a symmetric key* k *and a response covertext* $rc \in \mathcal{C}_r$*, output a response message* $rm \in \mathcal{M}_r$*; or the special symbol* $\perp$ *indicating an error in the decoding process.*

Theorem 1 (Stego-Bootstrapping Scheme Correctness)**.** *Let* vk_d, sk_d, τ *be the output of* $\mathsf{SBSetup}(\lambda)$*. Let* $cm \in \mathcal{M}_c$ *and* $rm \in \mathcal{C}_r$ *be a pair of challenge and response messages. A stego-bootstrapping scheme is considered correct if all the following conditions hold:*

1. *Let* (cc, k) *be the output of* $\mathsf{SBEncode}_f(vk_d, cm, \tau)$*. Then, the algorithm* $\mathsf{SBDecode}_f(sk_d, cc, \tau)$ *returns a tuple* (cm', k') *such that* $cm' = cm$ *and* $k' = k$*.*
2. *Let* rc *be the output of* $\mathsf{SBEncode}_b(sk_d, k', rm)$*. Then, the algorithm* $\mathsf{SBDecode}_b(vk_d, k, rc)$ *returns a response message* rm' *such that* $rm' = rm$*.*

2.2 Threat Model

We consider the censor as a malicious adversary with network capabilities within the censored area. It can use a wide range of passive and active censorship techniques proposed in the literature such as eavesdrop, block and inject traffic, statistical traffic analysis or active probing [47, 51, 72, 75, 77]. Moreover, we assume that the censor knows the public key associated to the decoder. The censor has, however, limited capabilities outside the censored area. In particular, the censor has no control over the decoder or her network communications.

We further require that the censor does not block the communications between censored users and the rendezvous system where both censored user and decoder post their messages. We believe that this assumption is realistic in practice, for instance, using a blockchain as an instantiation of the rendezvous. The user base for different cryptocurrencies are continuously growing even in countries with heavy censorship, and banning them all implies the banning of a cryptocurrency, having thereby economic consequences for the censored area [61]. Finally, we require that the censor must not be able to modify or delete any information stored in the rendezvous. Following with our running example, this can be achieved in practice as cryptocurrency transactions are authenticated and replicated among all miners worldwide so that any attempt of modifying such data is detectable by the censored user.

2.3 Security Goals

We characterize two fundamental security properties for a stego-bootstrapping scheme, namely, *rareness* and *security against chosen-covertext attacks*.

Intuitively, we say that a stego-bootstrapping scheme achieves rareness if the probability of decoding a covertext chosen uniformly at random into a valid message is negligible. Moreover, we say that a stego-bootstrapping scheme is secure against chosen-covertext attacks if the adversary, given a pair of challenge and response covertexts, cannot tell better than guessing whether they encode a pair of challenge and response messages of his choice.

In the following, we formalize the notion of rareness in Definition 2. We inspire this definition from that of Ruffing et al. [65]. They, however, propose a definition for an identity-based steganography scheme with a single encoding and decoding algorithms. We adapt it for a stego-bootstrapping scheme by considering the two encoding and decoding algorithms as described in Definition 1.

Definition 2 (Stego-Bootstrapping Rareness)**.** *Let $\epsilon_1(\lambda)$ and $\epsilon_2(\lambda)$ be two negligible functions. A stego-bootstrapping scheme achieves* rareness *if for all tuples (vk_d, sk_d, τ) output by $\mathsf{SBSetup}(\lambda)$, all the following conditions hold:*

1. $\Pr[\mathsf{SBDecode}_f(sk_d, cc, \tau) \neq \perp \mid cc \leftarrow_\$ \mathcal{C}_c] \leq \epsilon_1(\lambda)$;
2. *Let (cc, k) be the output of $\mathsf{SBEncode}_f(vk_d, cm, \tau)$ for an arbitrary $cm \in \mathcal{M}_c$. Then,* $\Pr[\mathsf{SBDecode}_b(vk_d, k, rc) \neq \perp \mid rc \leftarrow_\$ \mathcal{C}_r] \leq \epsilon_2(\lambda)$.

Next, we characterize a fundamental security property for a stego-bootstrapping scheme, namely, security against chosen-covertext attacks (SBS-CCA). We inspire this definition from SS-CCA by Backes et al. [30] and adapt it to consider the two encoding and decoding algorithms as defined in Definition 1.

We define SBS-CCA as a cryptographic game ($\mathsf{Exp}_A^{SBS\text{-}CCA}(\lambda)$) between two players, a challenger and an attacker. This game is played in five rounds: key generation, first decoding stage, challenge, second decoding stage and guessing stage.

In the key generation stage, the challenger runs $vk_d, sk_d, \tau \leftarrow \mathsf{SBSetup}(\lambda)$ and hands in the public key vk_d and the tag τ to the attacker.

In the first decoding stage, the attacker has access to an encoding oracle O^{enc} and a decoding oracle O_1^{dec} defined as follows:

- O^{enc} has access to the public key vk_d, the private key sk_d and the tag τ. On input a tuple $(cm, rm) \in \mathcal{M}_c \times \mathcal{M}_r$, O^{enc} returns the special symbol $\perp$ or a tuple (cc, k, rc) constructed as follows:
  1. Run $\{(cc, k), \perp\} \leftarrow \mathsf{SBEncode}_f(vk_d, cm, \tau)$. If output is $\perp$, return $\perp$.
  2. Run $\{(cm, k'), \perp\} \leftarrow \mathsf{SBDecode}_f(sk_d, cc, \tau)$. If output is $\perp$, return $\perp$.
  3. Run $\{rc, \perp\} \leftarrow \mathsf{SBEncode}_b(sk_d, k', rm)$. If output is $\perp$, return $\perp$.

4. Return (cc, k, rc).

- O_1^{dec} has access to the public key vk_d, the private key sk_d and the tag τ. On input a tuple (cc, k, rc) where $cc \in \mathcal{C}_c$, $rc \in \mathcal{C}_r$ and k is a symmetric key, the decoding oracle O_1^{dec} outputs either the special symbol $\perp$ or a tuple (cm, rm) constructed as follows:

  1. Run $\{(cm, k'), \perp\} \leftarrow \mathsf{SBDecode}_f(sk_d, cc, \tau)$. If output is $\perp$, return $\perp$.
  2. Run $\{rm, \perp\} \leftarrow \mathsf{SBDecode}_b(vk_d, k, rc)$. If output is $\perp$, return $cm, \perp$.
  3. Output (cm, rm).

In the challenge phase, the attacker sends a pair of messages $(cm^*, rm^*) \in \mathcal{M}_c \times \mathcal{M}_r$ to the challenger. Then, challenger chooses a bit b and carries out the following steps depending on it.

- $b = 0$: The challenger sets $(cc^*, rc^*) \leftarrow_{\$} \mathcal{C}_c \times \mathcal{C}_r$ and returns the tuple (cc^*, rc^*).
- $b = 1$: The challenger carries out the following steps:

  1. Run $\{(cc^*, k^*), \perp\} \leftarrow \mathsf{SBEncode}_f(vk_d, cm^*, \tau)$. If output is $\perp$, return $\perp$.
  2. Run $\{(cm^*, k'^*), \perp\} \leftarrow \mathsf{SBDecode}_f(sk_d, cc^*, \tau)$. If output is $\perp$, return $\perp$.
  3. Run $\{rc^*, \perp\} \leftarrow \mathsf{SBEncode}_b(sk_d, k'^*, rm^*)$. If output is $\perp$, return $\perp$.
  4. Return (cc^*, rc^*).

The challenge phase finishes by sending the tuple cc^*, rc^* to the attacker. The idea is that the attacker should guess the value of b, i.e., the attacker should determine whether the messages cm^*, rm^* have been encoded in the covertexts cc^*, rc^* or cc^*, rc^* have been chosen at random from $\mathcal{C}_c \times \mathcal{C}_r$ instead. Note that, here the challenger does not reveal the symmetric key k^* to the adversary when $b = 1$, as otherwise, the adversary can trivially guess the bit b by locally running the algorithm $\{rm, \perp\} \leftarrow \mathsf{SBDecode}_b(vk_d, k^*, rc^*)$ and checking whether the output is a message $rm = rm^*$. In practice, this models the fact that the intermediate key material (i.e., symmetric keys k, k') are known only to the honest participants and not the the adversary (e.g., the censor). In the second decoding stage, the attacker has access to the encoding oracle O^{enc} described above. Moreover, the attacker has access to a decoding oracle O_2^{dec}, which is analogous to O_1^{dec} except that upon receiving the pair (cc^*, k'', rc^*), where k'' denotes any symmetric key, returns $\perp$. Finally, the game enters the guessing stage where the attacker simply outputs a bit b^*.

Definition 3 (Security against covertext-chosen attacks)**.** *A stego-bootstrapping scheme is secure against chosen-covertext attacks if every probabilistic polynomial-time adversary A has negligible advantage in the game* $\mathsf{Exp}_A^{SBS\text{-}CCA}(\lambda)$*. We define the adversary's advantage as* $|\Pr[\mathsf{Exp}_A^{SBS\text{-}CCA}(\lambda) = b] - 1/2|$

2.4 System Goals

A stego-bootstrapping system should further preserve the following system properties.

- **Harvesting-Resistance** The stego-bootstrapping system must ensure that an attacker cannot carry out unlimited protocol executions. This goal aims thereby at preventing the censor from obtaining all the entry points.
- **Cost-Efficiency** The stego-bootstrapping system must provide a bootstrapping solution at a reduced cost for the honest censored users and the decoder. We measure the cost in the USD currency.
- **Compatibility** The stego-bootstrapping system must reuse the functionality already provided by the chosen rendezvous.

3 Key Ideas and Solution Overview

3.1 Key Ideas

Blockchain as rendezvous. Our solution, *R3C3*, leverages a blockchain as rendezvous for censored messages encoded as blockchain transactions. The many blockchain-based systems existing today such as cryptocurrencies (e.g., Bitcoin), privacy-preserving cryptocurrencies (e.g., Zcash or Monero), or smart contracts (e.g., Ethereum) are managed in a distributed fashion by users located worldwide. This hinders the censor's task of stopping these systems, as demonstrated by other distributed systems like Tor. Moreover, the fact that the use of blockchain-based systems is spread worldwide (even in regions with heavily active censorship [62]) adds an economic penalty for the censor to prevent the censored users from using it.

A difference between R3C3 and other censorship circumvention protocols is that covertexts remain visible in the corresponding blockchain even after the bootstrapping process has finished. However, this cannot be leveraged by the censor because R3C3 is secure against chosen-covertext attacks. Therefore, the adversary cannot tell better than guessing whether a covertext (i.e., a cryptocurrency transaction as implemented in R3C3) contains bootstrapping data. Therefore, the censor is left with the choice of banning the complete blockchain system or allow it completely.

Steganographic tagging scheme. Another building block required in our system consists of a cryptographic construction that converts censored messages into ciphertexts that can be then encoded into a covertext transaction. For this purpose, there exist several public-key steganographic tagging schemes in the literature [28, 30, 43, 49, 71]. However, they assume in general high bandwidth not available in many different blockchain transactions. Therefore, in this work, we adapt the construction used by Wustrow et al. [78]. The main advantage of this approach is its succinctness. A ciphertext is composed of a group element (e.g., an elliptic curve point) representing a public key and a random-looking bitstring of the size similar to the plaintext message. Another advantage is that the group element can be easily included in a blockchain transaction as it already uses elliptic curve points to represent public keys.

Fees. R3C3 introduces a fee overhead, which is inevitable due to the use of cryptocurrencies. To process and confirm a transaction in each of the widely used cryptocurrencies, a transaction fee is paid to the miners. We leverage this fee to increase the cost of a harvesting attack from the adversary, aiming thereby at a harvesting-resistance system.

Paid Services. Currently, many censorship circumvention systems are paid services including even a premium account to provide better performance. VPNs and domain fronting techniques [44] such as Meek [13, 17], Lantern [12], Psiphon [15], are examples of systems that require payments for their high-performance services. R3C3 can be used to not only bootstrap free of charge services such as Tor, but also bootstrap the mentioned paid services. The fee of R3C3 compared to the actual cost of mentioned services is negligible. Moreover, the bootstrapping process takes place infrequently.

3.2 Solution Overview

In a nutshell, the R3C3 protocol works as depicted in Figure 2. The censored user, with access to the decoder's public key vk_d, a challenge message cm and a tag τ, encodes cm and τ into a challenge covertext cc (i.e., a blockchain transaction) and transfers cc along the chosen communication medium. We consider a 64 bit tag τ as described in [78] to flag the communication as part of a R3C3 execution. Additionally, cm contains the

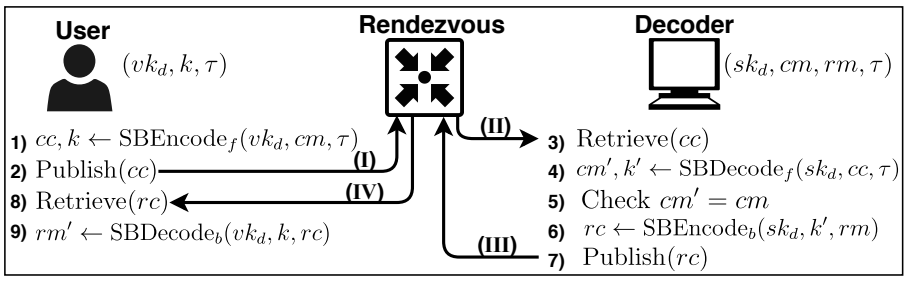


Figure 2: Overview of the R3C3 protocol.

information request about the anonymous communication protocol to be used after the bootstrapping (e.g., Tor in obfs3 mode [20]). As *cm* carries little information, it requires small bandwidth.

After that, the decoder, which is continuously getting covertexts from the chosen rendezvous, eventually gets the covertext *cc*, decodes it and gets the challenge message *cm*. This notifies the decoder that some censored user is trying to get the bootstrapping information and that such information must be included in a response message *rm*.

For that, the decoder encodes *rm* into a new response covertext (*rc*) and adds it to the rendezvous. The response message *rm* conveys information such as the IP address, connection port, and other optional configuration information for the entry point requested by the censored user. Therefore, we envision that *rm* is longer than *cm* and therefore it requires larger bandwidth. Finally, the censored user can obtain *rc* and decode it so that the censored user gets the bootstrapping information.

What happens after this point is out of the scope of this work. For instance, the bootstrapping information could contain the IP address, port and public key of a Tor bridge that allows the censored user to connect to the Tor network. This is, however, an orthogonal problem discussed in depth in the literature so far [8, 12, 13, 20, 23, 33, 36, 40, 57, 73, 76]. We rely on these complementary solutions for the censorship-resistant communication after bootstrapping.

3.3 Summary of Our Findings

We summarize our findings associated with the feasibility of different cryptocurrencies as rendezvous in Table 1.

We observe that shielded Zcash transactions provide the most bandwidth with 1168 bytes with the lowest cost of 0.03 USD among all other cryptocurrencies. The downside is that only 8.5% of the transactions within Zcash are shielded and is not the most prominent type of transaction. Moreover, Zcash currently has the lowest market capitalization and therefore exerts the lowest economic impact for a censor if it decides to ban it.

Bitcoin, the most used cryptocurrency, provides 20 and 40 bytes for the challenge and response messages correspondingly. Our Bitcoin-based solution relies on a transaction type used by more than 89% of the Bitcoin transactions, therefore hindering the censor's task. However, the fees in Bitcoin are the largest of all cryptocurrencies and our encoding method entails the loss of coins as they are sent to unrecoverable addresses. Fortunately, it is possible to lower the cost by deploying the same encoding techniques over Zcash transparent transactions as they are conceptually identical to Bitcoin transactions.

After shielded Zcash, Monero provides the most bandwidth with 256 bytes. However, the fee associated with a Monero transaction is similar to the one of Bitcoin. The type of transactions we consider in Monero blends in with 33% of all Monero transactions, making it difficult for the censor to block all such transactions. Ethereum, a blockchain-based system for smart contracts, provides the least amount of bandwidth with 16 bytes and a moderate cost of 20-40 cents when compared to other approaches.

Lastly, we explored the possibility of encoding data at the formation of blocks for the blockchain. This possibility arises when both the censored users and decoder carry out the miner functionality within the blockchain system. Our evaluation results show that, when the censored user and the decoder play the miner role in Bitcoin, they can gain a bandwidth of up to 575 bytes without paying for a transaction fee and even receiving profit as adding a new block to the blockchain results the corresponding block mining reward.

		Bitcoin	Zcash (Tr)	Zcash (Sh)	Monero	Ethereum	Miner
Challenge	Bandwidth (bytes)	20	20	1168	256	16	575
	Tx fee	0.88USD	0.03 USD	0.03 USD	0.77 USD	0.4 USD	—
	Burnt amount	0.22 USD	0.01 USD	—	—	—	—
Response	Bandwidth (bytes)	40	40	1168	256	16	575
	Tx fee	0.88 USD	0.03 USD	0.03 USD	0.77 USD	0.2 USD	—
	Burnt amount	0.44 USD	0.02 USD	—	—	—	—
	% Sibling transactions	89	91	8.5	33	> 8	100
	Market cap USD (rank)	146B (1)	1B (25)	1B (25)	36B (12)	83B (2)	—

Table 1: Comparison of the different rendezvous. Here, we consider the coins market value [6] at the time of writing. We denote *transparent* by (Tr) and *shielded* by (Sh). Similar to Zcash(Tr), results for Bitcoin can be applied to Altcoins following the Bitcoin transaction patterns.

4 Our Protocol

4.1 Building Blocks

Encoding Scheme. For ease of exposition of R3C3, we describe here the notion of *encoding scheme*. It allows to encode challenge and response data as a transaction compatible with the rendezvous available between censored user and decoder. In this manner, we abstract away the details dependent on the different rendezvous.

Let $\mathcal{D}_c$ and $\mathcal{D}_r$ be a set of challenge and response data respectively. Let $\mathcal{A}_c$ and $\mathcal{A}_r$ be a set of challenge and response auxiliary information. Let $\mathcal{T}_c$ and $\mathcal{T}_r$ be a set of challenge and response transactions respectively.

Definition 4 (Encoding Scheme)**.** *An* encoding scheme *is a tuple of algorithms* ($\mathsf{TxEncode}_f$, $\mathsf{TxDecode}_f$, $\mathsf{TxEncode}_b$, $\mathsf{TxDecode}_b$) *defined as below:*

- $\{ctx, \perp\} \leftarrow \mathsf{TxEncode}_f(cd, ca)$. *On input a piece of challenge data* $cd \in \mathcal{D}_c$ *and the challenge auxiliary information* $ca \in \mathcal{A}_c$*, output a challenge transaction* $ctx \in \mathcal{T}_c$ *or the special symbol* $\perp$ *indicating an error in the encoding process.*
- $\{cd, \perp\} \leftarrow \mathsf{TxDecode}_f(ctx)$. *On input a challenge transaction* $ctx \in \mathcal{T}_c$*, output a piece of challenge data* $cd \in \mathcal{D}_c$ *or the special symbol* $\perp$ *indicating an error in the decoding process.*
- $\{rtx, \perp\} \leftarrow \mathsf{TxEncode}_b(rd, ra)$. *On input a piece of response data* $rd \in \mathcal{D}_r$ *and the response auxiliary information* $ra \in \mathcal{A}_r$*, output a response transaction* $rtx \in \mathcal{T}_r$ *or the special symbol* $\perp$ *indicating an error in the encoding process.*
- $\{rd, \perp\} \leftarrow \mathsf{TxDecode}_b(rtx)$. *On input a response transaction* $rtx \in \mathcal{T}_r$*, output a piece of response data* $rd \in \mathcal{D}_r$ *or the special symbol* $\perp$ *indicating an error in the decoding process.*

Definition 5 (Encoding Scheme Correctness)**.** *We say that a encoding scheme is* correct *if for all pieces of challenge data* $cd \in \mathcal{D}_c$*, challenge auxiliary information* $ca \in \mathcal{A}_c$*, response data* $rd \in \mathcal{D}_r$ *and response auxiliary information* $ra \in \mathcal{A}_r$*, all the following conditions hold:*

- *Let* $ctx \leftarrow \mathsf{TxEncode}_f(cd, ca)$*. Then,* $cd^* \leftarrow \mathsf{TxDecode}_f(ctx)$ *and* $cd^* = cd$*.*
- *Let* $rtx \leftarrow \mathsf{TxEncode}_b(rd, ra)$*. Then,* $rd^* \leftarrow \mathsf{TxDecode}_b(rtx)$ *and* $rd^* = rd$*.*

We instantiate the encoding scheme using the different cryptocurrencies as described in Section 5. Additionally, we instantiate the encoding scheme leveraging the mining process in Section 6.

Non-interactive Key Exchange. A non-interactive key exchange (NIKE) mechanism is a tuple of algorithms ($\mathsf{NIKE.KeyGen}, \mathsf{NIKE.SharedKey}$). The algorithm $(vk, sk) \leftarrow \mathsf{NIKE.KeyGen}(id)$ outputs a public key vk and a secret key sk for a given party identifier id. The algorithm $k \leftarrow \mathsf{NIKE.SharedKey}(id_1, id_2, sk_1, vk_2)$ outputs a shared key k for the two parties id_1 and id_2. We require a non-interactive key exchange mechanism secure in the CKS model. Static Diffie-Hellman key exchange satisfies these requirements [35, 45]. Additionally, we require a function $\mathsf{ID}(vk_u)$ that on input a public key vk_u returns the corresponding identifier id_u. We implement this function as the identity function.

Key Derivation Function. A key derivation function $\mathsf{KDF}(k, l)$ takes as input a key k and a length value l and outputs a string of l bits. We require a secure key derivation function [53]. We use the hash-based key derivation function (HKDF) defined in [53].

4.2 Our Construction

We inspire from TapDance [78] to build the cryptographic construction at the core of R3C3. In TapDance, a Diffie-Hellman key exchange is leveraged to create a symmetric key and an initialization vector (IV) between the censored user and the decoder. This symmetric key and IV are then used to encrypt the message using AES. This approach has the drawback that ciphertexts always have a length multiple of the AES block size, what supposes an overhead that we want to avoid given the somewhat restricted bandwidth in our choice of rendezvous.

In our construction for R3C3, we aim at optimizing the succinctness of the ciphertext. Although we also use the Diffie-Hellman key exchange to generate a symmetric key between the censored user and the decoder,

R3C3

- $vk_d, sk_d, \tau \leftarrow \mathsf{SBSetup}(\lambda)$. Generate $sk_d \leftarrow_\$ \{0,1\}^\lambda$. Compute $vk_d \leftarrow \mathsf{BBK}(sk_d)$. Set $\tau \leftarrow \{0,1\}^{64}$. Return vk_d, sk_d, τ.
- $\{(cc, k), \perp\} \leftarrow \mathsf{SBEncode}_f(vk_d, cm, \tau)$:
  - Compute $vk_u, sk_u \leftarrow \mathsf{NIKE.KeyGen}(id_u)$
  - Compute $k_d \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{ID}(vk_u), \mathsf{ID}(vk_d), sk_u, vk_d)$
  - Compute $sk_s||k_c||k_r \leftarrow \mathsf{HKDF}(k_d, \lambda + l_c + l_r)$
  - Compute $vk_p \leftarrow vk_d^{sk_s}$
  - Set $ct_c := \tau||cm \oplus k_c$
  - Compute $cc \leftarrow \mathsf{TxEncode}_f((vk_u, \mathsf{H}(vk_p), ct_c), sk_u)$
  - If $cc = \perp$, return $\perp$. Else, return the tuple cc, k_d
- $\{(cm, k'), \perp\} \leftarrow \mathsf{SBDecode}_f(sk_d, cc, \tau)$.
  - Compute $cd \leftarrow \mathsf{TxDecode}_f(cc)$
  - If $cd = \perp$, return $\perp$. Otherwise:
    - Parse $vk'_u, \mathsf{H}(vk'_p), ct'_c \leftarrow cd$
    - Compute $k'_d \leftarrow \mathsf{NIKE.SharedKey}(\mathsf{ID}(vk_d), \mathsf{ID}(vk'_u), sk_d, vk'_u)$
    - Compute $sk'_s||k'_c||k'_r \leftarrow \mathsf{HKDF}(k'_d, \lambda + l_c + l_r)$
    - Compute $vk_d \leftarrow g^{sk_d}$
    - Compute $vk''_p \leftarrow vk_d^{sk'_s}$
    - Set $m' := ct'_c \oplus k'_c$
    - Parse $\tau'||cm' \leftarrow m'$
    - Set $b := (\tau' = \tau) \wedge (\mathsf{H}(vk'_p) = \mathsf{H}(vk''_p))$
    - If $b = 0$, return $\perp$. Else, return the tuple cm', k'_d
- $\{rc, \perp\} \leftarrow \mathsf{SBEncode}_b(sk_d, k', rm)$.
  - Compute $sk'_s||k'_c||k'_r \leftarrow \mathsf{HKDF}(k', \lambda + l_c + l_r)$
  - Compute $sk''_p \leftarrow sk_d \cdot sk'_s$
  - Compute $vk''_p := g^{sk''_p}$
  - Set $ct_r := rm \oplus k'_r$
  - Compute $rc \leftarrow \mathsf{TxEncode}_b((ct_r, vk''_p), sk''_p)$
  - Return rc
- $\{rm, \perp\} \leftarrow \mathsf{SBDecode}_b(vk_d, k, rc)$
  - Compute $rd \leftarrow \mathsf{TxDecode}_b(rc)$
  - If $rd = \perp$, return $\perp$. Otherwise:
    - parse $ct'_r, vk_p \leftarrow rd$
    - Compute $sk_s||k_c||k_r \leftarrow \mathsf{HKDF}(k, \lambda + l_c + l_r)$
    - Set $rm := ct'_r \oplus k_r$
    - If $vk_p \neq vk_d^{sk_s}$, return $\perp$. Otherwise, return rm

Figure 3: The R3C3 construction. Here, we denote by l_c the bandwidth available for the challenge message and by l_r, the bandwidth available for the response message. We denote string concatenation by $||$. Here, H is a cryptographic hash as implemented in the corresponding encoding scheme.

we use the symmetric key differently. The symmetric key becomes a master key for a key derivation function to derive three other keys. Two of the three derived keys become fresh symmetric encryption keys to encrypt (and decrypt) the challenge and response messages.

The last derived key becomes a fresh private key sk_s shared between the censored user and the decoder. From sk_s, the censored user can create a public key vk_p such that only the intended decoder knows the corresponding private key sk_p. We call the key pair vk_p, sk_p as the *paying key pair*. The paying key pair is used by honest users to pay for the service provided by the decoder. The censored user can associate coins to vk_p before including it in the covertext sent to the decoder so that vk_p becomes a funded address in the blockchain used as rendezvous. When the covertext arrives to the decoder, the decoder can use the coins associated to vk_p and the corresponding sk_p to pay for the cost of sending the response covertext to the censored user. Remember that the decoder can use the coins at vk_p because our construction reconstructs the corresponding sk_p only at the decoder side.

We formalize our construction for R3C3 in Figure 3.

4.3 Security Analysis

In this section, we note that R3C3 is correct, it achieves rareness and it is secure against covertext-chosen attacks. We then discuss further properties that can be achieved by R3C3 with trivial modifications.

Theorem 2 (R3C3 is correct)**.** *Let NIKE be a correct non-interactive key exchange protocol. Let HKDF be a correct key derivation function. Let H be a collision-resistance hash function. Let* Π *be a correct encoding scheme. Then, R3C3 is a correct stego-bootstrapping scheme as defined in Theorem 1.*

Proof. We start by showing that condition 1 holds. In particular, given a pair $(cc, k) \leftarrow \mathsf{SBEncode}_f(vk_d, cm, \tau)$, we need to show that $\mathsf{SBDecode}_f(sk_d, cc, \tau)$ returns a pair (cm', k') such that $cm' = cm$ and $k' = k$.

By correctness of Π, $vk'_u = vk_u$, $\mathsf{H}(vk'_p) = \mathsf{H}(vk_p)$ and $ct'_c = ct_c$. Moreover, as H is collision-resistant, $vk'_p = vk_p$. As NIKE is a correct non-interactive key exchange protocol, $k'_d = k_d$. If $\mathsf{SBDecode}_f$ does not return $\perp$, this proves that both functionalities output the same symmetric key. Now, we show that $\mathsf{SBDecode}_f$ does not return $\perp$.

Given the correctness of HKDF, the symmetric key $k'_c = k_c$. Then, it is easy to see that $ct'_c \oplus k'_c = \tau'||cm' \oplus k'_c \oplus k'_c = \tau'||cm'$. The fact that $ct'_c = ct_c$ implies that $\tau' = \tau$ and $cm' = cm$. Finally, it is easy to see that vk''_p and vk'_p are constructed equally, and therefore $\mathsf{SBDecode}_f$ returns a tuple (cm', k'_d).

The condition 2 holds following similar arguments. This concludes the proof. □

Theorem 3 (R3C3 achieves rareness)**.** *Let NIKE be a secure non-interactive key exchange protocol in the CKS model. Let HKDF be a secure key derivation function. Let* Π *be a correct encoding scheme. Then R3C3 achieves rareness as defined in Definition 2.*

Proof. We start by showing that condition 1 holds. For that, we need to show that the probability $\Pr[\mathsf{SBDecode}_f(sk_d, cc, \tau) \neq \perp \mid cc \leftarrow_\$ \mathcal{C}_c] < \epsilon_1(\lambda)$.

Let $vk'_u, \mathsf{H}(vk'_p), ct'_c$ be the tuple extracted by $\mathsf{TxDecode}_f(cc)$. W.l.o.g., let $ct'_c := \tau'||cm' \oplus k'_c$. Now, let k''_c be the symmetric key generated after running NIKE and HKDF functions as defined in $\mathsf{SBDecode}_f$. It is easy to see that the probability that $k'_c = k''_c$ is negligible. Therefore, $ct'_c \oplus k''_c = cm'||\tau' \oplus k' \oplus k''_c = \tau^*||cm^*$. Given that, τ^* is pseudorandom string, the probability that $\tau^* = \tau$ is $\frac{1}{2^{|\tau|}}$, and therefore negligible.

Now, we show that the condition 2 holds. For that, we need to show that the probability $\Pr[\mathsf{SBDecode}_b(vk_d, k, rc) \neq \perp \mid rc \leftarrow_\$ \mathcal{C}_r] \leq \epsilon_2(\lambda)$ where k is part of the pair $(cc, k) \leftarrow \mathsf{SBEncode}_f(vk_d, cm, \tau)$.

Let $vk_p := g^{sk_s}$ the public key encoded in cc after executing $\mathsf{SBEncode}_f$. Note that, the same vk_p is generated in $\mathsf{SBDecode}_b$ given that HKDF is a correct key derivation function invoked on the same input. Let $vk'_p := g^{sk'_s}$ be the public key encoded in rc.

Looking at the code, it is clear that each covertext encodes fresh (and therefore different) keys. Therefore, as $cc \neq rc$, it implies that $vk_p \neq vk'_p$. This concludes the proof.

□

Theorem 4 (R3C3 is secure against covertext-chosen attacks)**.** *Let NIKE be a correct and secure non-interactive key exchange protocol in the CKS model. Let HKDF be a correct and secure key derivation function. Let* Π *be a correct encoding scheme. Then, R3C3 is secure against covertext-chosen attacks as defined in Definition 3.*

Proof. Assume by contradiction that R3C3 is not secure against covertext-chosen attacks. Therefore, there must exist an adversary A such that $|\Pr[\mathsf{Exp}_A^{SBS\text{-}CCA}(\lambda) = b] - 1/2| > \epsilon_1(\lambda)$. Then, we construct an adversary B such that $|\Pr[\mathsf{Exp}_B^{NIKE}(\lambda) = b] - 1/2| > \epsilon_2(\lambda)$. We refer the reader to [45] for a formal description of $\mathsf{Exp}_B^{NIKE}(\lambda)$. We define B as follow:

- On input $(\lambda, params)$:
  - Query the challenger with input *register*($\mathsf{ID}(vk_d)$) and retrieve vk_d.
  - Query the challenger with input *extract*($\mathsf{ID}(vk_d)$) and retrieve sk_d.
  - Compute $vk'_d, sk'_d, \tau \leftarrow \mathsf{SBSetup}(\lambda)$. It is important to note that τ is independent of vk'_d and sk'_d. Therefore, we can discard vk'_d and sk'_d and use the pair vk_d, sk_d provided by the challenger.
  - Input (vk_d, τ, λ) to A.
- B simulates the oracle O^{enc} as follows. On input (cm, rm):
  - Query the challenger with *register*($\mathsf{ID}(vk_u)$) and retrieve vk_u.
  - Query the challenger with *extract*($\mathsf{ID}(vk_u)$) and retrieve sk_u.
  - Query the challenger with *reveal*($\mathsf{ID}(vk_u), \mathsf{ID}(vk_d)$) and retrieve k_d.
  - Compute $sk_s||k_c||k_r \leftarrow \mathsf{HKDF}(k_d, \lambda + l_c + l_r)$
  - Compute $vk_p \leftarrow vk_d^{sk_s}$
  - Set $ct_c := (\tau||cm) \oplus k_c$
  - Compute $cc \leftarrow \mathsf{TxEncode}_f((vk_u, \mathsf{H}(vk_p), ct_c), sk_u)$
  - Compute $sk_p \leftarrow sk_d \cdot sk_s$
  - Set $ct_r := rm \oplus k_r$
  - Compute $rc \leftarrow \mathsf{TxEncode}_b((ct_r, vk_p), sk_p)$
  - If $cc = \perp$ or $rc = \perp$, return $\perp$. Else, return the tuple (cc, k_d, rc)

  Due to the correctness of NIKE and HKDF there is no need to run $\mathsf{SBDecode}_f$ as the symmetric key generated in this function is equal to the one in $\mathsf{SBEncode}_f$. Similarly, the HKDF function in $\mathsf{SBEncode}_b$ is not necessary to be computed.
- B simulates the oracle O_1^{dec} as follows. On input (cc, k_d, rc):
  - Compute $cd \leftarrow \mathsf{TxDecode}_f(cc)$
  - If $cd = \perp$, return $\perp$. Otherwise:
    - Parse $vk_u, \mathsf{H}(vk_p), ct_c \leftarrow cd$
    - Compute $sk_s||k_c||k_r \leftarrow \mathsf{HKDF}(k_d, \lambda + l_c + l_r)$
    - Compute $vk_d \leftarrow g^{sk_d}$
    - Compute $vk'_p \leftarrow vk_d^{sk_s}$
    - Set $m := ct_c \oplus k_c$
    - Parse $(\tau'||cm) \leftarrow m$
    - Set $b := (\tau' = \tau) \wedge (\mathsf{H}(vk'_p) = \mathsf{H}(vk_p))$
    - If $b = 0$, return $\perp$.
  - Compute $rd \leftarrow \mathsf{TxDecode}_b(rc)$
  - If $rd = \perp$, return $\perp$. Otherwise:
    - parse $ct_r, vk_p \leftarrow rd$
    - Set $rm := ct_r \oplus k_r$
  - If $b = 0$ or $vk_p \neq vk_d^{sk_s}$, return $\perp$. Otherwise, return (cm, rm)

  Due to the correctness of HKDF there is no need to run $\mathsf{SBEncode}_b$ as the key generated in this function is equal to the one in $\mathsf{SBEncode}_f$.

- At some point A outputs the challenge messages (cm^*, rm^*). Then B proceeds as follows and passes the returned message to A:
  - Query the challenger with $\mathit{register}(\mathsf{ID}(vk_u^*))$ and retrieve vk_u^*.
  - Query the challenger with $\mathit{extract}(\mathsf{ID}(sk_u^*))$ and retrieve sk_u^*.
  - Query the challenger with $\mathit{test}(\mathsf{ID}(sk_u^*), \mathsf{ID}(vk_d))$ and retrieve k_d^*.
  - Compute $sk_s^*||k_c^*||k_r^* \leftarrow \mathsf{HKDF}(k_d^*, \lambda + l_c + l_r)$
  - Compute $vk_p^* \leftarrow vk_d^{sk_s^*}$
  - Set $ct_c := (\tau||cm^*) \oplus k_c^*$
  - Compute $cc^* \leftarrow \mathsf{TxEncode}_f((vk_u^*, \mathsf{H}(vk_p^*), ct_c^*), sk_u^*)$
  - Compute $sk_p^* \leftarrow sk_d \cdot sk_s^*$
  - Set $ct_r^* := rm^* \oplus k_r^*$
  - Compute $rc^* \leftarrow \mathsf{TxEncode}_b((ct_r^*, vk_p^*), sk_p^*)$
  - If $cc^* = \perp$ or $rc^* = \perp$, return $\perp$. Else, return the tuple (cc^*, k_d^*, rc^*)
- A outputs as b as its response to the challenge. Then, B sends response $1 - b$ to the challenger. The b value indicates if A has discovered a random tuple ($b = 0$) or a valid one executed by the protocol ($b = 1$). In the case of challenger the value of b indicates the opposite. If $b = 0$ then a key generated by the protocol is returned, otherwise ($b = 1$) a randomly generated key is returned. Therefore, the value $1 - b$ is passed to the challenger.
- B simulates the decoding oracle O_2^{dec} as defined for O_1^{dec} with the exception of the input (cc^*, k'', rc^*), for any symmetric key k''. In this case B forwards $\perp$ to A.

Analysis B is efficient, i.e., number of queries made to O^{enc}, O_1^{dec}, O_2^{dec}, by A is polynomial and the overall protocol is completed in polynomial time. B faithfully simulates A, i.e., for each of the queries made to the oracles, B executes the steps of the protocol as it is expected by A.

Now, every time A wins the $\mathsf{Exp}_A^{SBS\text{-}CCA}(\lambda)$, B wins the $\mathsf{Exp}_B^{NIKE}(\lambda)$ except for negligible probability. A can distinguish between well formed challenges and random challenges. In other words, she differentiates if B was using the proper NIKE key or a random key.

Therefore, we have that $|\Pr[\mathsf{Exp}_B^{NIKE}(\lambda) = b] - 1/2| = \Pr[\mathsf{Exp}_A^{SBS\text{-}CCA}(\lambda) = b] - 1/2| - \epsilon_3(\lambda)$. By assumption, $\Pr[\mathsf{Exp}_A^{SBS\text{-}CCA}(\lambda) = b] - 1/2| > \epsilon_2(\lambda)$. Then, it holds that $|\Pr[\mathsf{Exp}_B^{NIKE}(\lambda) = b] - 1/2| \geq \epsilon_2(\lambda) - \epsilon_3(\lambda)$. This, however contradicts the fact that NIKE is secure. Therefore, such B must not exist and R3C3 must be secure against chosen covertext attacks.

□

In the following, we argue that R3C3 can be easily modified to achieve *eventual forward secrecy* for the challenge covertext and *immediate forward secrecy* for the response covertext. We also argue that our approach raises the bar for censor detection.

Eventual forward secrecy. The challenge covertext encodes information encrypted using a key derived from vk_d and vk_u. While the corresponding sk_u can be destroyed right after the creation of the corresponding challenge covertext (i.e., the associated coins have been already spent), the decoder might reuse vk_d for several users and therefore should keep the corresponding sk_d in this case. However, the decoder is not forced to keep the same vk_d during the whole R3C3 lifetime. Instead, the decoder can spend the coins associated to vk_d in a fresh public key vk_d' and destroy sk_d afterwards. Such transaction notifies to every user the change of the decoder's public key. Such transaction can also be seen by the censor. However, as sk_d has been destroyed, covertext challenges created with such key are secure. Therefore, R3C3 can achieve eventual forward secrecy for challenge covertexts.

Immediate forward secrecy. In R3C3, the decoder uses its pair of keys vk_d, sk_d also for the response covertext. However, it is possible to modify R3C3 so that the decoder generates instead a fresh pair of keys vk_r, sk_r and uses the newly generated keys to create the response covertext. The decoder can immediately delete sk_r so that forward secrecy is preserved. Additionally, the decoder can include vk_r in the response covertext so that the censored user has enough cryptographic information to decode it.

Hindering censor detection. In R3C3 the public key of decoder vk_d, is used for deriving a shared key for encryption. We, however, note that users do not use vk_d explicitly as the receiver of coins in their transactions. Instead, they encrypt their messages with the shared key and use a fresh non-detectable address as the recipient in the transactions, as explained in Section 4.2. Given that, censor cannot distinguish between a transactions encoding a challenge message and any other transaction. Only the decoder, with access to sk_d can do it. Hence, R3C3 raises the bar to limit the detection capabilities of the censor.

System Fee. R3C3 introduces a fee leveraging a cryptocurrency system. This fee is, however, negligible compared to the cost of the current pay-per-service censor circumvention systems. R3C3 is a bootstrapping mechanism that is employed infrequently, resulting in a low cost amortized over a long period of time. Moreover, this can also improve the performance of non-paid censor circumvention systems. The associated cost raises the bar for the censor to obtain the entry points of a circumvention system. Currently, censors are retrieving these information at no cost (e.g., sending an email), making it easy to enumerate and block the entry points.

5 Cryptocurrency-Based Encodings

5.1 Encoding Scheme in Bitcoin

In this section, we describe the instantiation of the encoding scheme using the Bitcoin transactions. In particular, we first describe the format of Bitcoin addresses and transactions, we then delve on the different possibilities to encode data within them. By studying the a snapshot of the currently deployed Bitcoin blockchain, we then set up the transaction parameters to be used in R3C3 and finally formally describe our instantiation of encoding scheme with Bitcoin transactions. We emphasize that, although we focus on Bitcoin in this section, similar design patterns and decisions apply to several other cryptocurrencies such as Litecoin that follow Bitcoin core functionality. The use of these alternative cryptocurrencies might bring cheaper and yet fully functional instantiations for the encoding scheme required in R3C3.

5.1.1 Address and Transaction Format

A Bitcoin address is composed of a pair of verification and signing ECDSA keys. A Bitcoin address is then represented by the Base58 encoding for the hash of the verification key [2]. Bitcoins are exchanged between addresses by means of a transaction. In its simplest form, a transaction transfers a certain amount of coins from one *input* address to an *output* address. A transaction can contain several input and output addresses.

The Bitcoin protocol uses a scripting system called *Script* [26] that governs how bitcoins can be transferred between addresses within a transaction. In particular, coins are locked in an address according to $\textsc{scriptPubKey}$, a Script excerpt that encodes the conditions to unlock the coins. The fulfillment of such conditions are encoded in another Script excerpt called $\textsc{scriptSig}$. In the illustrative example of Bitcoin transaction in Figure 4, 5BTC are transferred from the input addresses included in $\textsc{scriptPubKey}_{\{0,1\}}$ to the output addresses included in $\textsc{scriptPubKey}_{\{2,3,4\}}$. A transaction is *valid* if coins unlocked (or spent) in the transaction has not been spent previously; the sum of input coins is greater or equal than the sum of output coins; and for each input $\textsc{scriptPubKey}_i$ there exists the corresponding $\textsc{scriptSig}_i$ such that a function $\mathsf{Eval}(\textsc{scriptPubKey}_i, \textsc{scriptSig}_i)$ returns `true`, where Eval evaluates whether $\textsc{scriptSig}_i$ contains the correct fulfillment for the conditions encoded in $\textsc{scriptPubKey}_i$.

Inputs	Outputs
$\textsc{scriptPubKey}_0$, 2BTC	$\textsc{scriptPubKey}_2$, 1.0BTC
$\textsc{scriptPubKey}_1$, 3BTC	$\textsc{scriptPubKey}_3$, 3.5BTC
	$\textsc{scriptPubKey}_4$, 0.5BTC
$\textsc{scriptSig}_0$	
$\textsc{scriptSig}_1$	

Figure 4: Illustrative example of Bitcoin transaction.

Table 2: Description of the Script excerpts used in the Bitcoin transactions. Text in blue denotes scriptPubKey and orange denotes the corresponding scriptSig.

Lock's name	Script	Description of unlocking conditions
Pay-to-PubKey	⟨<pubKey> OP_CHECKSIG⟩ ⟨<sig>⟩	Including a signature ⟨<sig>⟩ of the Bitcoin transaction verifiable using the verification key ⟨<pubKey>⟩.
Pay-to-PubKeyHash	⟨OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG⟩ ⟨<sig> <pubKey>⟩	Including a verification key ⟨<pubKey>⟩ such that ⟨<pubKeyHash>⟩ = H(⟨<pubKey>⟩) and a signature ⟨<sig>⟩ of the Bitcoin transaction verifiable using the verification key ⟨<pubKey>⟩
Pay-to-ScriptHash	⟨OP_HASH160 H(*script*) OP_EQUAL⟩ ⟨<sig> <redeem_script>⟩	Include a ⟨redeem_script⟩ such that $H(redeem\_script) = H(script)$ and Eval (redeem_script, ⟨<sig>⟩) returns true.
Pay-to-Null	⟨OP_RETURN [data]⟩ ⟨<empty>⟩	Coins can never be unlocked. Data can contain up to 80 bytes [18].
Pay-to-Multisig	⟨OP_M <pubkey1> ... <pubkeyn> OP_N OP_CHECKMULTISIG⟩ ⟨<sig1> ... <sigm>⟩	Including M signatures ⟨<sig1> ... <sigm>⟩ of the Bitcoin transaction, verifiable using the corresponding verification keys ⟨<pubkey1> ... <pubkeyn>⟩

5.1.2 Possibilities for Encoding Data

Our approach consists on encoding the tagged message originated by the censored user as (some of) the conditions defined in the outputs scriptPubKey$_i$. We describe the different possible formats of the Bitcoin standard locking mechanism in Table 2. In the following, we describe how to re-use each of them to encode data within a transaction.

Pay-to-PubKey. Instead of including an actual verification key within the ⟨<pubKey>⟩ field, it is possible to encode 33 bytes of data simulating thereby an ECDSA verification key. This encoding, however, implies the loss of locked coins as it is not feasible to come up with a signing key corresponding to the data encoded as verification key.

Pay-to-PubKeyHash. Instead of including the 20 bytes corresponding to the hash of a verification key within the ⟨<pubKeyHash>⟩ field, it is possible to encode 20 bytes of data. This encoding does not restrict the encoded data to an ECDSA verification key. Nevertheless, this encoding also implies the loss (or burnt in Bitcoin terms) of the locked coins since it is not feasible to come up with the pre-image of a random hash value.

Pay-to-ScriptHash. Similar to the Pay-to-PubkeyHash, it is possible to encode 20 bytes of data replacing the field H(*script*). As before, this approach allows the inclusion of arbitrary random data but implies the loss of the locked coins.

Pay-to-Null. By definition of the lock mechanism, it is possible to encode up to 80 bytes of data within the field ⟨[data]⟩. Although, this lock mechanism provides the maximum bandwidth so far, it also implies the loss of the locked coins.

Pay-to-Multisig. As only M verification keys are actually used in this lock mechanism, it is possible to encode 33 bytes of data in each of the remaining $N-M$ keys, simulating thereby an $N-M$ ECDSA verification keys. The advantage of this encoding is that the locked coins can be unlocked as the necessary M verifications are not modified. It is possible, however, to encode 33 bytes of data in each of the N verification keys at the cost of losing the locked coins.

5.1.3 Setting R3C3 Parameters

The censored user could use any of the encoding methods presented in the previous section to send a small message once. However, it has two drawbacks. First, using a single scriptPubKey provides really limited bandwidth, and second, repeated usage of a certain lock mechanism could make the such usage pattern differ from the overall distribution of lock mechanisms in the Bitcoin blockchain, easing thereby a possible detection

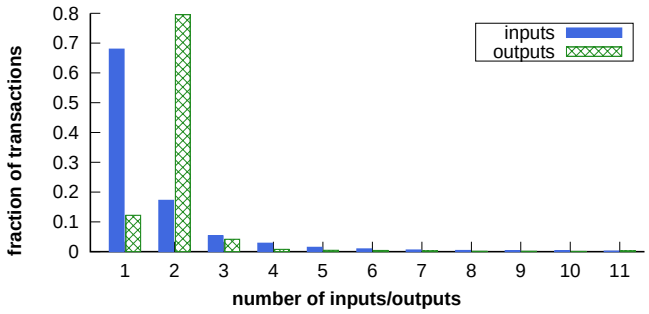


Figure 5: Distribution of number input and outputs in Pay-to-PubKeyHash transactions.

from the censor. In this state of affairs, in this section we describe how a censored user can get larger bandwidth without easing the task of the censor.

We have downloaded a snapshot of the Bitcoin blockchain containing blocks 0 to 511000, containing a total of $301,855,762$ Bitcoin transactions carried out from the inception of Bitcoin until the time of writing. From this dataset, we have extracted the number of appearances of each Bitcoin lock mechanism. In particular, we observe that Pay-to-PubKeyHash is used by the 88.73% of the transactions, Pay-to-ScriptHash by the 11.22% and the rest by less than 1% of the tansactions. From this distribution, we conclude that R3C3 should use Pay-to-PubKeyHash as the lock mechanism.

The available bandwidth can be enlarged by using more than one output in a Bitcoin transaction. To explore this possibility, we have extracted the distribution of the number of input and outputs used in Bitcoin transactions that contain Pay-to-PubKeyHash, obtaining the results shown in Figure 5. From this, we conclude that R3C3 should use Bitcoin transactions with one input and two outputs.

5.1.4 Implementation Encoding Scheme in Bitcoin

In this section, we describe our implementation of the encoding scheme using Bitcoin and following the guidelines mentioned in the previous section. For readability, we highlight the encoded fields in blue.

Notation. We denote by $\mathsf{ECDSA.KeyGen}$, $\mathsf{ECDSA.Sign}$, $\mathsf{ECDSA.Verify}$ the three algorithms for the ECDSA digital signature scheme as implemented in Bitcoin. In particular, $\mathsf{ECDSA.KeyGen}(\lambda)$ takes as input the security parameter λ and outputs a pair of keys vk, sk. The algorithm $\mathsf{ECDSA.Sign}(sk, tx)$ takes as input the private key sk and a transaction tx and outputs a signature σ. Finally, the algorithm $\mathsf{ECDSA.Verify}(vk, tx, \sigma)$ takes as input a public key vk, a transaction tx and a signature σ and returns $\top$ if σ is a signature on tx created by the corresponding private key sk. Otherwise, it returns $\bot$.

We denote

$$\langle \texttt{OP\_DUP OP\_HASH160 H (vk) OP\_EQUALVERIFY OP\_CHECKSIG} \rangle$$

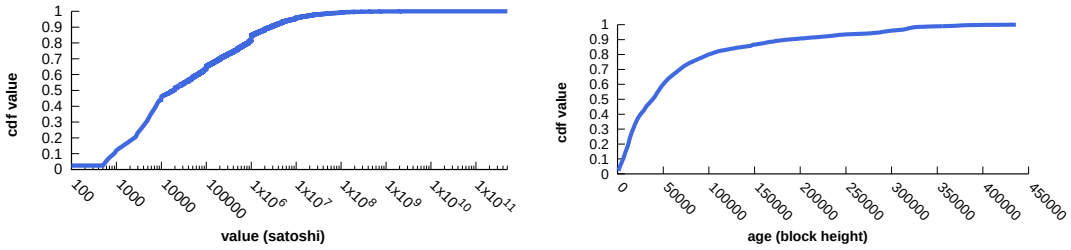
by $\textsc{scriptPubKey}(\mathsf{H}(vk))$. Similarly, we denote by $\textsc{scriptSig}(tx, sk, vk)$ the fulfillment condition defined as $\langle \mathsf{ECDSA.Sign}(tx, sk)\ vk \rangle$. Finally, we denote by $\mathsf{Extract}$ an extraction function such that $\mathsf{Extract}(\textsc{scriptPubKey}(x)) = x$ as well as $\mathsf{Extract}(\textsc{scriptSig}(tx, sk, vk)) = vk$.

$\{\boldsymbol{ctx}, \bot\} \leftarrow \mathbf{TxEncode}_f(\boldsymbol{cd}, \boldsymbol{ca})$**.** Parse $vk_u, \mathsf{H}(vk_p), ct \leftarrow cd$ and $sk_u \leftarrow ca$. If $|ct| > 20$ bytes, return $\bot$. Otherwise, create a Bitcoin transaction tx_1 as described below. Return tx_1.

Here we assume that vk_u has been funded eariler with x BTC and that ct has been padded with pseudorandom bytes so that $|ct| = 20$ bytes. The minimum value of x to not raise any suspicion from the censor is about $30,000$ Satoshi[2]. This amount of coins encode the coins to be burnt at the output $\textsc{scriptPubKey}_2$ as well as the amount of coins required by the decoder to pay for the transaction fee of the response covertext.

tx_1	
Inputs	Outputs
$\textsc{scriptPubKey}_1(\mathsf{H}(vk_u))$, x BTC	$\textsc{scriptPubKey}_2(ct)$, γ_1BTC $\textsc{scriptPubKey}_3(\mathsf{H}(vk_p))$, $(\text{x} - \gamma_1)$BTC
$\textsc{scriptSig}(tx_1, sk_u, vk_u)$	

[2]Each BTC is 10^8 Satoshi.



(a) value (b) age

Figure 6: CDF of the value and age, in the outputs of Pay-to-PubKeyHash transactions that contain only one input and two outputs. The value is given in Satoshi (a Satoshi is 10^{-8} BTC). The age is given in block height (on average each block in the Bitcoin network is created every 10 minutes).

$\{\boldsymbol{rtx}, \perp\} \leftarrow \mathbf{TxEncode}_b(\boldsymbol{rd}, \boldsymbol{ra})$**.** Parse $ct, vk_p \leftarrow rd$ and $sk_p \leftarrow ra$. If $|ct| > 40$ bytes, return $\perp$. Otherwise, create a Bitcoin transaction tx_2 as described below. Return tx_2.

As before, here we assume that ct has been padded with pseudorandom bytes so that $|ct| = 40$ bytes.

tx_2	
Inputs	Outputs
$\text{SCRIPTPUBKEY}_1(\mathsf{H}(vk_p))$, x BTC	$\text{SCRIPTPUBKEY}_2(ct[0:19])$, γ_2BTC $\text{SCRIPTPUBKEY}_3(ct[20:39])$, $(\text{x} - \gamma_2)$BTC
$\text{SCRIPTSIG}(tx_2, sk_p, vk_p)$	

$\{\boldsymbol{cd}, \perp\} \leftarrow \mathbf{TxDecode}_f(\boldsymbol{ctx})$**.** If ctx does not have one input and two outputs, return $\perp$. If any of the lock mechanisms is not of the type Pay-to-PubKeyHash, return $\perp$.
Otherwise, compute $ct \leftarrow \mathsf{Extract}(\text{SCRIPTPUBKEY}_2(ct))$. Compute $vk_u \leftarrow \mathsf{Extract}(\text{SCRIPTSIG}(tx, sk_u, vk_u))$. Compute $\mathsf{H}(vk_p) \leftarrow \mathsf{Extract}(\text{SCRIPTPUBKEY}_3(\mathsf{H}(vk_p)))$. If vk_p does not have at least 16,000 Satoshi associated with it, return $\perp$ as it does not contain enough coins to pay for the response transaction fee. Otherwise, return the tuple $cd := (vk_u, \mathsf{H}(vk_p), ct)$.

$\{\boldsymbol{rd}, \perp\} \leftarrow \mathbf{TxDecode}_b(\boldsymbol{rtx})$**.** If rtx does not have one input and two outputs, return $\perp$. If any of the lock mechanisms is not of the type Pay-to-PubKeyHash, return $\perp$. Otherwise, compute $ct[0:19] \leftarrow \mathsf{Extract}(\text{SCRIPTPUBKEY}_2(ct[0:19]))$ and $ct[20:39] \leftarrow \mathsf{Extract}(\text{SCRIPTPUBKEY}_3(ct[20:39]))$. Compute $vk_p \leftarrow \mathsf{Extract}(\text{SCRIPTSIG}(tx, sk_p, vk_p))$. Return the tuple $rd := (ct, vk_p)$.

Theorem 5 (Bitcoin Encoding Scheme Correctness)**.** *Let* $\mathsf{ECDSA.KeyGen}, \mathsf{ECDSA.Sign}, \mathsf{ECDSA.Verify}$ *be correct ECDSA digital signature scheme as implemented in Bitcoin. Let* $\mathsf{Extract}$ *be correct extract function for* SCRIPTPUBKEY *and* SCRIPTSIG*. Then, Bitcoin encoding scheme is correct according to Definition 5.*

Proof. We start by showing that condition 1 holds. In particular, given $ctx \leftarrow \mathsf{TxEncode}_f(cd, ca)$ we need to show that $\mathsf{TxDecode}_f(ctx)$ return cd^* such that $cd = cd^*$.

By the correctness of $\mathsf{Extract}$ function $ct^* = ct$, $vk_u^* = vk_u$ and $\mathsf{H}(vk_p^*) = \mathsf{H}(vk_p)$. Moreover, as H is collision-resistant, $vk_p^* = vk_p$. If $\mathsf{TxDecode}_f$ does not return $\perp$, this proves that both functionalities are correct. Now we show that $\mathsf{TxDecode}_f$ does not return $\perp$.

Since $\mathsf{TxEncode}_f$ has not returned $\perp$ it means that $|ct|$ is exactly 20 bytes (after padding) and sufficient fund has been associated to the accounts. Upon receiving a transaction, $\mathsf{TxDecode}_f$ checks the number of inputs and outputs. tx_1 has exactly one input and two outputs, along with sufficient funds, therefore $\mathsf{TxDecode}_f$ will not return $\perp$.

The condition 2 holds following similar arguments. This concludes the proof.

□

5.1.5 System Discussion

Cost. R3C3 (BTC) requires to pay two transaction fees as well as burn coins three times because the SCRIPTPUBKEY outputs used to encode the ciphertexts are no longer spendable. At the time of writing, using an online transaction fee estimator [3], the fee for a transaction is about 11,000 Satoshi (0.88 USD). If the

censored user is willing to wait four hours for its transaction to get into the blockchain, the cost is reduced to 900 Satoshi (0.07 USD). Furthermore, to find the minimum value for burning coins without raising any suspicious, we investigated all the Pay-to-PubKeyHash transactions with one input and two outputs. Figure 6 shows the CDF value of the amounts in the outputs of such transactions. To blend with at least 25% of the outputs the burned amount should be at least $2,500$ Satoshi.

Undetectability. The verification key vk is included in a field where a ECDSA verification key is expected and the ct bytes are encoded as the output of a hash function. Therefore undetectability is preserved under the assumption that the output of a hash function is indistinguishable from a uniform random number.[3]

Usage Pattern. Each transaction used in R3C3 (BTC) is structurally identical to the most widely used transactions included in the Bitcoin blockchain. The pattern of a transaction with two outputs, one with a much smaller value than the other, is highly used in Bitcoin as it represents a user that pays a little amount to the payee and gets the rest back in the *change address*. Finally, R3C3 (BTC) creates several outputs that can never be spent. This, however, remains in tune with the current usage of the Bitcoin blockchain. In particular, as shown in Figure 6, around 50% of unspent outputs were created more than one year ago.

Harvesting-Resistance. By construction of R3C3 (BTC), the censored user funds the address represented by vk_p in the protocol and whose private key is known by the decoder only. Therefore, the decoder can use the coins at vk_p to send a covertext to the censored user without investing any of its coins. If such address is not funded, the decoder aborts the protocol.

Efficiency. The decoder has to read all the transactions in the blockchain until it finds a transaction ctx such that $\mathsf{SBDecode}_f(ctx)$ does not return $\perp$. This is, however, inevitable. The censored user only has to look for the transaction rtx where coins associated to vk_p are spent as this is the only transaction where the decoder might have added the expected response data rd.

Compatibility. R3C3 (BTC) uses two transactions following the format defined in the Bitcoin protocol. Therefore, R3C3 (BTC) preserves compatibility.

Bandwidth. R3C3 (BTC) uses scriptPubKey fields to encode the data. Each scriptPubKey field is leveraged to encode exactly 20 bytes. This provides a 20 byte bandwidth for the challenge covertext, as one of the two scriptPubKey fields is used. On the other hand, the response covertext has a 40 byte bandwidth as it uses both of scriptPubKey fields to encode encrypted data.

5.2 Encoding Scheme in Zcash

5.2.1 Addresses and Transactions

Ben-Sasson et al. [66] proposed Zerocash, a privacy-preserving cryptocurrency scheme. The core idea behind Zerocash has been implemented in Zcash [25, 50]. Nevertheless, the implementation slightly differs from the original description in Zerocash paper. Therefore, in this section, we focus our description on the cryptocurrency as detailed in [66] and extend the implementation details when it applies.

Zerocash supports two types of addresses: *shielded* and *unshielded*. An unshielded address is defined in the same terms as a Bitcoin address and supports the same type of transactions. Therefore, it is possible to extend the same approach as in Bitcoin by using unshielded addresses in Zerocash.

The main difference with Bitcoin resides on the shielded addresses. A shielded address (or *coin*) is a tuple of the form $(pk, \mathrm{x}, \rho, r, s, com)$, where pk is a public key generated as $\mathsf{PRF}_{sk}(0)$ with PRF being a pseudo-random function and sk being a private key; x is the value associated to this coin, ρ, r and s are random seeds and com denotes a commitment that represents the coin. We describe the format of com later in this section.

Zerocash supports two type of transactions: *minting* and *pouring* transactions. A mint transaction is used to create new coins while a pouring transaction transfers the value of a certain coin into a fresh coin. Therefore, a coin can be used only once in the blockchain.

Figure 7 (top) shows an illustrative example of minting transaction where a user with an unshielded address scriptPubKey_0 creates a coin $c := (pk, \mathrm{x}, \rho, r, s, com)$. For this transaction, the user locally samples and stores ρ, r and s, computes $k := \mathsf{Com}_r(pk||\rho)$ and $com := \mathsf{Com}_s(\mathrm{x}||k)$. The transaction thereby constructed is valid if $\mathsf{Eval}(\text{scriptPubKey}_0, \text{scriptSig}_0)$ returns `true`, the value x is the same in the input and the output, and $com = \mathsf{Com}_s(\mathrm{x}||k)$. Here, Eval denotes a function that returns `true` if scriptSig contains the correct fulfillments

[3]This is possible if we model the hash function as a random oracle.

for the conditions expressed in scriptPubKey. A minting transaction can have multiple unshielded inputs and multiple shielded outputs.

An illustrative example of pouring transaction is depicted in Figure 7 (bottom). Omit for a moment the unshielded addresses. Then, the rest of the transaction is an example of a user that wants to split the coin (i.e., c^{old}) created in the minting transaction of Figure 7 into two new coins c_1^{new} and c_2^{new}. For that, she creates a single shielded output $(rt, sn^{old}, com_1^{new}, ct_1, com_2^{new}, ct_2, h, vk^*, \sigma^*, \Pi_{\mathsf{pour}})$ as follows: rt denotes the root of a Merkle tree whose leafs contains all the commitments $\{com_i\}$ included so far in the blockchain; $sn^{old} := \mathsf{PRF}_{sk^{old}}(\rho)$ is the serial number associated to the coin being spent; com_1^{new} and com_2^{new} are the commitments formed as described for com but for new values x_1^{new} and x_2^{new}. The values ct_1^{new} and ct_2^{new} are two ciphertext that contain the corresponding $(\mathrm{x}_i^{new}, \rho_i^{new}, r_1^{new}, s_1^{new})$. These ciphertexts are encrypted for the corresponding payee. In this manner, the payee can locally reconstruct the complete coin information as $c_i^{new} := (pk_i^{new}, \mathrm{x}_i^{new}, \rho_i^{new}, r_1^{new}, s_1^{new}, com_i^{new})$. Finally, vk^* is a fresh ECDSA verification key, $h := \mathsf{PRF}_{sk^{old}}(\mathsf{H}(vk^*))$, σ^* is a signature of the complete output under sk^*, and Π_{pour} is a zero-knowledge proof of the correctness of the output (e.g., sn^{old} corresponds to one of the shielded coins ever created or σ^* can be correctly verified using vk^*). We refer the reader to [66] for a detailed explanation.

A pouring transaction can have several unshielded inputs and any combination of shielded and unshielded outputs. A pouring transaction is valid if, for every shielded output, σ^* is a valid signature under verification key vk^* and Π_{pour} correctly verifies; additionally, the rest of the transaction must be valid as defined for a Bitcoin transaction.

5.2.2 Possibilities for Encoding Data

Unshielded addresses in Zerocash are handled in the same manner as described in Section 5.1 for Bitcoin and therefore we do not discuss them here. In a shielded output, there are several fields such that rt, sn^{old} or com_i^{new} that cannot be used to encode our data without making the zero-knowledge proof Π_{pour} fail. However, assume that a user creates a transaction to send a coin to herself, then the data encrypted in ct_i is not required as the intended payee is the user itself. Our insight then consists of encoding our data as the different ciphertexts available in the shielded outputs for coins sent to the user herself. An important detail here is that this ciphertext field as defined in the Zcash implementation contains an ephemeral public key for a Diffie-Hellman key exchange, followed by a bitstring of encrypted data with the corresponding symmetric key. The portion of encrypted data constitutes 584 bytes that can be reused to encode steganographic data in our system.

5.2.3 Setting R3C3 Parameters

We have downloaded a snapshot of the Zcash blockchain that contains blocks 0 to 277,000, containing a total of 2,314,139 transactions (including the coinbase transactions). These transactions have been carried out since the inception of Zcash until the time of writing. From these datasets, we have studied the following parameters.

First, we extracted the number of shielded and unshielded outputs used in Zcash transactions. We obtain that 8.1% of outputs are shielded while the rest are unshielded. Although, small yet, this observation shows that shielded addresses have started to be used and as of the time of this study, more than 165 thousand shielded transactions have been made (that is, more than 8% of the total). Therefore, we use shielded addresses in the design of R3C3 in Zerocash to exploit the potential of privacy-preserving transactions.

Second, we study the number of new coins that are created at each shielded output and we observe that there exist two new coins per output, and therefore two pairs of (com, ct) at all the shielded outputs. Third, we study the distribution of additional unshielded input/outputs in Zcash shielded transactions. We observe from Figure 8, that around 90% of shielded Zcash transactions also included one unshielded input and that around 10% include one additional unshielded output.

Finally, we observe that in the Zcash implementation, shielded outputs in a minting transaction are formatted

Inputs	Outputs
scriptPubKey$_0$, x ZEC	(x, k, s, com)
scriptSig$_0$	

Inputs	Outputs
scriptPubKey$_0$, x_0 ZEC	scriptPubKey$_1$, x_1 ZEC $(rt, sn^{old}, com_1^{new}, ct_1, com_2^{new}, ct_2, h, vk^*, \sigma^*, \Pi_{\mathsf{pour}})$
scriptSig$_0$	

Figure 7: Illustrative example of minting (left) and pouring (right) transaction in Zerocash.

tx_1	
Inputs	Outputs
$\text{SCRIPTPUBKEY}_1(\mathsf{H}(vk_u))$, x ZEC	$(rt, sn^{old}, com_1^{new}, ct[0:583], com_2^{new}, ct[584:1167], h, vk', \sigma, \Pi_{\mathsf{pour}})$ $\text{SCRIPTPUBKEY}_2(\mathsf{H}(vk_p))$, x_y ZEC
$\text{SCRIPTSIG}(tx, sk_u, vk_u)$	

tx_2	
Inputs	Outputs
$\text{SCRIPTPUBKEY}(\mathsf{H}(vk_p))$, x^* ZEC	$(rt, sn_1^{old}, com_3^{new}, ct[0:583], com_4^{new}, ct[584:1167], h', vk'', \sigma', \Pi'_{\mathsf{pour}})$
$\text{SCRIPTSIG}(tx_2, sk_p, vk_p)$	

Table 3: Example of covertexts produced by $\mathsf{TxEncode}_f$ (top) and $\mathsf{TxEncode}_b$ (bottom) algorithms for Zerocash.

identically to shielded outputs in pouring transactions to make them indistinguishable from each other. In summary, most Zcash shielded transactions contain one additional unshielded input and possibly an additional unshielded output. We use this conclusion in the construction of R3C3 for Zerocash.

5.2.4 Implementation Encoding Scheme in Zcash

Next, we describe our instantiation of the encoding scheme in Zcash, following the guidelines mentioned in the previous section.

Notation. We denote by the terms ZC.KeyGen, ZC.Sign, ZC.Verify the three algorithms for the Ed25519 digital signature scheme as implemented in Zerocash. We denote by Extract an extraction function working as follows: $\mathsf{Extract}(\text{SCRIPTPUBKEY}(x)) = x$ as well as $\mathsf{Extract}(\text{SCRIPTSIG}(tx, sk, vk)) = vk$. Additionally, on input a field f and a transaction tx, it returns the value of the field f if present in the shielded output of the transaction tx.

$\{\boldsymbol{ctx}, \perp\} \leftarrow \mathbf{TxEncode}_f(\boldsymbol{cd}, \boldsymbol{ca})$**.** Parse $vk_u, \mathsf{H}(vk_p), ct \leftarrow cd$ and $sk_u \leftarrow ca$. If $|ct| > 1168$ bytes, return $\perp$. Otherwise, create a Zcash transaction tx_1 as shown in Table 3 and return tx_1. Here, we assume that vk_u has been funded earlier with x ZEC and that there exists an old coin with a value x^{old}, previously funded in a shielded output, whose serial number is sn^{old}.

The pair $(com_1^{new}, ct_1^{new})$ is set to an honest shielded coin for the censored user to get the remaining coins back. Therefore, com_1^{new} is wellformed committing to a coin with value $\mathrm{x} + \mathrm{x}^{old} - \mathrm{x}_y$. However, as the censored user is sending coins to herself, ct_1 is not required and can be used to encode $ct[0:583]$. Moreover, we include the public key vk_p encoded in SCRIPTPUBKEY_2 similar to what has been defined in the encoding for Bitcoin. The value x_y must be enough to pay for the transaction fee. Finally, the pair (com_2^{new}, ct_2) is used to encode the rest of the ciphertext ct. For that, com_2^{new} is set to a commitment for a coin with value $\mathrm{x} = 0$ and set $ct_2^{new} := ct[584:1167]$. If $|ct| < 1168$ bytes, ct is padded with pseudorandom bytes. Finally, a fresh key pair $vk', sk' \leftarrow \mathsf{ZC.KeyGen}(\lambda)$ is generated and used for the signature σ and the hash h of the shielded output.

$\{\boldsymbol{rtx}, \perp\} \leftarrow \mathbf{TxEncode}_b(\boldsymbol{rd}, \boldsymbol{ra})$**.** Parse $ct, vk_p \leftarrow rd$ and $sk_p \leftarrow ra$. If $|ct| > 1168$ bytes, return $\perp$. Otherwise, create a transaction tx_2 as shown in Table 3 and return tx_2.

This transaction spends the coins on vk_p previously funded by the censored user in tx_1. As before, the pair (com_3^{new}, ct_3) and (com_4^{new}, ct_4) are used for encoding the ciphertext ct as follows. First, com_3^{new} and com_4^{new}

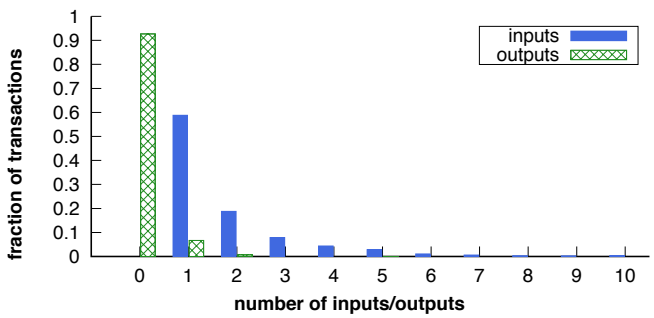


Figure 8: Distribution of the unshielded inputs and outputs for Zcash shielded transactions.

are two commitment to two different coins, both with value $\mathrm{x} = 0$. Then ct_3 and ct_4 are set to $ct[0:583]$ and $ct[584:1167]$ respectively. If $|ct| < 1168$ bytes, ct is padded with pseudorandom bytes. Finally, a fresh pair $vk'', sk'' \leftarrow \mathsf{ZC.DeriveKey}(\lambda)$ is used for the signature σ' and the hash h' of the shielded output.

$\{\boldsymbol{cd}, \perp\} \leftarrow \mathbf{TxDecode}_f(\boldsymbol{ctx})$**.** If ctx does not have an unshielded input, an unshielded output and a shielded output, return $\perp$. Otherwise, Compute $ct[0:583] \leftarrow \mathsf{Extract}(ctx, ct_1)$, $ct[584:1167] \leftarrow \mathsf{Extract}(ctx, ct_2)$, $\mathsf{H}(vk_p) \leftarrow \mathsf{Extract}(\textsc{scriptPubKey}(\mathsf{H}(vk_p)))$ and $vk_u \leftarrow \mathsf{Extract}(\textsc{scriptSig}(tx, sk_u, vk_u))$. Finally, return $cd := (vk_u, \mathsf{H}(vk_p), ct)$.

$\{\boldsymbol{rd}, \perp\} \leftarrow \mathbf{TxDecode}_b(\boldsymbol{rtx})$**.** If rtx does not have an unshielded input and a shielded output, return $\perp$. Otherwise, compute $ct[0:583] \leftarrow \mathsf{Extract}(rtx, ct_3)$ and $ct[584:1167] \leftarrow \mathsf{Extract}(rtx, ct_4)$. Extract $vk_p \leftarrow \mathsf{Extract}(\textsc{scriptSig}(rtx, sk_p, vk_p))$. Finally, return $rd := (ct, vk_p)$.

Theorem 6 (Zerocash Encoding Scheme Correctness)**.** *The Zerocash encoding scheme is correct according to Definition 5.*

Proof. We start by showing that condition 1 holds. In particular, given $ctx \leftarrow \mathsf{TxEncode}_f(cd, ca)$ we need to show that $\mathsf{TxDecode}_f(ctx)$ return cd^* such that $cd = cd^*$.

By the correctness of $\mathsf{Extract}$ function $ct[0:583]^* = ct[0:583]$, $ct[584:1167]^* = ct[584:1167]$,$vk_u^* = vk_u$ and $\mathsf{H}(vk_p^*) = \mathsf{H}(vk_p)$. Moreover, as H is collision-resistant, $vk_p^* = vk_p$. If $\mathsf{TxDecode}_f$ does not return $\perp$, this proves that both functionalities are correct. Now we show that $\mathsf{TxDecode}_f$ does not return $\perp$.

Since $\mathsf{TxEncode}_f$ has not returned $\perp$ it means that $|ct|$ is exactly 1168 bytes (after padding) and sufficient fund has been associated to the accounts. Upon receiving a transaction, $\mathsf{TxDecode}_f$ checks to have an unshielded input and shielded output, along the fee associated to the unshielded output. tx_1 has exactly one unshielded input and a shielded output, along with sufficient funds in the unshielded output, therefore $\mathsf{TxDecode}_f$ will not return $\perp$.

The condition 2 holds following similar arguments. This concludes the proof.

□

5.2.5 System Discussion

Cost. R3C3 (ZEC) requires to pay only two transaction fees. The rest of the coins are sent back to the censored user using the shielded coins. The price of the transaction fee is 0.0001 ZEC (0.03 USD).

Undetectability. The random bitstrings are included in the part of the ciphertexts reserved for the output of a symmetric key encryption algorithm. Moreover, we fit the censored user's verification key and the shared public key for the pre-payment within the slot reserved for a public key in the scriptSig.

Usage Pattern. R3C3 over Zerocash uses a transaction pattern that is followed by around 10% of the current transactions. Moreover, as values are hidden within the shielded address, the values cannot be used by the censor to distinguish between covertext and other transactions.

Harvesting-Resistance. An adversary must pay for each protocol execution. Therefore, the adversary harvesting capability is bounded by the number of coins that he possesses.

Efficiency. The decoder has to read all the transactions in the blockchain until it finds a transaction ctx such that $\mathsf{SBDecode}_f(ctx)$ does not return $\perp$. This is, however, inevitable. The censored user only has to look for the transaction rtx that spends the coins associated to the public key vk_p.

Compatibility. R3C3 (ZEC) uses two transactions following the format defined in the Zerocash protocol. Therefore, R3C3 (ZEC) preserves compatibility. In fact, we have prototyped the complete R3C3 (ZEC) as we describe in Section 7.

Bandwidth. R3C3 (ZEC) uses the ciphertext ct_i of a Zcash transaction, encoding 1168 bytes for the challenge and response covertexts.

5.3 Encoding Scheme in Monero

5.3.1 Addresses and Transactions

A Monero address is of the form (A, B), where A and B are two points of the curve ed25519, as defined in Monero. In order to avoid the linkability of different transactions that use the same public key, a payer does not send the coins to the Monero address of the payee. Instead, the payer derives a *one-time key* verification key

Inputs	Outputs
$(\{vk_i\}, \{\mathsf{Com}(x_i, r_i)\}, \{\Pi_i\})$	$((vk'_1, R'_1), \mathsf{Com}(x'_1, r'_1), \Pi'_1)$
	$((vk'_2, R'_2), \mathsf{Com}(x'_2, r'_2), \Pi'_2)$
σ_{ring}	

Figure 9: Illustrative example of a Monero transaction.

vk and an extra random point R, from the payee's address using the Monero Stealth Address mechanism [70]. Given an arbitrary pair (vk', R') set as an output in the blockchain, a payee can use her Monero address to figure out whether the pair (vk', R') was intended for her and, if so, compute the signing key sk' associated to vk'. We refer the reader to [70] for further reading about the stealth addresses mechanism.

The Monero coins are transferred by means of a Monero transaction. As in Bitcoin, a Monero transaction is divided in *inputs* and *outputs*. However, in contrast to Bitcoin, an input consists of a tuple $(\{vk_i\}, \{\mathsf{Com}(x_i, r_i)\}, \{\Pi_i\})$, where $\{vk_i\}$ denotes a *ring* of one-time keys that have previously appeared in the blockchain, each $\mathsf{Com}(x_i, r_i)$ denotes a cryptographic commitment of the amount of coins x_i locked in the corresponding public key vk_i, and each Π_i denotes a zero-knowledge range proof proving that x_i is in the range $[0 : 2^k]$, where k is a constant defined in the Monero protocol.

An output consists of a tuple $((vk', R'), \mathsf{Com}(\mathrm{x}', r'), \Pi')$, where each element is defined as aforementioned. Finally, a transaction contains a signature σ_{ring}, created following the *linkable ring signature scheme* defined in Monero [69]. A Monero transaction is valid if the following conditions hold. First, σ_{ring} demonstrates that the transaction sender knows the signing key sk^* associated to a verification key within the set $\{vk_i\}$ and such verification key has not been spent before.

Second, let x^* be the amount of coins encoded in the input commitment corresponding to the one-time key being spent. Then it must hold that x^* equals the sum of the output values. Finally, all zero-knowledge range proofs correctly verify that all amounts are within the expected range. An illustrative example of a Monero transaction with one input and two outputs is depicted in Figure 9.

5.3.2 Linkable Ring Signature Scheme

In the following, we describe the linkable ring signature scheme as implemented in Monero as it becomes crucial for our approach of encoding a covertext within a Monero transaction.

Let λ be the security parameter, and let $\mathbb{Z}_p$ be a group of primer order p. Moreover, let $\mathbb{G}$ be a cyclic group generated by the generator g as defined in the Monero protocol. Then, a linkable ring signature scheme is a tuple of algorithms $(\mathsf{LRS.KeyGen}, \mathsf{LRS.Sign}, \mathsf{LRS.Verify})$ defined as below:

- $vk, sk \leftarrow \mathsf{LRS.KeyGen}(\lambda)$: Sample $sk \leftarrow_\$ \mathbb{Z}_p$. Compute $vk := g^{sk}$. Return vk, sk.
- $\sigma \leftarrow \mathsf{LRS.Sign}((vk_1, \ldots, vk_{n-1}, vk_n), sk_n, m)$: Sample $r \leftarrow_\$ \mathbb{Z}_p$. Compute $\mathcal{I} := sk_n \cdot \mathsf{H}(vk_n)$ and $h_0 \leftarrow \mathsf{H}(m||g^r||\mathsf{H}(vk_n)^r)$. Then, sample $s_1, \ldots s_{n-1} \leftarrow_\$ \mathbb{Z}_p^{n-1}$ and compute the following series:

$$
\begin{aligned}
h_1 &:= \mathsf{H}(m||g^{s_1} \cdot vk_1^{h_0}||\mathsf{H}(vk_1)^{s_1} \cdot \mathcal{I}^{h_0}) \\
h_2 &:= \mathsf{H}(m||g^{s_2} \cdot vk_2^{h_1}||\mathsf{H}(vk_2)^{s_2} \cdot \mathcal{I}^{h_1}) \\
&\vdots \\
h_{n-1} &:= \mathsf{H}(m||g^{s_{n-1}} \cdot vk_{n-1}^{h_{n-2}}||\mathsf{H}(vk_{n-1})^{s_{n-1}} \cdot \mathcal{I}^{h_{n-2}})
\end{aligned}
$$

  Now, solve s_0 such that $\mathsf{H}(m||g^{s_0} \cdot vk_n^{h_{n-1}}||\mathsf{H}(vk_n)^{s_0} \cdot \mathcal{I}^{h_{n-1}} = h_0$. For that, solve $g^{s_{n-1}} \cdot vk_{n-1}^{h_{n-2}} = g^r$, getting that $s_0 = r - h_{n-1} \cdot sk_n$. Finally, output $\sigma := (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I})$.
- $\{\top, \bot\} \leftarrow \mathsf{LRS.Verify}((vk_1, \ldots, vk_n), m, \sigma)$: Parse

$$(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I}) \leftarrow \sigma$$

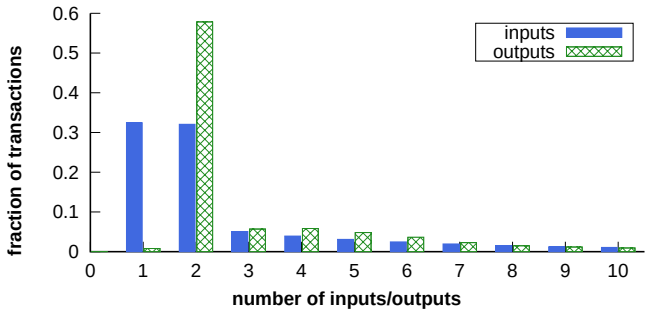


Figure 10: Distribution of inputs and outputs in Monero transactions.

and compute the series:

$$
\begin{aligned}
h_1 &:= \mathsf{H}(m||g^{s_1} \cdot vk_1^{h_0}||\mathsf{H}(vk_1)^{s_1} \cdot \mathcal{I}^{h_0}) \\
h_2 &:= \mathsf{H}(m||g^{s_2} \cdot vk_2^{h_1}||\mathsf{H}(vk_2)^{s_2} \cdot \mathcal{I}^{h_1}) \\
&\vdots \\
h_{n-1} &:= \mathsf{H}(m||g^{s_{n-1}} \cdot vk_{n-1}^{h_{n-2}}||\mathsf{H}(vk_{n-1})^{s_{n-1}} \cdot \mathcal{I}^{h_{n-2}}) \\
h_n &:= \mathsf{H}(m||g^{s_0} \cdot vk_n^{h_{n-1}}||\mathsf{H}(vk_n)^{s_0} \cdot \mathcal{I}^{h_{n-1}}
\end{aligned}
$$

Return $\top$ if $h_0 = h_n$. Otherwise, return $\bot$.

5.3.3 Possibilities for Encoding Data

As the information included in an input tuple must previously exist in the blockchain, we cannot modify them. Our approach consists then in crafting an output tuple that encodes certain amount of data while maintaining the invariants for the validity of the transaction. In particular, our insight is that if a user uses a Monero transaction to transfer coins to herself, she does not need to create the pair (vk, R) from her own address (A, B), using the stealth addresses mechanism. Instead, she can simply create a fresh vk and use the point R to encode data. However, the commitment and the range proof must be computed honestly, as otherwise transaction verification fails and the transaction does not get added to the blockchain. This allows to encode 32 bytes instead of the R point.

Further, we can encode extra data within the signature. In particular, we observe that the $\mathsf{LRS.Sign}$ algorithm samples $n-1$ random values from $\mathbb{Z}_p$. Our insight is that instead of sampling random numbers, we can use the corresponding bytes from a ciphertexts as random numbers. As values $s_1, \ldots, s_{n-1}$ are included in the signature, and therefore in the transaction, they allow to increase the bandwidth. Currently, the Monero protocol establishes that the rings must be composed of 5 public keys and therefore 4 random numbers can be used to encode data.

5.3.4 Setting R3C3 parameters

The construction previously sketched allows to encode one key in one output as well as 4 random points per signature. However, it is interesting to study the transaction pattern in the Monero blockchain to see what transaction format is actually used by Monero users and how to exploit it to encode data.

Towards this goal, we have downloaded a snapshot of the Monero blockchain that contains blocks 0 to $1,500,000$, containing a total of $2,376,896$ Monero transactions. These transactions have thereby been performed since the inception of Monero until the time of writing. From this dataset, we have extracted the distribution on the number of inputs and outputs used by each transaction, obtaining the results depicted in Figure 10. From this results, we conclude that R3C3 should be based on transactions with one or two inputs and two outputs. As a transaction contains as many signatures as inputs, we opt for transactions with two inputs and two outputs.

5.3.5 Implementation of the Encoding Scheme in Monero

In this section, we describe our instantiation of the encoding scheme in Monero, following the guidelines mentioned in the previous section.

Notation. Here, we denote by $\mathsf{LRS.KeyGen}, \mathsf{LRS.Sign}$ and $\mathsf{LRS.Verify}$ the key generation, signature and verification algorithms of the linkable ring signature scheme implemented in Monero (see Section 5.3.2).

$\{\boldsymbol{ctx}, \perp\} \leftarrow \mathbf{TxEncode}_f(\boldsymbol{cd}, \boldsymbol{ca})$**.** Parse $vk_u, vk_p, ct \leftarrow cd$ [4] and parse $sk_u \leftarrow ca$. If $|ct| > 256$ bytes, return $\perp$. Otherwise, create a Monero transaction tx_1 as described below. Return tx_1. The ciphertext ct is split in chunks of 32 bytes and each chunck is included as a value s_i in the signature. Here we assume that the input keys have been funded earlier with $\mathrm{x}'_1 + \mathrm{x}'_2$ XMR Additionally, we assume that ct has been padded with pseudorandom bytes so that $|ct| = 256$ bytes.

tx_1	
Inputs	Outputs
$(\{vk_1^i\}, \{\mathsf{Com}(\mathrm{x}_1^i, r_1^i)\}, \{\Pi_1^i\})$	$((vk_u, R'_1), \mathsf{Com}(\mathrm{x}'_1, r'_1), \Pi'_1)$
$(\{vk_2^i\}, \{\mathsf{Com}(\mathrm{x}_2^i, r_2^i)\}, \{\Pi_2^i\})$	$((vk_p, R'_2), \mathsf{Com}(\mathrm{x}'_2, r'_2), \Pi'_2)$
$\sigma_{ring} := (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I})$	
$\sigma'_{ring} := (s'_0, s'_1, \ldots, s'_{n-1}, h'_0, \mathcal{I}')$	

$\{\boldsymbol{rtx}, \perp\} \leftarrow \mathbf{TxEncode}_b(\boldsymbol{rd}, \boldsymbol{ra})$**.** Parse $ct, vk_p \leftarrow rd$ and parse $sk_p, \leftarrow ra$. If $|ct| > 256$ bytes, return $\perp$. Otherwise, create a Bitcoin transaction tx_2 as described below. Return tx_2.

As before, here we assume that ct has been padded with pseudorandom bytes so that $|ct| = 256$ bytes.

tx_2	
Inputs	Outputs
$(\{vk_1^i\} \cup vk_p, \{\mathsf{Com}(\mathrm{x}_1^i, r_1^i)\}, \{\Pi_1^i\})$	$((vk'_1, R'_1), \mathsf{Com}(\mathrm{x}'_1, r'_1), \Pi'_1)$
$(\{vk_2^i\}, \{\mathsf{Com}(\mathrm{x}_2^i, r_2^i)\}, \{\Pi_2^i\})$	$((vk'_2, R'_2), \mathsf{Com}(\mathrm{x}'_2, r'_2), \Pi'_2)$
$\sigma_{ring} := (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I})$	
$\sigma'_{ring} := (s'_0, s'_1, \ldots, s'_{n-1}, h'_0, \mathcal{I}')$	

$\{\boldsymbol{cd}, \perp\} \leftarrow \mathbf{TxDecode}_f(\boldsymbol{ctx})$**.** If ctx does not have two inputs and two outputs, return $\perp$. Otherwise, extract the ciphertext ct by concatenating the values in the signature, and the pair vk_u, vk_p from each of the outputs. If vk_p does not have at least 0.02 XMR associated coins, return $\perp$. Otherwise, return the tuple $cd := (vk_u, vk_p, ct)$.

$\{\boldsymbol{rd}, \perp\} \leftarrow \mathbf{TxDecode}_b(\boldsymbol{rtx})$**.** If rtx does not have two inputs and two outputs, return $\perp$. Otherwise, extract the ciphertext ct from the values included in the signature and extract vk_p from the input ring. Finally, return the tuple $rd := (vk_p, ct)$.[5]

Theorem 7 (Monero Encoding Scheme Correctness)**.** *The Monero encoding scheme is correct according to Definition 5.*

Proof. We start by showing that condition 1 holds. In particular, given $ctx \leftarrow \mathsf{TxEncode}_f(cd, ca)$ we need to show that $\mathsf{TxDecode}_f(ctx)$ return cd^* such that $cd = cd^*$.

By the correctness of $\mathsf{Extract}$ function $s_i^* = s_i, vk_u^* = vk_u$ and $vk_p^* = vk_p$. If $\mathsf{TxDecode}_f$ does not return $\perp$, this proves that both functionalities are correct. Now we show that $\mathsf{TxDecode}_f$ does not return $\perp$.

Since $\mathsf{TxEncode}_f$ has not returned $\perp$ it means that $|ct|$ is exactly 256 bytes (after padding) and sufficient fund has been associated to the accounts. Upon receiving a transaction, $\mathsf{TxDecode}_f$ checks the number of inputs and outputs. tx_1 has exactly one input and two outputs, therefore $\mathsf{TxDecode}_f$ will not return $\perp$.

The condition 2 holds following similar arguments. This concludes the proof.

□

[4] Using Monero as rendezvous requires that the argument $\mathsf{H}(vk_p)$ to be vk_p,however, the required changes are trivial and do not compromise any of the security properties.

[5] For ease of exposition, we assume that it is possible to guess where vk_p is situated within the keys conforming the ring. In practice, $\mathsf{TxDecode}_b$ returns the complete ring and the calling algorithm $\mathsf{SBDecode}_b$ checks each key individually looking for vk_p.

Field	Receiver	Amount	Fee	Signature	Data
Value	$\mathsf{H}(vk_2)$	x ETH	γ ETH	$\sigma(sk_1), vk_1$	0xfe...

Figure 11: Illustrative example of Ethereum transaction. The sender address is derived as the H(Ver. Key), in this case $\mathsf{H}(vk_1)$. Note, irrelevant fields have been omitted.

5.3.6 Discussion

Cost. R3C3 (XMR) requires to pay two transaction fees. This cost can be paid by the censored user by directly paying the fee for the first transaction and by including enough coins in vk_p so that the decoder can pay from there the transaction fee for the second transaction. On average the censored user needs to pay 0.003 XMR (0.77 USD) [14] for both of the transactions.

Undetectability. The verification key vk from censored user and decoder are included in a field where a verification key is expected and the ct bytes are encoded as values that must be drawn uniformly at random for the signature scheme to be secure.

Usage Pattern. Each transaction used in R3C3 (XMR) is structurally identical to more than 33% of the transactions included in the Monero blockchain.

Harvesting-Resistance. By construction of R3C3 (XMR), the censored user funds the address represented by vk_p in the protocol and whose private key is known by the decoder only. Therefore, the decoder can use the coins at vk_p to send the rtx to the censored user without investing any of its coins. If such address is not funded, the decoder aborts the protocol.

Efficiency. The decoder has to read all the transactions in the blockchain until it finds a transaction ctx such that $\mathsf{SBDecode}_f(ctx)$ does not return $\perp$. This is, however, inevitable. The censored user only has to look for the transaction rtx where vk_p is included in one of the rings in the input part of the transaction.

Compatibility. R3C3 (XMR) uses two transactions following the format defined in the Monero protocol. Therefore, R3C3 (BTC) preserves compatibility.

Bandwidth. R3C3 (XMR) uses four s_i values in each of the two signatures, where each s_i value is 32 bytes, provides a total of 256 bytes of bandwidth. Usage of more public keys in the ring signature will provide more bandwidth, however, this diverges from the most common pattern.

5.4 Encoding Scheme in Ethereum

5.4.1 Addresses and Transactions

Similar to Bitcoin, an Ethereum address is composed of a pair of verification and signing ECDSA keys. The address is then represented by the hash of the verification key. Each transaction is associated with a signature field, composed of 32 bytes of signature using the signing key of the sender$\sigma(sk_1)$, along with a 32 bytes of the verification key vk_1. There are two main differences with Bitcoin addresses. First, there exist two types of addresses: *external addresses* and *contracts*. An external address holds a certain amount of ETH, the Ethereum native coin. A contract has associated a piece of software that implements a certain business logic. Second, there exist two types of transactions, defined by the type of address included in the *receiver* field.

If the receiver field is an external address, the transaction transfers ETH between addresses. For instance, the illustrative transaction example in Figure 11 deduces x ETH from the sender's balance $\mathsf{H}(vk_1)$ and includes them to the receiver's balance $\mathsf{H}(vk_2)$. The data field is then filled by the sender to include the description of the payment. Note that, unlike Bitcoin and Zerocash, addresses in Ethereum can be used multiple times.

The second type of transaction appears when the receiver is set to a contract address. For instance, if $\mathsf{H}(vk_2)$ is a contract address, the transaction illustrated in Figure 11 is used by the owner of $\mathsf{H}(vk_1)$ to invoke the smart contract associated to $\mathsf{H}(vk_2)$. The amount field includes then the number of ETH to be deduced from the sender's address, paying thereby for the fee associated to running the contract. Finally, the data field includes the input to the function defined in the smart contract.

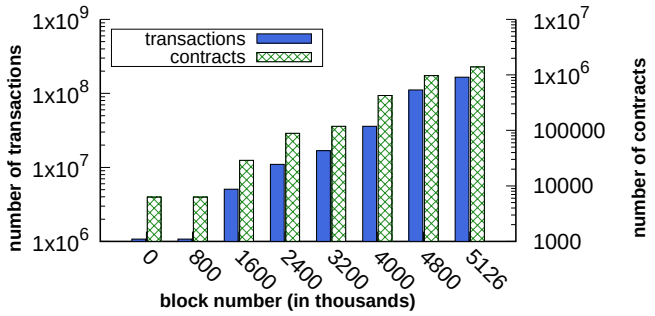


Figure 12: Number of transactions and contracts created at each block height. Block heights are shown as a multiple of thousand.

Contract Name	Contract Address	Number of Transactions
Etherdelta_2	0x8d12a197cb00d4747a1fe03395095ce2a5cc6819	8,660,074
BinanceWallet	0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be	1,850,473
CryptoKitties	0x06012c8cf97BEaD5deAe237070F9587f8E7A266d	1,645,667
Poloniex_2	0x209c4784AB1E8183Cf58cA33cb740efbF3FC18EF	1,612,733
EOSToken	0x86fa049857e0209aa7d9e616f7eb3b3b78ecfdb0	1,558,325

Table 4: Top five most used contracts in the Ethereum blockchain. They can be verified and examined in `https://etherscan.io`

5.4.2 Possibilities for Encoding Data

Similar to Bitcoin in Section 5.1, we can use `ETH` transfer transactions to encode 20 bytes of data in the receiver field given that external addresses are defined using the same script language as in Bitcoin. However, this encoding implies the loss of sent coins as it is not feasible to come up with a private key corresponding to the encoded data in the receiver field. Although useful and feasible in practice, we omit this possibility here in favor of exploring new alternative communication channels.

The contract invocation transaction can be exploited to encode information within the *data* field. This field is mostly used in transactions that are invoking a contract method or in the case of contract creation. In those cases, it contains the input data for the invoked contract or recently created contract.

5.4.3 Setting R3C3 parameters

We have downloaded a snapshot of the Ethereum blockchain that contains blocks 0 to 5126000, containing a total of 165,912,123 Ethereum transactions. These transactions have thereby been performed since the inception of Ethereum until the time of writing.

From this dataset, we have first extracted the transactions that create a contract and from them, the list of contracts created in Ethereum. As shown in Figure 12, we see that more than 1,386,985 contracts have been created in the Ethereum blockchain. Moreover, the growth rate of contract creation is similar to the growth rate of transactions. Therefore, contract creation is a potential possibility for encoding data to send to the decoder that we leave to explore as future work.

Instead, in this work we focus on another crucial use of data field other than contract creation, and that is the method invocation of an already deployed contract. We have extracted the transactions invoking contracts already created. Table 4 shows the top five most used contracts within Ethereum along with their addresses and number of transactions made to them. Among them, the contract *Etherdelta_2* is the most invoked ever with around 8.6 million transactions invoking it at the time of writing.

Therefore, we conclude that we can use the data field of a transaction invoking the *Etherdelta_2* contract to encode our data. Nevertheless, the approach described here can be easily extended if any other contract is invoked. *Etherdelta_2* is a contract deployed by a decentralized trading platform that exchanges different types of coins. Source code of this contract is available in [9,16]. Inspecting such code, there exist multiple methods that can be used to encode the data. For instance, the *testTrade* method checks whether a trade between two

different addresses can take place. The outcome of the method is reduced, either `true` or `false`, resulting in minimum suspicious for the censor compared to other methods with richer outcomes.

The signature of this method is as follow:

```
function testTrade(address tokenGet,
uint amountGet, address tokenGive, uint amountGive,
uint expires, int nonce, address user, uint8 v,
bytes32 r, bytes32 s, uint amount, address sender)
```

Each of the addresses are 160 bits and the uint values are 256 bits. To minimize the censorship capabilities from the censor, one can use valid addresses of Tokens, users and signatures. However, one can still encode data in the *amountGet*, *amountGive*, *expires*, *amount* field, resulting in a maximum of 1024 bits. In order to encode data so that it simulates realistic amounts, we encode 32 bits in each field and simulate floating point numbers. We thereby get a total bandwidth of 16 bytes by reusing these four fields to encode data. Although it may not be much bandwidth, it is enough to bootstrap the censored user.

5.4.4 Implementation of the Encoding Scheme in Ethereum

In this section, we describe our implementation of the encoding scheme in Ethereum following the guidelines mentioned in the previous section. For readability, we highlight the encoded fields in blue.

Notation. We denote by $\mathsf{ECDSA.KeyGen}$, $\mathsf{ECDSA.Sign}$, $\mathsf{ECDSA.Verify}$ the three algorithms for the ECDSA digital signature scheme as implemented in Ethereum. We denote $\mathsf{H}(vk)$ by $\textsc{address}(vk)$. We denote by $\mathsf{Extract}(tx, tag)$ a function that returns the value of the field tag included in tx, e.g. $\mathsf{Extract}(tx, Receiver) = \mathsf{H}(vk_2)$.

$\{\boldsymbol{ctx}, \perp\} \leftarrow \mathbf{TxEncode}_f(\boldsymbol{cd}, \boldsymbol{ca})$**.** Parse $vk_u, \mathsf{H}(vk_p), ct \leftarrow cd$ and $sk_u \leftarrow ca$. If $|ct| > 16$ bytes, return $\perp$. Otherwise, first create an Ethereum transaction tx_0 to fund the one-time key vk_p with x `ETH` as described below. The minimum value of x is 0.0003 `ETH`, which is equivalent to one transaction fee.

tx_0 (Fund one-time key)					
Field	Receiver	Amount	Fee	Signature	Data
Value	$\mathsf{H}(vk_p)$	x `ETH`	γ `ETH`	$\sigma(sk_u), vk_u$	—

Next, create an Ethereum transaction tx_1. Split the ciphertext ct is in chunks of 4 bytes. Each chunck is included as a value v_i in the low bit order of the *amountGet*, *amountGive*, *expires*, *amount* fields. Rest of the fields are not changed and will contain proper addresses and values as explained earlier. Additionally, we assume that ct has been padded with pseudorandom bytes so that $|ct| = 16$ bytes. $\mathsf{H}(vk_e)$ denotes the address of the *Etherdelta_2* contract. The value is set to zero and the only cost will be the transaction fee. Finally return $tx_0||tx_1$.

tx_1					
Field	Receiver	Amount	Fee	Signature	Data
Value	$\mathsf{H}(vk_e)$	0	γ `ETH`	$\sigma(sk_u), vk_u$	v_1, v_2, v_3, v_4

$\{\boldsymbol{rtx}, \perp\} \leftarrow \mathbf{TxEncode}_b(\boldsymbol{rd}, \boldsymbol{ra})$**.** Parse $ct, vk_p \leftarrow rd$ and $sk_p \leftarrow ra$. If $|ct| > 16$ bytes, return $\perp$. Otherwise, create an Ethereum transaction tx_2 as described below. Return tx_2.

As before, here we assume that ct has been padded with pseudorandom bytes so that $|ct| = 16$ bytes.

tx_2					
Field	Receiver	Amount	Fee	Signature	Data
Value	$\mathsf{H}(vk_e)$	0	γ `ETH`	$\sigma(sk_p), vk_p$	v_1, v_2, v_3, v_4

$\{\boldsymbol{cd}, \perp\} \leftarrow \mathbf{TxDecode}_f(\boldsymbol{ctx})$**.** Parse $tx_0||tx_1 \leftarrow ctx$. If tx_1 does not have $\mathsf{H}(vk_e)$ as the receiver, return $\perp$. Otherwise, extract the ciphertext ct by concatenating the values of the *amountGet*, *amountGive*, *expires*, *amount* fields as contained in $data \leftarrow \mathsf{Extract}(tx_1, data)$. Compute $\sigma(sk_u), vk_u \leftarrow \mathsf{Extract}(tx_1, signature)$. Compute $\mathsf{H}(vk_p) \leftarrow \mathsf{Extract}(tx_0, receiver)$. If $\mathsf{H}(vk_p)$ does not have at least 0.0003 `ETH` associated coins, return $\perp$. Otherwise, return the tuple $cd := (vk_u, \mathsf{H}(vk_p), ct)$.

$\{\boldsymbol{rd}, \perp\} \leftarrow \mathbf{TxDecode}_b(\boldsymbol{rtx})$**.** If rtx does not have $\mathsf{H}(vk_e)$ as the receiver and vk_p as the verification key, return $\perp$. Otherwise, extract the ciphertext ct by concatenating the values of the *amountGet*, *amountGive*, *expires*, *amount* fields as contained in $data \leftarrow \mathsf{Extract}(tx_2, data)$. Compute $\sigma(sk_p), vk_p \leftarrow \mathsf{Extract}(tx_2, signature)$. Return the tuple $rd := (vk_p, ct)$.

Theorem 8 (Ethereum Encoding Scheme Correctness)**.** *The Ethereum encoding scheme is correct according to Definition 5.*

Proof. We start by showing that condition 1 holds. In particular, given $ctx \leftarrow \mathsf{TxEncode}_f(cd, ca)$ we need to show that $\mathsf{TxDecode}_f(ctx)$ return cd^* such that $cd = cd^*$.

By the correctness of $\mathsf{Extract}$ function $ct^* = ct$, $vk_u^* = vk_u$ and $\mathsf{H}(vk_p^*) = \mathsf{H}(vk_p)$. Moreover, as H is collision-resistant, $vk_p^* = vk_p$. If $\mathsf{TxDecode}_f$ does not return $\perp$, this proves that both functionalities are correct. Now we show that $\mathsf{TxDecode}_f$ does not return $\perp$.

Since $\mathsf{TxEncode}_f$ has not returned $\perp$ it means that $|ct|$ is exactly 16 bytes (after padding) and sufficient fund has been associated to the accounts. Upon receiving $tx_0||tx_1$, $\mathsf{TxDecode}_f$ checks the recipient of the transaction tx_1 to be $\mathsf{H}(vk_e)$. tx_1, along with sufficient funds in address $\mathsf{H}(vk_p)$, therefore $\mathsf{TxDecode}_f$ will not return $\perp$.

The condition 2 holds following similar arguments. This concludes the proof.

□

5.4.5 System Discussion

Cost. R3C3 (ETH) requires to pay three transaction fees. The first transaction tx_0 is funded by the censored user with the value of one transaction fee γ (0.0003 ETH) [10], to be used in tx_2. To create tx_0 and tx_1, censored user needs to pay two transaction fees. Therefore, the total cost will be three transaction fees. Two for the challenge and one for the response.

Undetectability. The verification key vk is included in a field where a ECDSA verification key is expected and the ct bytes are encoded as the low order 32 bits in each field of the $uint256$ to simulate realistic amounts. Any amount in the low order 32 bits of the fields is likely and will not raise any suspicious to the censor.

Usage Pattern. Each transaction used in R3C3 (ETH) is sent to the most used contract within Ethereum namely *ehterdelta_2*. More than 8 million transactions have been made to this contract within the last year. The same technique explained in this work can be applied to similar contracts mentioned in Table 4. Just looking at the top 10 most used contracts we see that five of them are contracts created by exchanges. In total they have more than 15 million transactions. Therefore, transactions resulting from our technique blend in with at least 8% of all transactions in Ethereum.

Harvesting-Resistance. By construction of R3C3 (ETH), the censored user funds the address represented by vk_p in the protocol and whose private key is known by the decoder as well. Therefore, the decoder can use the coins at vk_p to send a covertext to the censored user without investing any of its coins. If such address is not funded, the decoder aborts the protocol.

Efficiency. The decoder has to read all the transactions in the blockchain until it finds a ctx, consisting of a pair of transactions tx_0 and tx_1, such that $\mathsf{SBDecode}_f(ctx)$ does not return $\perp$. This results in $O(n^2)$ possibilities. However, this can be improved in practice. The decoder can scan through all the transactions, namely tx_1, and extracts the sender vk_u of the transaction. Then, checks all the transactions that were sent by vk_u within a time frame and considers each to be tx_0. For a normal user, the number of such tx_0 transactions are constant. Resulting in a $O(n)$ runtime.

Furthermore, the censored user only has to look for the transaction rtx where coins associated to vk_p are spent as this is the only transaction where the decoder might have added the expected response data rd.

Compatibility. R3C3 (ETH) uses three transactions following the format defined in the Ethereum protocol. Therefore, R3C3 (ETH) preserves compatibility.

Bandwidth. R3C3 (ETH) In the chosen contract the censored user can use up to 1024 bits (four fields of 256 bits), however, to simulate realistic amounts to not raise the suspicion of the censor we use 32 bits of each field resulting 16 bytes in total.

6 Mining-Based Encoding

In this section, we focus on how a censored user and a decoder performing the miner functionality can use the structure of a block of transactions to convey information to the uncensored area.

Possibilities for Encoding Data. A miner creates the new blocks to be added to the blockchain. A block contains a set of transactions submitted by the blockchain users. In principle, the miner can decide on its own how to arrange the transactions within the block, except for the coinbase transaction, that must always be the first. Therefore, our insight is that the miner can arrange the transactions in such a manner that conveys information for the decoder.

Setting R3C3 parameters. We have studied the average number of transactions per block for different cryptocurrencies in Table 5. On average, Bitcoin uses the most number of transactions per block, followed by Ethereum. However, Bitcoin has the lowest block creation rate. Ethereum has the fastest block creation, followed by Zcash. While Ethereum provides 68 bytes per minute compared, Bitcoin provides 57 bytes per minute.

Implementation of the Encoding Scheme using Blocks. We denote by $\mathbf{tx'} \leftarrow \Pi(k, \mathbf{tx})$ a permutation function that takes as input a list of transactions sorted lexicographically $\mathbf{tx}$ and a permutation key k, and returns a permuted list of the transactions $\mathbf{tx'}$. Similarly, we denote by $k \leftarrow \mathsf{GetPerm}(\mathbf{tx}, \mathbf{tx'})$ an algorithm that on input a list of transactions and its permutation, returns the key k used for the permutation. We observe that both of these operations can be performed efficiently in practice [1]. For simplicity, we denote by $tx(vk, vk', \mathrm{x})$ a transaction that sends x coins from vk to vk'. In practice, we use this transaction to pay for the service provided by the decoder. The details of this transaction depend on the cryptocurrency used. Finally, we assume that the mechanism provides β bytes of bandwidth.

$\{\boldsymbol{ctx}, \perp\} \leftarrow \mathbf{TxEncode}_f(\boldsymbol{cd}, \boldsymbol{ca})$**.** Parse $vk_u, \mathsf{H}(vk_p), ct \leftarrow cd$ and $sk_u \leftarrow ca$. If $|ct| > \beta$ bytes, return $\perp$. Otherwise, create a block that contains the set of transactions $\mathbf{tx'} \cup tx(vk_u, vk_p, \mathrm{x})$, where $\mathbf{tx'} := \Pi(ct, \mathbf{tx})$. Finally, publish the block.

$\{\boldsymbol{rtx}, \perp\} \leftarrow \mathbf{TxEncode}_b(\boldsymbol{rd}, \boldsymbol{ra})$**.** Parse $ct, vk_p \leftarrow rd$ and $sk_p \leftarrow ra$. If $|ct| > \beta$ bytes, return $\perp$. Otherwise, create a block that contains the set of transactions $\mathbf{tx'} \cup tx(vk_p, vk', \mathrm{x})$, where $\mathbf{tx'} := \Pi(ct, \mathbf{tx})$. Finally, publish the block. Here vk' is a change address to recover the coins spent from vk_p.

$\{\boldsymbol{cd}, \perp\} \leftarrow \mathbf{TxDecode}_f(\boldsymbol{ctx})$**.** If ctx is not a block of transactions, return $\perp$. Otherwise, compute $ct \leftarrow \mathsf{GetPerm}(\mathbf{tx}, \mathbf{tx'})$. Moreover, extract vk_u and $\mathsf{H}(vk_p)$ from the extra transaction in the block of the form $tx(vk_u, vk_p, \mathrm{x})$. If x is not enough to pay for a transaction fee, return $\perp$. Otherwise, return the tuple $cd := (vk_u, \mathsf{H}(vk_p), ct)$.

$\{\boldsymbol{rd}, \perp\} \leftarrow \mathbf{TxDecode}_b(\boldsymbol{rtx})$**.** If ctx is not a block of transactions, return $\perp$. Otherwise, compute $ct \leftarrow \mathsf{GetPerm}(\mathbf{tx}, \mathbf{tx'})$. Moreover, extract vk_p from the extra transaction in the block of the form $tx(vk_p, vk', \mathrm{x})$. Finally, return the tuple $cd := (vk_p, ct)$.

Theorem 9 (Block based Encoding Scheme Correctness)**.** *The Miner encoding scheme is correct according to Definition 5.*

Proof. We start by showing that condition 1 holds. In particular, given $ctx \leftarrow \mathsf{TxEncode}_f(cd, ca)$ we need to show that $\mathsf{TxDecode}_f(ctx)$ return cd^* such that $cd = cd^*$.

By the correctness of Π and $\mathsf{GetPerm}$ function $ct^* = ct$, $vk_u^* = vk_u$ and $\mathsf{H}(vk_p^*) = \mathsf{H}(vk_p)$. Moreover, as H is collision-resistant, $vk_p^* = vk_p$. If $\mathsf{TxDecode}_f$ does not return $\perp$, this proves that both functionalities are correct. Now we show that $\mathsf{TxDecode}_f$ does not return $\perp$.

Cryptocurrency	Tx per Block	Bandwidth	Block Time
Bitcoin	592	575	600
Zcash	9	2	150
Ethereum	33	16	14
Monero	3	3 bit	120

Table 5: Comparison of cryptocurrencies in terms of average number of transactions per block, bandwidth (in bytes) provided per block and block creation rate (in seconds).

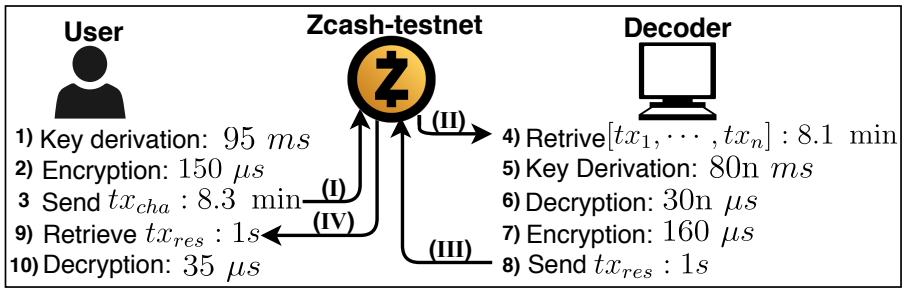


Figure 13: Performance of the R3C3 protocol in the Zcash.

Since $\mathsf{TxEncode}_f$ has not returned $\perp$ it means that $|ct|$ is less than β bytes and sufficient fund has been associated to $tx(vk_u, vk_p, \mathrm{x})$. Upon receiving a block b, $\mathsf{TxDecode}_f$ checks the last transaction $tx(vk_u, vk_p, \mathrm{x})$ to have sufficient funds. Therefore $\mathsf{TxDecode}_f$ will not return $\perp$.

The condition 2 holds following similar arguments. This concludes the proof.

□

Cost. The proposed solution imposes a small cost to the censored user that is compensated by the block creation reward. There is no const associated to the decoder. **Undetectability.** The transaction ordering within a block is arbitrary and defined by the miner. Some blockchains require the miner to place a child transaction after the parent transaction (output of parent transaction used as an input of the child transaction), if such transactions exist in the same block. This may decrease the bandwidth slightly and further study has to be done to investigate the frequency of such transactions in a block.

Harvesting-Resistance. In the block, censored user adds an extra transaction $tx(vk_u, vk_p, \mathrm{x})$, which is used to pay for the fees and services that decoder faces for the response. If such a transaction is not present in the block, then decoder refuses to respond.

Efficiency. Decoder retrieves all blocks in the blockchain until it finds a block of transactions ctx such that $\mathsf{SBDecode}_f(ctx)$ does not return $\perp$. The censored user only has to look for the blocks of transaction rtx that were created after the block ctx created by itself, where it includes a transaction from the sender vk_p.

Compatibility. Each block creation used in R3C3 (`MINER`) is structurally identical to the other created blocks as there is no specific guideline on how to arrange transactions in a block.

Bandwidth. We calculate the bandwidth of each block as $bandwidth = \log_2(N_{tx}!)$, where N_{tx} is the number of transactions in the block. Considering only one block, we see that Bitcoin provides the most bandwidth due to having more transactions within its block. However, we have to consider the rate of block creation as well. Although, Bitcoin has the most bandwidth for one block we need to consider the rate of block creation as well. Considering a 10 minute time frame, miner can potentially (mining all the blocks) gain a bandwidth of 685 bytes in the Ethereum blockchain respectively as compared to 575 bytes in Bitcoin.

7 Implementation and Evaluation

We have developed a prototypical python implementation to show the feasibility of our system. The implementation encompasses the complete encoding and decoding operations. We use the Charm-Crypto library [5] for the Diffie-Hellman key exchange and Cryptography [7] for the key derivation function. Given its higher bandwidth, we use Zcash testnet as rendezvous and we interact with it using the Insight API [11] and Zcash client [24].

7.1 Performance

We conducted our experiments on a machine with an Intel Core i7, 2.2 GHz processor and 16 GB RAM. For our evaluation, we divide our experiments in two separated tasks: the cryptographic operations required in the stego-bootstrapping scheme abstracting away the encoding scheme, and the instantiation of the encoding scheme using Zcash. The results are shown in Figure 13.

Cryptographic Operations in Stego-Bootstrapping Scheme. We study the four algorithms composing the stego-bootstrapping scheme. As shown in Figure 3, $\mathsf{SBEncode}_f$ and $\mathsf{SBDecode}_f$ require a key derivation procedure. Figure 13 shows that the process of creating a fresh pair of keys and deriving a shared key and symmetric key for encryption takes the censored user 95 milliseconds on average. After deriving the symmetric key, it takes the censored user 150 microseconds on average to encrypt the challenge message.

Upon retrieving a block of transactions, the decoder considers each of the transactions at a time and performs a key exchange to derive the shared and symmetric key in $\mathsf{SBDecode}_f$. After deriving the keys, decoder attempts to decrypt the cipher within the transactions. If the tag τ is present in the decrypted cipher then it knows that the transaction is from a censored user. The time of key derivation and decryption for each of the transactions is on average 80 milliseconds and 30 microseconds respectively.

Then, for each of the transactions from the censored users, decoder performs the $\mathsf{SBEncode}_b$ function, using the keys derived in $\mathsf{SBDecode}_f$ (all the keys including the decryption and encryption ones are generated in the $\mathsf{SBEncode}_f$ and $\mathsf{SBDecode}_f$). The encryption of each response message takes on average 160 microseconds. Lastly, the censored user performs $\mathsf{SBDecode}_b$ to decrypt the response message from the decoder, using the keys derived in $\mathsf{SBEncode}_f$. This process takes on average 35 microseconds.

The decoder can parallelize the key derivation and decryption of each transaction. In particular, all the transactions of a block can be decoded simultaneously if enough parallelization is available. Moreover, these performance results apply to the other instantiations since they follow the same key derivation procedure and the ciphertext length in all other instantiations is shorter than Zcash (and therefore it takes less time to encrypt and decrypt). These results show that even a decoder running in a commodity machine can simultaneously decode transactions from the different blockchain systems where R3C3 is used. This widens the possible anticensorship channels available for censored users simultaneously and hinders the task of the censor.

Encoding Scheme Instantiated in Zcash. After the covertext is created, the censored user signs it and sends it to the Zcash testnet. These operations are performed locally and take 8.3 minutes. The bottleneck is the import of a fresh private key to the wallet using the Zcash client [24]. After further investigation, we notice that the import of a fresh private key triggers a rescan of the entire blockchain and checks all the transactions for the address associated with that private key. The censored user and decoder can benefit from a customized wallet that does not perform such checks.

The decoder retrieves a block that contains n transactions. At this point, the decoder faces a similar bottleneck as the censored user in the previous step. On average, it takes the decoder 8.1 minutes to get the transaction from Insight API [11] and process it. We observe that after fresh keys have been imported, the response transaction takes 1 second on average.

We note that the elapsed times to send and retrieve the challenge transactions are long. The reason is due to the Zcash wallet [24] implementation for importing keys into the wallet. A better and more efficient implementation of the Zcash wallet can improve these times significantly. Additionally, part of the implementation will be dominated by the time of publishing a transaction in the blockchain. In the case of Zcash, on average it takes about 2.5 minutes to include a transaction in a block.

8 Related Work

The traditional censorship circumvention systems such such as VPNs [60, 63], Dynaweb [8], Ultrasurf [23], Lantern [12], Tor [38, 39], and other [32, 40] benefit from establishing proxy servers outside of the censored area. However, these systems are vulnerable to blockage. Censors actively scan and block the IP addresses of the proxies. Circumvention systems respond with introducing new IP addresses. A prominent example of such cat-and-mouse game is the Tor [39] and Great Firewall of China. Resulting in introducing mirrors [22], bridges [20] and secret entry servers [21] in the Tor system. At the same time multiple attacks, such as active probing and insider attacks have been proposed to discover the Tor bridges [4, 42, 55, 75, 77]. In recent years domain fronting [13, 44] has been introduced, as a way to resist IP address filtering. However, due to the high bandwidth and CPU usage it can be costly for the hosts [17]. To reduce the cost, we can benefit from the use content delivery networks (CDNs) namely CDNBrowsing [48, 81]. CDN's disadvantage is the unblocking of limited censored contents [81].

The most recent line of work in censor circumvention is decoy routing [27, 46, 52, 59, 78, 79]. Decoy routing, unlike the typical end-to-end approach, it is an end-to-middle proxy with no IP address. The proxy is located within the network infrastructure. Clients invoke the proxy by using public-key steganography to "tag" otherwise

ordinary sessions destined for uncensored websites. Message In A Bottle [56] follows a different anti-censorship approach. Similar to our work, it is considered a bootstrapping mechanism where the communication medium is the blog pings [74]. DEFIANCE [54] is an another bootstrapping mechanism that is proposed for finding Tor bridges. It suggest use of proof-of-life and proof-of-work to make the task of censors harvesting all the bridges harder. Our complementary work raises the bar for the censors by the inherent fees of cryptocurrencies.

All of these approaches are orthogonal to what we present in this paper. R3C3 exploits a new form of communication channel that has been widely developed only recently. Therefore, we believe it can coexist with current approaches and help augment the plethora of possibilities for anti-censorship.

9 Conclusions and Future Work

Despite the many academic and practical alternatives for censorship resistance, censorship remains today an important problem that hinders numerous people from freely accessing and communicating information. In this work, we explore the use of the widely deployed blockchain technologies as a communication channel in the presence of a censor and we observe that the blockchain transactions enable multiple communication channels offering interesting tradeoffs between bandwidth, costs and censorship resistance. Interestingly, we observe that blockchain components other than transactions can also be used to construct communication channels. For instance, we leverage the block creation process to build a communication channel.

However, in this work we are only scratching the tip of the iceberg. The different blockchain technologies are in continuous development and new features are being added continuously. These additions may come with yet unexplored possibilities to build a communication channel. For instance, the imminent deployment of off-chain payment channels [37, 64] adds new locking mechanism to Bitcoin and the alike cryptocurrencies with extra fields that can be used to encode extra covertext bytes. Therefore, we believe that this work sets the grounds for future research works exploring the use of blockchain for censorship resistance communications.

Acknowledgements

We thank Tim Ruffing and Siddharth Gupta for their efforts with a preliminary manuscript associated with the work. We thank Amir Houmansadr for encouraging suggestions on an early draft.

References

[1] Algorithm to generate all possible permutations of a list? https://stackoverflow.com/questions/2710713/algorithm-to-generate-all-possible-permutations-of-a-list. (Accessed May, 2018).

[2] Base58Check encoding - Bitcoin Wiki. https://en.bitcoin.it/wiki/Base58Check_encoding#Encoding_a_Bitcoin_address. (Accessed May, 2018).

[3] Bitcoin transaction fee estimator. https://estimatefee.com/. (Accessed May, 2018).

[4] Bridge Easily Detected by GFW. https://trac.torproject.org/projects/tor/ticket/4185. (Accessed May, 2018).

[5] Charm-crypto docs. https://jhuisi.github.io/charm/. (Accessed May, 2018).

[6] Cryptocurrency market capitalizations. https://coinmarketcap.com/. (Accessed May, 2018).

[7] Cryptography python library docs. https://cryptography.io. (Accessed May, 2018).

[8] Dynaweb. http://us.dongtaiwang.com/loc/download.en.php. (Accessed May, 2018).

[9] Etherdelta source code. https://etherscan.io/address/0x8d12a197cb00d4747a1fe03395095ce2a5cc6819#code. (Accessed May, 2018).

[10] Ethereum transaction fee estimator. https://z.cash/download.html. (Accessed May, 2018).

[11] Insight: Zcash testnet explorer. https://explorer.testnet.z.cash/. (Accessed May, 2018).

[12] Lantern. https://getlantern.org. (Accessed May, 2018).

[13] meek pluggable transport. https://trac.torproject.org/projects/tor/wiki/doc/meek. (Accessed May, 2018).

[14] Monero transaction fee estimator. https://z.cash/download.html. (Accessed May, 2018).

[15] Psiphon. https://www.psiphon3.com/en/index.html. (Accessed May, 2018).

[16] Read methods of etherdelta contract. https://etherscan.io/address/0x8d12a197cb00d4747a1fe03395095ce2a5cc6819#readContract. (Accessed May, 2018).

[17] Summary of meek's costs, july 2016. https://lists.torproject.org/pipermail/tor-project/2016-August/000690.html. (Accessed May, 2018).

[18] Talk Crypto Blog OP_RETURN 40 to 80 bytes. http://www.talkcrypto.org/blog/2016/12/30/op_return-40-to-80-bytes/. (Accessed May, 2018).

[19] TOR. https://www.torproject.org. (Accessed May, 2018).

[20] Tor:Bridges. https://www.torproject.org/docs/bridges. (Accessed May, 2018).

[21] Tor:hidden service protocol. https://www.torproject.org/docs/hidden-services. (Accessed May, 2018).

[22] Tor:mirrors. https://www.torproject.org/getinvolved/mirrors.html.en. (Accessed May, 2018).

[23] Ultrasurf. https://ultrasurf.us/. (Accessed May, 2018).

[24] Zcash client. https://z.cash/download.html. (Accessed May, 2018).

[25] Zcash project website. https://z.cash/. (Accessed May, 2018).

[26] Script - Bitcoin Wiki. https://en.bitcoin.it/wiki/Script, Accessed May, 2018.

[27] Amir Houmansadr, Giang T. K. Nguyen, M. C., and Borisov, N. Cirripede: Circumvention infrastructure using router redirection with plausible deniability. *Computer and Communications Security (CCS)* (2011), Pages 187–200.

[28] Anderson, R. J. Stretching the Limits of Steganography. In *Information Hiding* (1996), pp. 39–48.

[29] Aryan, S., Aryan, H., and Halderman, J. A. Internet Censorship in Iran: A First Look. In *USENIX Workshop on Free and Open Communications on the Internet, FOCI* (2013).

[30] Backes, M., and Cachin, C. Public-Key Steganography with Active Attacks. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC* (2005), pp. 210–226.

[31] Borisov, N., Danezis, G., and Goldberg, I. DP5: A private presence service. *PoPETs 2015*, 2 (2015), 4–24.

[32] Boyan, J. The anonymizer - protecting user privacy on the web. Computer-Mediated Communication Magazine 4, 1997.

[33] Burnett, S., Feamster, N., and Vempala, S. Chipping away at censorship firewalls with user-generated content. In *USENIX Security* (2010), pp. 29–29.

[34] Buterin, V., and Foundation, E. A next-generation smart contract and decentralized application platform. (Accessed May, 2018).

[35] Cash, D., Kiltz, E., and Shoup, V. The twin diffie-hellman problem and applications. *J. Cryptology 22*, 4 (2009), 470–504.

[36] De Cristofaro, E., Soriente, C., Tsudik, G., and Williams, A. Hummingbird: Privacy at the time of twitter. In *Security and Privacy (SP)* (2012), pp. 285–299.

[37] Decker, C., and Wattenhofer, R. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems (SSS)* (2015), pp. 3–18.

[38] Dingledine, R., and Mathewson, N. Design of a blocking-resistant anonymity system. Tech. rep., 2006.

[39] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. Tech. rep., Naval Research Lab Washington DC, 2004.

[40] Dyer, K. P., Coull, S. E., Ristenpart, T., and Shrimpton, T. Protocol misidentification made easy with format-transforming encryption. In *Computer and Communications Security (CCS)* (2013), pp. 61–72.

[41] Dyer, K. P., Coull, S. E., and Shrimpton, T. Marionette: A Programmable Network Traffic Obfuscation System. In *USENIX Security* (2015), pp. 367–382.

[42] Ensafi, R., Winter, P., Mueen, A., and Crandall, J. R. Analyzing the great firewall of china over space and time. *Proceedings on Privacy Enhancing Technologies (PoPET)*, 1 (2015), 61–76.

[43] Fazio, N., Nicolosi, A., and Perera, I. M. Broadcast Steganography. In *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track* (2014), pp. 64–84.

[44] Fifield, D., Lan, C., Hynes, R., Wegmann, P., and Paxson, V. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies (PoPET)*, 2 (2015), 46–64.

[45] Freire, E. S. V., Hofheinz, D., Kiltz, E., and Paterson, K. G. Non-interactive key exchange. In *Public-Key Cryptography - PKC* (2013), pp. 254–271.

[46] Frolov, S., Douglas, F., Scott, W., McDonald, A., VanderSloot, B., Hynes, R., Kruger, A., Kallitsis, M., Robinson, D. G., Schultze, S., et al. An isp-scale deployment of tapdance. In *Free and Open Communications on the Internet (FOCI)* (2017).

[47] Geddes, J., Schuchard, M., and Hopper, N. Cover Your ACKs: Pitfalls of Covert Channel Censorship Circumvention. In *Computer and Communications Security (CCS)* (2013), pp. 361–372.

[48] Holowczak, J., and Houmansadr, A. Cachebrowser: Bypassing chinese censorship without proxies using cached content. In *Computer and Communications Security (CCS)* (2015), pp. 70–83.

[49] Hopper, N. On Steganographic Chosen Covertext Security. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP* (2005), pp. 311–323.

[50] Hopwood, D., Bowe, S., Hornby, T., and Wilcox, N. Zcash Protocol Specification, 2018.

[51] Houmansadr, A., Brubaker, C., and Shmatikov, V. The Parrot Is Dead: Observing Unobservable Network Communications. In *Security and Privacy (SP)* (2013), pp. 65–79.

[52] Karlin, J., Ellard, D., Jackson, A. W., Jones, C. E., Lauer, G., Mankins, D., and Strayer, W. T. Decoy routing: Toward unblockable internet communication. In *Free and Open Communications on the Internet (FOCI)* (2011).

[53] Krawczyk, H. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology - CRYPTO* (2010), pp. 631–648.

[54] Lincoln, P., Mason, I., Porras, P. A., Yegneswaran, V., Weinberg, Z., Massar, J., Simpson, W. A., Vixie, P., and Boneh, D. Bootstrapping communications into an anti-censorship system. In *FOCI* (2012).

[55] Ling, Z., Luo, J., Yu, W., Yang, M., and Fu, X. Extensive analysis and large-scale empirical evaluation of tor bridge discovery. In *INFOCOM* (2012), pp. 2381–2389.

[56] Luca Invernizzi, C. K., and Vigna, G. Message in a bottle: Sailing past censorship. *Computer Security Applications* (2013), 39–48.

[57] Mohajeri Moghaddam, H., Li, B., Derakhshani, M., and Goldberg, I. Skypemorph: Protocol obfuscation for tor bridges. In *Computer and Communications Security (CCS)* (2012), pp. 97–108.

[58] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. (Accessed May, 2018).

[59] Nasr, M., Zolfaghari, H., and Houmansadr, A. The waterfall of liberty: Decoy routing circumvention that resists routing attacks. In *Computer and Communications Security (CCS)* (2017), pp. 2037–2052.

[60] Nobori, D., and Shinjo, Y. Vpn gate: A volunteer-organized public vpn relay system with blocking resistance for bypassing government censorship firewalls. In *Networked Systems Design and Implementation (NSDI)*.

[61] Parker, E. Can china contain bitcoin?

[62] Peck, M. Why the biggest bitcoin mines are in china.

[63] Perta, V., Barbera, M., Tyson, G., Haddadi, H., and Mei, A. A glance through the vpn looking glass: Ipv6 leakage and dns hijacking in commercial vpn clients. *Proceedings on Privacy Enhancing Technologies (PoPET)*, 1 (2015), 77–91.

[64] Poon, J., and Dryja, T. The bitcoin lightning network: Scalable off-chain instant payments.

[65] Ruffing, T., Schneider, J., and Kate, A. Identity-based steganography and its applications to censorship resistance. In *Communications Security, (CCS)* (2013), pp. 1461–1464.

[66] Sasson, E. B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Security and Privacy (SP)* (May 2014), pp. 459–474.

[67] Tschantz, M. C., Afroz, S., Anonymous, and Paxson, V. SoK: Towards Grounding Censorship Circumvention in Empiricism. In *Security and Privacy (SP)* (May 2016), pp. 914–933.

[68] van Saberhagen, N. Cryptonote v 2.0.

[69] van Saberhagen, N., Meier, J., and Juarez, A. M. CryptoNote Signatures.

[70] van Saberhagen, N., null, S., Meier, J., and Lem, R. CryptoNote One-Time Keys.

[71] von Ahn, L., and Hopper, N. J. Public-Key Steganography. In *Advances in Cryptology - EUROCRYPT* (2004), pp. 323–341.

[72] Wang, L., Dyer, K. P., Akella, A., Ristenpart, T., and Shrimpton, T. Seeing Through Network-Protocol Obfuscation. In *Computer and Communications Security (CCS)* (2015), pp. 57–69.

[73] Weinberg, Z., Wang, J., Yegneswaran, V., Briesemeister, L., Cheung, S., Wang, F., and Boneh, D. Stegotorus: a camouflage proxy for the tor anonymity system. In *Computer and Communications Security (CCS)* (2012), pp. 109–120.

[74] Wikipedia. Ping (blogging).

[75] Wilde, T. Knock Knock Knockin' on Bridges' Doors — Tor Blog. https://blog.torproject.org/knock-knock-knockin-bridges-doors, 2017.

[76] Wiley, B. Dust: A blocking-resistant internet transport protocol. (Accessed May, 2018).

[77] Winter, P., and Lindskog, S. How the Great Firewall of China is Blocking Tor, 2012.

[78] Wustrow, E., Swanson, C. M., and Halderman, J. A. Tapdance: End-to-middle anticensorship without flow blocking. In *USENIX Security Symposium* (2014), pp. 159–174.

[79] Wustrow, E., Wolchok, S., Goldberg, I., and Halderman, J. Telex :anticensorship in network infratructure. *USENIX Security Symposium* (2011).

[80] Xu, X., Mao, Z. M., and Halderman, J. A. Internet Censorship in China: Where Does the Filtering Occur? In *Passive and Active Measurement* (2011), pp. 133–142.

[81] Zolfaghari, H., and Houmansadr, A. Practical censorship evasion leveraging content delivery networks. In *Computer and Communications Security (CCS)* (2016), pp. 1715–1726.