Mohsen Minaei*, Pedro Moreno-Sanchez*, and Aniket Kate

MoneyMorph: Censorship Resistant Rendezvous using Permissionless Cryptocurrencies

Abstract: Cryptocurrencies play a major role in the global financial ecosystem. Their presence across different geopolitical corridors, including in repressive regimes, has been one of their striking features. In this work, we leverage this feature for bootstrapping Censorship Resistant communication. We conceptualize the notion of stego-bootstrapping scheme and its security in terms of rareness and security against chosen-covertext attacks. We present *MoneyMorph*, a provably secure stego-bootstrapping scheme using cryptocurrencies. *MoneyMorph* 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 censorship-resistant proxy and its public key. Unlike the usual bootstrapping approaches (e.g., emailing) with heuristic security, if any, *MoneyMorph* employs public-key steganography over blockchain transactions to ensure provable cryptographic security. We design rendezvous over Bitcoin, Zcash, Monero, and Ethereum, and analyze their effectiveness in terms of available bandwidth and transaction cost. With its highly cryptographic structure, we show that Zcash provides 1148 byte bandwidth per transaction costing less than 0.01 USD as fee.

DOI 10.2478/popets-2020-0058
Received 2019-11-30; revised 2020-03-15; accepted 2020-03-16.

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 to censor Internet content to their citizens. Censors employ several techniques ranging from IP address filtering to deep-packet inspection in order to block disfavored Internet content [62]. Censored users are thereby prevented from not only accessing information on the Internet but also from expressing their views freely. Given that, several circumvention systems have been proposed over the last decade [47]. Nevertheless, censorship still remains a challenge to be fully resolved.

Nowadays, Bitcoin is observing a worldwide presence. Interestingly, this presence is prevalent in countries with large-scale censorship [3, 4], and although possible in theory, completely censoring Bitcoin may not be in the best interest of most countries [56]. The same holds true for other cryptocurrencies focused on smart contracts as in Ethereum [29] or privacy-preserving coin transfers as in Zcash [60] and Monero [63].

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

In preparation, inspired from the key encapsulation mechanism (KEM), we conceptualize the notion of *stego-bootstrapping (SB) scheme* that uses a two-way handshake between a censored user and an uncensored entity (i.e., decoder) where a decoder can transmit bootstrapping credentials of an entry point for a censorship-circumvention protocol (e.g., Tor Bridge) to the censored user in the presence of the censor. We then formally describe the security properties for an SB scheme in terms of *rareness* and *security against chosen-covertext attacks*. Intuitively, the SB scheme achieves rareness if it does not decode a valid message from a regular transaction (i.e., not carrying steganographic data). Moreover, an SB scheme is secure against chosen-covertext attacks if the adversary, given a covertext, cannot tell whether it encodes a message of his choice.

Our contributions. Firstly, we contribute *MoneyMorph*, our instantiation of the SB scheme. The cornerstone of *MoneyMorph* consists in reusing functionality from cryptocurrencies to make it seamlessly and interchangeably deployable with the major cryptocurrencies available today. In fact, *MoneyMorph* works using Zcash, Bitcoin, Monero, and Ethereum as rendezvous. Although we focus on widely deployed cryptocurren-

***Corresponding Author: Mohsen Minaei:** Purdue University, E-mail: mohsen@purdue.edu
***Corresponding Author: Pedro Moreno-Sanchez:** TU Wien, E-mail: pedro.sanchez@tuwien.ac.at, First and second authors contributed equally and are considered co-first authors
Aniket Kate: Purdue University, E-mail: aniket@purdue.edu

cies, our techniques can be leveraged in many other cryptocurrencies that share design principles with them. Supporting a wide range of cryptocurrencies increases the rendezvous available for the censored users and thus their chances of getting bootstrapping credentials.

Secondly, we carry out a comparative study of the different rendezvous by evaluating their effectiveness in terms of available bandwidth per transaction, monetary costs, and percentage of sibling transactions. In our study, Zcash is the preferable option with 1148 byte bandwidth per transaction costing less than 0.01 USD.

Finally, we have implemented *MoneyMorph* using Bitcoin, Ethereum, and Zcash as rendezvous, demonstrating its practicality and backward compatibility. Our evaluation shows that encoding/decoding operations can be completed in less than 50 milliseconds. Moreover, at given block creation rates, the decoder and censored user can simultaneously monitor several cryptocurrency blockchains looking for encoded data in real-time, even with their commodity equipment.

2 Problem Statement

We refer to the problem of bootstrapping communication into an uncensored area as *stego-bootstrapping*. As shown in Fig. 1, a *censored user* wants to receive the credentials of a censorship-resistance protocol entry point (e.g., Tor bridge). For that, the user encodes a challenge message into a short covertext *cc* (e.g., a blockchain transaction) and sends *cc* to the rendezvous.

A *decoder* in the uncensored area provides the censored users with an entry point credentials. For that, the decoder continuously inspects the chosen rendezvous for covertexts, eventually getting the covertext *cc*, decoding it, and obtaining the *challenge* message from the user. Then, the decoder encodes the credentials in a new *response* covertext *rc* and adds it to the rendezvous. The censored user can then obtain *rc*, decode it, and get the bootstrapping details. What happens after this point is out of the scope of this work. We rely on complementary solutions for the censorship-resistant communication.

The communication between censored user and decoder is hindered by the *censor*, an entity that decides what messages enter or exit the censored area. The censor can also run the protocol impersonating a censored user, learn the identity of the decoder, and easily stop the messages that are directly addressed to it. Therefore, we require a solution that communicates with the decoder without directly addressing messages to it.

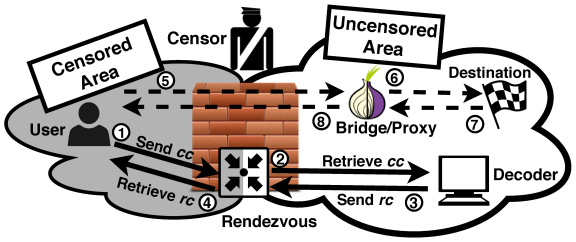

Fig. 1. Censorship circumvention bootstrapping problem. Censored user sends a covertext to the decoder, who replies with another covertext including proxy's details. Then, the censored user can access censored information through the proxy. We focus on the bootstrapping process (solid arrows).

2.1 Stego-Bootstrapping Scheme

The stego-bootstrapping (SB) problem can be seen as a *two-way handshake*, a *challenge* from the censored user to the decoder (*forward* direction), and the corresponding *response* from the decoder to the censored user (*backward* direction). The two-way handshake can be considered as two *independent* "one-way handshakes", each defined in terms of a public-key stegosystem [27], with a single setup, encoding and decoding algorithms. This approach, however, requires the decoder and censored user to know 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. Therefore, inspired by the key encapsulation mechanism, we consider the two-way handshake as a whole and only require that the censored user knows in advance the decoder's public key. Our SB scheme definition contains two pairs of encoding and decoding algorithms, the first for the challenge operations and the second for the response operations.

Here, λ is the security parameter; $\epsilon_1(\lambda)$, $\epsilon_2(\lambda)$ are negligible functions; $\mathcal{M}_c$, $\mathcal{M}_r$ are sets of challenge and response messages; $\mathcal{C}_c$, $\mathcal{C}_r$ are sets of challenge and response authenticated covertexts; and $\mathcal{T}$ is a set of tags. The f and b subscripts stand for forward (challenge operation) and backward (response operations) directions.

Definition 1 (Stego-Bootstrapping (SB) Scheme)**.** *The SB scheme is a tuple of algorithms (SBSet, $SBEnc_f$, $SBDec_f$, $SBEnc_b$, $SBDec_b$) defined as below:*

- $vk_d, sk_d, \tau \leftarrow SBSet(\lambda)$. *On input the security parameter λ, output a key pair vk_d, sk_d and a tag $\tau \in \mathcal{T}$.*
- $\{((cc, \sigma), k), \perp\} \leftarrow SBEnc_f(vk_d, cm, \tau)$. *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 with an authenticated challenge covertext $(cc, \sigma) \in \mathcal{C}_c$ and a symmetric key k; or the symbol $\perp$ to indicate an error.*

- $\{(cm, k'), \perp\} \leftarrow SBDec_f(sk_d, (cc, \sigma), \tau)$. *On input a private key sk_d, an authenticated challenge covertext $(cc, \sigma) \in \mathcal{C}_c$ and a tag $\tau \in \mathcal{T}$, output either a tuple with a challenge message $cm \in \mathcal{M}_c$ and a symmetric key k'; or $\perp$.*
- $\{(rc, \sigma'), \perp\} \leftarrow SBEnc_b(sk_d, k', rm)$. *On input the private key sk_d, a symmetric key k' and a response message $rm \in \mathcal{M}_r$, output either an authenticated response covertext $(rc, \sigma') \in \mathcal{C}_r$; or $\perp$*
- $\{rm, \perp\} \leftarrow SBDec_b(vk_d, k, (rc, \sigma'))$. *On input the public key vk_d, a symmetric key k and an authenticated response covertext $(rc, \sigma') \in \mathcal{C}_r$, output a response message $rm \in \mathcal{M}_r$; or $\perp$.*

Definition 2 (Correctness)**.** *Let vk_d, sk_d, τ be the output of $SBSet(\lambda)$. Let $cm \in \mathcal{M}_c$ and $rm \in \mathcal{C}_r$ be a pair of challenge and response messages. A SB scheme is considered correct if the following conditions hold: (i) Let $((cc, \sigma), k)$ be the output of $SBEnc_f(vk_d, cm, \tau)$. Then, $SBDec_f(sk_d, (cc, \sigma), \tau)$ returns a tuple (cm', k') such that $cm' = cm$ and $k' = k$. (ii) Let (rc, σ') be the output of $SBEnc_b(sk_d, k', rm)$. Then, $SBDec_b(vk_d, k, (rc, \sigma'))$ returns a response message rm' so that $rm' = rm$.*

2.2 Threat Model

We consider the censor as a malicious adversary with network capabilities within the censored area and additionally knows the public key of the decoder. The censor does not control the decoder or its communications.

The censor is able to selectively inspect, fingerprint, block, or inject traffic within the censored area. We assume, however, that the censor is restricted in two ways. First, we assume that there are negative (economic) consequences for the censor to block all communications between censored users and the rendezvous system where both censored user and decoder post their messages. While the censor can always prevent the access to a rendezvous as it has happened before with Telegram or SSL connections [20, 62], we believe that this assumption is realistic in practice using cryptocurrencies as rendezvous system. As the user base for different cryptocurrencies grows, even in countries with heavy censorship, banning them all may have economic consequences for the censor and the censored area [56]. In fact, this assumption is already followed by other works in the community [71]. Second, we assume that the censor cannot alter the information included in the rendezvous (e.g., the censor does control the majority of the Bitcoin network hash rate). We find this assumption realistic as it is required for the security of the rendezvous itself.

2.3 Security Goals

We characterize two security properties for an SB scheme, *rareness* and *security against chosen-covertext attacks*. First, we define the rareness property, a notion closely related to the concept of γ-uniformity [39].

Definition 3 (Rareness)**.** *For negligible functions $\epsilon_1(\lambda)$ and $\epsilon_2(\lambda)$, a SB scheme achieves* rareness *if for all tuples (vk_d, sk_d, τ) output by $SBSet(\lambda)$, the following conditions hold:*
(i) $\Pr[SBDec_f(sk_d, (cc, \sigma), \tau) \neq \perp \mid (cc, \sigma) \leftarrow_\$ \mathcal{C}_c] \leq \epsilon_1(\lambda)$*;*
(ii) Let $((cc, \sigma), k)$ be the output of $SBEnc_f(vk_d, cm, \tau)$ for an arbitrary $cm \in \mathcal{M}_c$. Then, $\Pr[SBDec_b(vk_d, k, (rc, \sigma')) \neq \perp \mid (rc, \sigma') \leftarrow_\$ \mathcal{C}_r] \leq \epsilon_2(\lambda)$.

Next, we characterize security against chosen-covertext attacks (SBS-CCA). We inspire it from SS-CCA by Backes et al. [27] 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 a challenger and an attacker with five rounds as described below.

R1: Key generation stage. The challenger runs $vk_d, sk_d, \tau \leftarrow SBSet(\lambda)$ and gives the public key vk_d and the tag τ to the attacker.

R2: First decoding stage. The attacker has access to an encoding O^{enc} and decoding O_1^{dec} oracles. 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 $((cc, \sigma), k, rc)$ constructed as follows: (i) Run $\{((cc, \sigma), k), \perp\} \leftarrow SBEnc_f(vk_d, cm, \tau)$. (ii) Run $\{(cm, k'), \perp\} \leftarrow SBDec_f(sk_d, (cc, \sigma), \tau)$. (iii) Run $\{(rc, \sigma'), \perp\} \leftarrow SBEnc_b(sk_d, k', rm)$. (iv) If the output in any of the previous steps is $\perp$, return $\perp$, else return $((cc, \sigma), k, (rc, \sigma'))$.

O_1^{dec} has access to the public key vk_d, the private key sk_d and the tag τ. On input a tuple $((cc, \sigma), k, (rc, \sigma'))$ where $(cc, \sigma) \in \mathcal{C}_c$, $(rc, \sigma') \in \mathcal{C}_r$ 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: (i) Run $\{(cm, k'), \perp\} \leftarrow SBDec_f(sk_d, (cc, \sigma), \tau)$. (ii) Run $\{rm, \perp\} \leftarrow SBDec_b(vk_d, k, (rc, \sigma'))$. (iii) If the output in step (i) or (ii) is $\perp$, return $\perp$, else return (cm, rm).

R3: 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 does the following steps depending on it. For $b = 0$, the challenger sets $((cc^*, \sigma^*), (rc^*, \sigma'^*) \leftarrow_\$ \mathcal{C}_c \times \mathcal{C}_r$ and returns $((cc^*, \sigma^*), (rc^*, \sigma'^*))$. For $b = 1$, the

challenger carries out the following steps: (i) Run $\{((cc^*, \sigma^*), k^*), \perp\} \leftarrow SBEnc_f(vk_d, cm^*, \tau)$. (ii) Run $\{(cm^*, k'^*), \perp\} \leftarrow SBDec_f(sk_d, (cc^*, \sigma^*, \tau)$. (iii) Run $\{(rc^*, \sigma'^*, \perp\} \leftarrow SBEnc_b(sk_d, k'^*, rm^*)$. (iv) If any of the previous steps outputs $\perp$, return $\perp$, else return $((cc^*, \sigma^*), (rc^*, \sigma'^*))$.

The challenge phase finishes by sending the tuple $((cc^*, \sigma^*), (rc^*, \sigma'^*))$ 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 authenticated covertexts $((cc^*, \sigma^*), (rc^*, \sigma'^*))$ or they 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 $\{rm, \perp\} \leftarrow SBDec_b(vk_d, k^*, (rc^*, \sigma'^*)$ and checking whether the output is a message $rm = rm^*$. In practice, this models the fact that the intermediate key materials (i.e., symmetric keys k, k') are known only to the honest participants and not the adversary (e.g., the censor).

R4: Second decoding stage. The attacker has access to O^{enc} as described above. Moreover, the attacker has access to a decoding oracle O_2^{dec}, which is similar to O_1^{dec} except that upon receiving the pair $((cc^*, \sigma^*), k'', (rc^*, \sigma'^*))$, where k'' denotes any symmetric key, returns $\perp$.

R5: Guessing stage. The attacker outputs a bit b^*.

Definition 4 (Security against chosen-covertext attacks)**.** *A SB scheme is secure against chosen-covertext attacks if every probabilistic polynomial-time adversary A has negligible advantage in $\mathsf{Exp}_A^{SBS\text{-}CCA}(\lambda)$. We define the adversary's advantage as $|\Pr[\mathsf{Exp}_A^{SBS\text{-}CCA}(\lambda) = b] - 1/2|$*

2.4 System Goals

Sibling Transactions. The SB system must maximize the number of messages that follow the structure of steganographic messages. Otherwise, the censor can deny service (e.g., drop and not route) to the uncommon messages and yet maintain a functional system.

Cost-Efficiency. The SB 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.

Bandwidth. The SB system must provide a bootstrapping solution that maximizes the bandwidth (number of Bytes available) between the censored users and the decoder for transferring the bootstrapping information.

3 Key Ideas

Blockchain as Rendezvous. We leverage the blockchain as rendezvous for censored messages encoded as blockchain transactions. The many blockchain 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 by distributed users worldwide. We observe that the cryptographic structure of blockchain transactions can be used to encode censored data.

Using a blockchain system as rendezvous implies that covertexts remain visible even after the bootstrapping has finished. However, this cannot be leveraged by the censor because *MoneyMorph* is secure against chosen-covertext attacks. Therefore, the adversary cannot tell better than guessing whether a covertext contains bootstrapping data. The censor is thereby left with the choice of banning the complete blockchain system or allowing it completely.

Steganographic Tagging Scheme. We design a cryptographic construction to convert censored messages into ciphertexts that can then be encoded into a covertext transaction. The several public-key steganographic tagging schemes in the literature [26, 27, 35, 42, 66] assume, in general, high bandwidth not available in blockchain transactions. We adapt the construction in [69] aiming to ciphertext succinctness: A ciphertext has 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. Moreover, the group element can be easily included in a blockchain transaction as it already handles public keys.

Fees. *MoneyMorph* introduces a fee overhead, which is inevitable due to the use of cryptocurrencies: A fee is paid to the miners to process and confirm transactions. Increased use of a cryptocurrency implies a fee raise. Fortunately, virtually all cryptocurrencies have built-in mechanisms to handle it.

Paid Services. Currently, many censorship circumvention systems are paid services, including even a premium account, to provide better performance [13, 16, 18, 36]. *MoneyMorph* can be used to bootstrap free of charge services (e.g., Tor) as well as the mentioned paid services. The fee of *MoneyMorph* compared to the actual cost of the mentioned services is negligible. Ultimately, *MoneyMorph* is a bootstrapping mechanism that is employed infrequently by the censored user, resulting in a low amortized cost over a long period of time.

4 Our Protocol

4.1 Building Blocks

Encoding Scheme. The *encoding scheme* allows to encode challenge and response data as a transaction compatible with the rendezvous. We defer our instantiations with different cryptocurrencies to Section 5.

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 5 (Encoding Scheme)**.** *An* encoding scheme *is a tuple of algorithms ($TxEnc_f$, $TxDec_f$, $TxEnc_b$, $TxDec_b$) defined as below:*

- $\{ctx, \perp\} \leftarrow TxEnc_f(cd, ca)$*. On input 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 $\perp$ to indicate an error.*
- $\{cd, \perp\} \leftarrow TxDec_f(ctx)$*. On input a challenge transaction $ctx \in \mathcal{T}_c$, output challenge data $cd \in \mathcal{D}_c$ or the special symbol $\perp$ to indicate an error.*
- $\{rtx, \perp\} \leftarrow TxEnc_b(rd, ra)$*. On input 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 $\perp$ to indicate an error.*
- $\{rd, \perp\} \leftarrow TxDec_b(rtx)$*. On input a response transaction $rtx \in \mathcal{T}_r$, output response data $rd \in \mathcal{D}_r$ or the special symbol $\perp$ to indicate an error.*

Definition 6 (Encoding Scheme Correctness)**.** *An encoding scheme is* correct *if for every 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 $ra \in \mathcal{A}_r$, it holds that: (i) Let $ctx \leftarrow TxEnc_f(cd, ca)$. Then, $cd^* \leftarrow TxDec_f(ctx)$ and $cd^* = cd$. (ii) Let $rtx \leftarrow TxEnc_b(rd, ra)$. Then, $rd^* \leftarrow TxDec_b(rtx)$ and $rd^* = rd$.*

Non-interactive Key Exchange. A non-interactive key exchange (*NIKE*) is a tuple of algorithms ($NIKE.KGen, NIKE.ShKey$), where $(vk, sk) \leftarrow NIKE.KGen(id)$ outputs a public-private key pair vk, sk for a given party identifier id. The algorithm $k \leftarrow NIKE.ShKey(id_1, id_2, sk_1, vk_2)$ outputs a shared key k for the two parties id_1 and id_2. We require a *NIKE* secure in the CKS model. Static Diffie-Hellman key exchange satisfies these requirements [30, 37]. Additionally, we require a function $ID(vk_u)$ that on input a public key vk_u returns the corresponding identifier id_u. We implement this function as the identity function.

MoneyMorph

- $\boldsymbol{vk_d}, \boldsymbol{sk_d}, \tau \leftarrow \boldsymbol{SBSet}(\lambda)$.
  Generate $vk_d, sk_d \leftarrow NIKE.KGen(id_d)$
  Set $\tau \leftarrow \{0,1\}^{64}$. Return vk_d, sk_d, τ.
- $\{((\boldsymbol{cc}, \sigma), \boldsymbol{k}), \perp\} \leftarrow \boldsymbol{SBEnc}_f(\boldsymbol{vk_d}, \boldsymbol{cm}, \tau)$**.**
  1. Compute $vk_u, sk_u \leftarrow NIKE.KGen(id_u)$
  2. Compute $k_d \leftarrow NIKE.ShKey(ID(vk_u), ID(vk_d), sk_u, vk_d)$
  3. Compute $sk_s||k_c||k_r \leftarrow HKDF(k_d, \lambda + l_c + l_r)$
  4. Compute $vk_p \leftarrow vk_d^{sk_s}$ and set $ct_c := (\tau||cm) \oplus k_c$
  5. Compute $ctx \leftarrow TxEnc_f((vk_u, H(vk_p), ct_c), sk_u)$
  6. If $ctx = \perp$, return $\perp$. Else, parse ctx as (cc, σ) and return $(cc, \sigma), k_d$
- $\{(\boldsymbol{cm}, \boldsymbol{k'}), \perp\} \leftarrow \boldsymbol{SBDec}_f(\boldsymbol{sk_d}, (\boldsymbol{cc}, \sigma), \tau)$**.**
  1. Compute $cd \leftarrow TxDec_f((cc, \sigma))$. If $cd = \perp$, return $\perp$
  2. Parse $vk'_u, H(vk'_p), ct'_c \leftarrow cd$
  3. Compute $k'_d \leftarrow NIKE.ShKey(ID(vk_d), ID(vk'_u), sk_d, vk'_u)$
  4. Compute $sk'_s||k'_c||k'_r \leftarrow HKDF(k'_d, \lambda + l_c + l_r)$
  5. Compute $vk_d \leftarrow g^{sk_d}$; $vk''_p \leftarrow vk_d^{sk'_s}$ and set $m' := ct'_c \oplus k'_c$
  6. Parse $\tau'||cm' \leftarrow m'$. Set $b := (\tau' = \tau) \wedge (H(vk'_p) = H(vk''_p))$
  7. If $b = 0$, return $\perp$. Else, return the tuple cm', k'_d
- $\{(\boldsymbol{rc}, \sigma'), \perp\} \leftarrow \boldsymbol{SBEnc}_b(\boldsymbol{sk_d}, \boldsymbol{k'}, \boldsymbol{rm})$**.**
  1. Compute $sk'_s||k'_c||k'_r \leftarrow HKDF(k', \lambda + l_c + l_r)$
  2. Compute $sk''_p \leftarrow sk_d \cdot sk'_s$; $vk''_p := g^{sk''_p}$ and set $ct_r := rm \oplus k'_r$
  3. Compute $rtx \leftarrow TxEnc_b((ct_r, vk''_p), sk''_p)$
  4. Parse rtx as (rc, σ') and return (rc, σ')
- $\{\boldsymbol{rm}, \perp\} \leftarrow \boldsymbol{SBDec}_b(\boldsymbol{vk_d}, \boldsymbol{k}, (\boldsymbol{rc}, \sigma'))$**.**
  1. Compute $rd \leftarrow TxDec_b((rc, \sigma'))$. If $rd = \perp$, return $\perp$.
  2. parse $ct'_r, vk_p \leftarrow rd$
  3. Compute $sk_s||k_c||k_r \leftarrow HKDF(k, \lambda + l_c + l_r)$
  4. Set $rm := ct'_r \oplus k_r$
  5. If $vk_p \neq vk_d^{sk_s}$, return $\perp$. Otherwise, return rm

Fig. 2. The *MoneyMorph* construction. We denote by l_c and l_r the number of bits for the challenge and response message respectively. We denote string concatenation by $||$. Here, H is a cryptographic hash as implemented in the encoding scheme and $\perp$ represents an error generated by the encoding schemes.

Key Derivation Function. A key derivation function $KDF(k, l)$ takes as input a key k and a length value l and outputs a string of l bits. We use the hash-based key derivation function (*HKDF*) in [48] as the secure key derivation function.

Our Construction. In *MoneyMorph* (see Fig. 2), we aim at optimizing the succinctness of the ciphertext. For that, we first use the Diffie-Hellman key exchange to generate a symmetric key (k_d) between the censored user ($SBEnc_f$, steps 1-2) and the decoder ($SBDec_f$, step 3). This symmetric key (k_d) shared between censored user and decoder becomes a master key for a key derivation function to derive three other keys (sk_s, k_c, k_r). Note that this key derivation function does not require interaction between the censored user ($SBEnc_f$, step 3) and decoder ($SBDec_f$, step 4).

The key k_c is used by the censored user to encrypt the challenge message (cm) along with a message tag τ ($SBEnc_f$, step 4). Correspondingly, the decoder uses k_c to decrypt the ciphertext created by the censored user ($SBDec_f$, step 5) and checks whether the decryption contains the tag τ ($SBDec_f$, step 6). The decoder thereby checks whether the ciphertext was the one created by the censored user or it does not contain any censored information otherwise. The key k_r is used similarly by the censored user and the decoder to encrypt ($SBEnc_b$, step 2) and decrypt ($SBDec_b$, step 4) the response message rm. Note that $SBEnc_b$ and $SBDec_b$ must be invoked with the same symmetric key k_d computed in $SBEnc_f$ and $SBDec_f$ to ensure correctness.

The last key (sk_s) becomes a fresh private key shared between the censored user and the decoder. From sk_s, the censored user can create a public key vk_p ($SBEnc_f$, step 4) such that only the intended decoder knows the corresponding private key sk_p (as computed in $SBEnc_b$, step 2). We call the key pair vk_p, sk_p as the *paying key pair*. This key pair is used by the censored user to pay for the service provided by the decoder. The censored user can associate coins to vk_p so that when vk_p becomes a funded address in the blockchain, the decoder, knowing sk_p, can use those coins to cover the cost of sending the response covertext to the censored user.

Note that the decoder can use the coins at vk_p because our construction reconstructs the corresponding sk_p only at the decoder side. This allows the decoder to claim economic rewards without providing the response covertext. However, a rational decoder would arguably respond faithfully to keep the business with the censored users. Further, the decoder is trusted not to be running by the censor, as otherwise, it can trivially link the censored user to the chosen covertext that it can successfully decode. In practice, similar to the Tor directory, trusted decoders can be publicly identified by their public keys. We formalize our construction for *MoneyMorph* in Fig. 2 and defer *all* security proofs to Appendix B.

Theorem 1 (*MoneyMorph* 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-resistant hash function. Let* Π *be a correct encoding scheme. Then, MoneyMorph is a correct stego-bootstrapping scheme as defined in Definition 2.*

4.2 Security Analysis

Theorem 2 (*MoneyMorph* 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 MoneyMorph achieves rareness as defined in Definition 3.*

Theorem 3 (*MoneyMorph* is secure)**.** *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, MoneyMorph is secure against covertext-chosen attacks as defined in Definition 4.*

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 can change vk_d during the *MoneyMorph* lifetime by spending the coins associated to vk_d in a fresh public key vk'_d and destroying sk_d afterwards. Such a transaction notifies every user (including the censor) the change of the decoder's public key. However, as sk_d has been destroyed, covertext challenges created with such a key are secure (achieving eventual forward secrecy).

Immediate Forward Secrecy. In *MoneyMorph*, the decoder uses its pair of keys vk_d, sk_d also for the response covertext. However, it is possible to modify *MoneyMorph* so that the decoder generates a fresh pair of keys vk_r, sk_r instead, 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 *MoneyMorph* 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 use the vk_d to derive a shared key and a non-detectable address. Then, 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 4.1. Given that, censor cannot

distinguish between a transaction encoding a challenge message and any other transaction. Only the decoder, with access to sk_d can do it. Hence, *MoneyMorph* raises the bar to limit the detection capabilities of the censor. This property also allows current widely deployed gateways to offer the decoder service. They might have a well-known cryptocurrency address, and yet the censor cannot easily censor transactions created by *MoneyMorph* and directed to them.

5 Cryptocurrency Encodings

5.1 Encoding Scheme in Bitcoin

Address and Transaction Format. A Bitcoin address is composed of a pair of signing and verification ECDSA keys. A Bitcoin address is then represented by the Base58 encoding for the hash of the verification key. Bitcoins are exchanged between addresses by means of a transaction. In its simplest form, a transaction transfers a certain amount of coins from one (or many) *input* address to one (or many) *output* address.

The Bitcoin protocol uses a scripting system called *Script* [17] that governs how bitcoins can be transferred between addresses within a transaction. In particular, coins are locked in an address according to SPKEY, a Script excerpt that encodes the conditions to unlock the coins. The fulfillment of such conditions are encoded in another Script excerpt called SSIG. A transaction is *valid* if coins unlocked (or spent) in the transaction have not been spent previously; the sum of input coins is greater or equal to the sum of output coins; and for each input SPKEY_i there exists a SSIG_i such that a function $Eval(\text{SPKEY}_i, \text{SSIG}_i)$ returns `true`, where *Eval* evaluates whether SSIG_i contains the correct fulfillment for the conditions encoded in SPKEY_i.

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 SPKEY_i. We describe the different possible formats of the Bitcoin standard locking mechanism in Table 2 (Appendix). In the following, we describe how to re-use each of them to encode data within a transaction.

Pay2PKey: Instead of including an actual verification key within the $\langle$*<pubKey>*$\rangle$ 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 guess a signing key corresponding to the data encoded as verification key.

Pay2PKeyHash: Instead of including the 20 bytes corresponding to the hash of a verification key within the $\langle$*<pubKeyHash>*$\rangle$ 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.

Pay2ScriptHash: Similar to the Pay2PKeyHash, it is possible to encode 20 bytes of data replacing the field $H(script)$. This approach allows the inclusion of arbitrary random data at the cost of losing the locked coins.

Pay2Null: It allows the encoding of up to 80 bytes of data within the field $\langle$*[data]*$\rangle$. Although, this lock mechanism provides the maximum bandwidth so far, it also implies the loss of the locked coins.

Pay2Multisig: 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. 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.

Implementation Details.

We encode the censored data within locking mechanisms of the type Pay2PKeyHash (pkh) or Pay2ScriptHash (psh). Hereby, we use $\text{SPKEY}_{\text{pkh}}(H(vk))$ to denote the sequence $\langle OP\_DUP\ \ OP\_HASH160\ \ H(vk)\ \ OP\_EQUALVERIFY\ \ OP\_CHECKSIG\rangle$ and $\text{SPKEY}_{\text{psh}}(H(vk))$ to denote the sequence $\langle OP\_HASH160\ \ H(vk)\ \ OP\_EQUAL\rangle$. Similarly, we denote by $\text{SSIG}(tx, sk, vk)$ the condition defined as $\langle ECDSA.Sign(tx, sk)\ vk\rangle$. Finally, we denote by *Extract* an extraction function such that $Extract_{\{\text{pkh,psh}\}}(\text{SPKEY}(x)) = x$ and $Extract(\text{SSIG}(tx, sk, vk)) = vk$.

$\{\boldsymbol{ctx}, \perp\} \leftarrow \boldsymbol{TxEnc}_f(\boldsymbol{cd}, \boldsymbol{ca})$**.** Parse $vk_u, H(vk_p), ct \leftarrow cd$ and $sk_u \leftarrow ca$. If $|ct| > 20$ bytes, return $\perp$. Otherwise, create a Bitcoin transaction tx_1 as described below.
We assume that ct has been padded with pseudorandom bytes so that $|ct| = 20$ bytes and vk_u has been funded earlier with X BTC. This amount of coins encode the coins to be burnt at the output $\text{SPKEY}_{\text{psh}}$ as well as the amount of coins required by the decoder to pay for the transaction fee of the response covertext.

tx_1	
Inputs	$\text{SPKEY}'_{\text{pkh}}(H(vk_u))$, X BTC
Outputs	$\text{SPKEY}_{\text{psh}}(ct)$, γ_1BTC
	$\text{SPKEY}_{\text{pkh}}(H(vk_p))$, $(\text{X} - \gamma_1)$BTC
Signature	$\text{SSIG}(tx_1, sk_u, vk_u)$

$\{cd, \perp\} \leftarrow \textbf{\textit{TxDec}}_f(\textbf{\textit{ctx}})$. If ctx does not have one input and two outputs or the lock mechanisms are not of the type Pay2PKeyHash and Pay2ScriptHash, return $\perp$. Otherwise, compute ct as $ct \leftarrow \textit{Extract}(\text{SPKEY}_{\text{psh}}(ct))$. Compute $vk_u \leftarrow \textit{Extract}(\text{SSIG}(tx, sk_u, vk_u))$. Compute $H(vk_p) \leftarrow \textit{Extract}(\text{SPKEY}_{\text{pkh}}(H(vk_p)))$. Return the tuple $cd := (vk_u, H(vk_p), ct)$.

$\{\textbf{\textit{rtx}}, \perp\} \leftarrow \textbf{\textit{TxEnc}}_b(\textbf{\textit{rd}}, \textbf{\textit{ra}})$. 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	$\text{SPKEY}'_{\text{pkh}}(H(vk_p))$, $(\text{X} - \gamma_1)$ BTC
Outputs	$\text{SPKEY}_{\text{psh}}(ct[0:19])$, γ_2BTC
	$\text{SPKEY}_{\text{pkh}}(ct[20:39])$, $(\text{X} - \gamma_1 - \gamma_2)$BTC
Signatures	$\text{SSIG}(tx_2, sk_p, vk_p)$

$\{\textbf{\textit{rd}}, \perp\} \leftarrow \textbf{\textit{TxDec}}_b(\textbf{\textit{rtx}})$. If rtx does not have one input and two outputs, return $\perp$. If the lock mechanisms are not of the type Pay2PKeyHash and Pay2ScriptHash, return $\perp$. Otherwise, compute $ct[0:19] \leftarrow \textit{Extract}(\text{SPKEY}_{\text{psh}}(ct[0:19]))$ and $ct[20:39] \leftarrow \textit{Extract}(\text{SPKEY}_{\text{pkh}}(ct[20:39]))$. Compute $vk_p \leftarrow \textit{Extract}(\text{SSIG}(tx, sk_p, vk_p))$. Return $rd := (ct, vk_p)$.

System Discussion.

Sibling Transactions. We have downloaded a snapshot of the Bitcoin blockchain containing blocks 580,000 to 600,000 (June-Oct 2019), containing around 45 million Bitcoin transactions. In this dataset, we observe that the majority of the transactions contain one input and two outputs. Furthermore, we examined the outputs' locking mechanisms and observed that the combination of one Pay2PKeyHash and one Pay2ScriptHash construct 32% of all transactions followed by two Pay2PKeyHash with 23%. To maximize the sibling transactions, we have selected the combination of Pay2PKeyHash and Pay2ScriptHash as the lock mechanism to be used by *MoneyMorph* to raise the bar for the censor targeting a single transaction type. Nevertheless, if the aforementioned transaction trend changes, any of the locking mechanisms mentioned earlier can be used to lock the coins while encoding censored data [61], which provide the same bandwidth as Pay2PKeyHash or more. Therefore, *MoneyMorph* users could *dynamically* adjust the encoding scheme based on the transaction output distribution in Bitcoin at different times.

Cost. *MoneyMorph* (BTC) requires to pay two transaction fees (tx_1 and tx_2) as well as burn coins three times because the SPKEY outputs used to encode the ciphertexts are no longer spendable. At the time of writing, the fastest and cheapest fee for a transaction is about 4,500 Satoshi (0.34 USD) [5]. If the censored user is willing to wait two hours for its transaction to get into the blockchain, the cost is reduced to 2,700 Satoshi (0.2 USD) Furthermore, to find the minimum value for burning coins (γ_1 and γ_2) without trivially being censored, we investigated all previous transactions with one input and two outputs. We observed that in order to blend with at least 25% of the outputs, the burned amount should be at least 2,500 Satoshi (0.25 USD). We further analyzed the UTXO set and observed that about 50% of the outputs remain unspent for more than 9 months; therefore, the *MoneyMorph* transactions could go unnoticed for a long period of time. We note that Bitcoin has a supply of 21 million Bitcoins. With the current amount of burnt coins per transaction, we would require $\sim 8 * 10^{11}$ transactions to terminate the supply, and thus burnt coins have a negligible effect on the economics of Bitcoin.

Bandwidth. *MoneyMorph* (BTC) uses SPKEY fields to encode the data. Each SPKEY field is leveraged to encode exactly 20 bytes (using Pay2PKeyHash or Pay2ScriptHash). This provides a 20 byte bandwidth for the challenge covertext, as one of the two SPKEY fields is used. On the other hand, the response covertext has a 40 byte bandwidth as it uses both of SPKEY fields to encode encrypted data.

Limitations. *MoneyMorph* (BTC) suffers from two limitations: (i) it requires to burn coins; and (ii) forces the censored user to prepare a spendable input with the precise value amount to generate transactions tx_1 and tx_2. These two limitations can be mitigated by using the Pay2Multisig script in the encoding (as explained earlier in this section and similarly used in the community [59]), or adding an extra output as a change address. Both mitigations come at the cost of reducing the number of sibling transactions. Alternatively, *MoneyMorph* can mitigate these limitations by leveraging other cryptocurrencies discussed in this section.

5.2 Encoding Scheme in Zcash

Addresses and Transactions. Ben-Sasson et al. [60] proposed Zerocash, a privacy-preserving cryptocurrency scheme. The core idea behind Zerocash has been implemented in Zcash [43]. As the implementation slightly differs, we focus our description on the cryptocurrency as detailed in the paper [60] and extend the implementation details when it applies.

Zerocash [43, 60] supports two types of addresses: *transparent* and *shielded*. A transparent address is defined by an amount x of coins that are locked according to a script excerpt SPKey that encodes the conditions to unlock the coins. The fulfillment of such conditions are encoded in another Script excerpt called SSig. More details are included in Section 5.1.

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 $PRF_{sk}(0)$ 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 is a commitment that represents the coin. We describe the format of com later.

Zcash transactions transfer coins from input(s) to output(s) which can be both transparent and shielded addresses. Fig. 3 shows an example of such transaction. Omit for a moment the transparent address in the input. Then, the rest of the transaction is an example of a user that wants to split the coin (i.e., c^{old}, sn^{old}) into two new coins c_1^{new} and c_2^{new}. For that, she creates a single shielded output $(rt, sn^{old}, com_1^{new}, ct_1, com_2^{new}, ct_2, h, vk^*, \sigma^*, \Pi_{pour})$ as follows: rt denotes the root of a Merkle tree whose leafs contains all the commitments $\{com_i\}$ included so far in the blockchain; $sn^{old} := PRF_{sk^{old}}(\rho)$ 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. The payee can then 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, $h := PRF_{sk^{old}}(H(vk^*))$, 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 readers to [60] for a detailed explanation.

Input	$\mathbf{SPKey_0}$, $\mathbf{x_0}$ ZEC
Output	$(rt, sn^{old}, com_1^{new}, ct_1, com_2^{new}, ct_2, h, vk^*, \sigma^*, \Pi_{pour})$
Sign.	$\mathbf{SSig_1}$

Fig. 3. Example of transaction in Zcash.

A transaction can have several transparent inputs and any combination of shielded and transparent outputs. A transaction is valid if, for every shielded output, σ^* is a valid signature under verification key vk^* and Π_{pour} correctly verifies. Furthermore, for every transparent output, the coins unlocked (or spent) have not been spent previously; the sum of input coins is greater or equal to the sum of output coins; and for each input SPKey_i there exists a SSig_i such that a function $Eval(\text{SPKey}_i, \text{SSig}_i)$ returns `true`, where $Eval$ evaluates whether SSig_i contains the correct fulfillment for the conditions encoded in SPKey_i.

Possibilities for Encoding Data. In a shielded output, there are several fields 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 in encoding our data as the different ciphertexts available in the shielded outputs for coins sent to the user herself. We note that this ciphertext field 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.

Furthermore, some data can be encoded in the conditions SPKey of the transaction. In this work we consider the Pay2PKeyHash condition format that includes the 20 bytes corresponding to the hash of a verification key. The SPKey $(H\ (vk))$ along with the SSig of the transaction allows us to encode a verification key. For more details we refer to Section 5.1.

Implementation Details. Here, $Extract$ is an extraction function working as follows: $Extract(\text{SPKey}(x)) = x$ as well as $Extract(\text{SSig}(tx, sk, vk)) = vk$. Additionally, on input a field f and a transaction tx, it returns the value of f if present in the shielded output of tx.

$\{\boldsymbol{ctx}, \perp\} \leftarrow \boldsymbol{TxEnc_f(cd, ca)}$**.** Parse $vk_u, H(vk_p), ct \leftarrow cd$ and $sk_u \leftarrow ca$. If $|ct| > 1148$ bytes, return $\perp$. Otherwise, create a Zcash transaction tx_1 as shown below 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}.

	tx_1
Input	$\mathsf{SPKey}_1(H(vk_u))$, x ZEC
Output	$(rt, sn^{old}, com_1^{new}, H(vk_p)\|\|ct[0,563], com_2^{new}, ct[564,1147], h, vk', \sigma, \Pi_{pour})$
Sign.	$\mathsf{SSig}(tx, sk_u, vk_u)$

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 well formed 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 the public key vk_p and $ct[0, 563]$ (i.e., first 564 bytes). 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 $ct_2^{new} := ct[564, 1147]$. If $|ct| < 1148$ bytes, ct is padded with pseudorandom bytes. Finally, a fresh key pair vk', sk' is used for the signature σ and the hash h of the shielded output.

$\{cd, \perp\} \leftarrow \boldsymbol{TxDec}_f(ctx)$. If ctx does not have a transparent input and a shielded output, return $\perp$. Otherwise, Compute $H(vk_p)||ct[0 : 563] \leftarrow Extract(ctx, ct_1)$, $ct[564 : 1147] \leftarrow Extract(ctx, ct_2)$, and $vk_u \leftarrow Extract(\mathrm{SSIG}(tx, sk_u, vk_u))$. Return $cd := (vk_u, H(vk_p), ct)$.

$\{rtx, \perp\} \leftarrow \boldsymbol{TxEnc}_b(rd, ra)$. Parse $ct, vk_p \leftarrow rd$ and $sk_p \leftarrow ra$. If $|ct| > 1168$ bytes, return $\perp$. Otherwise, create tx_2 as shown below and return tx_2.

	tx_2
Input	$\mathsf{SPKey}(H(vk_p))$, x* ZEC
Output	$(rt, sn_1^{old}, com_3^{new}, ct[0,583], com_4^{new}, ct[584,1167], h', vk'', \sigma', \Pi'_{pour})$
Sign.	$\mathsf{SSig}(tx_2, sk_p, vk_p)$

This transaction spends the coins on vk_p previously funded by the censored user in tx_1. The rest is constructed as in tx_1, with the difference that we don't include the paying public key and thus gaining an extra 20 bytes that we can use for the encoding of the response.

$\{rd, \perp\} \leftarrow \boldsymbol{TxDec}_b(rtx)$. If rtx does not have a transparent input and a shielded output, return $\perp$. Otherwise, compute $ct[0 : 583] \leftarrow Extract(rtx, ct_3)$ and $ct[584 : 1167] \leftarrow Extract(rtx, ct_4)$. Extract $vk_p \leftarrow Extract(\mathrm{SSIG}(rtx, sk_p, vk_p))$. Return $rd := (ct, vk_p)$.

System Discussion.

Sibling Transactions. We have obtained blocks 0 to 480,000 from the Zcash blockchain, containing about 4.2 million transactions. First, we observe that around 11% of the outputs are shielded. Although small, we note that shielded addresses have started to be used, and in Oct 2019, 19% of the transactions contained shielded outputs [25]. Second, we observe that there exist two new coins (and thus two pairs of (com, ct)) for the shielded outputs. Third, we observe, that around 60% of shielded Zcash transactions included one transparent input. Therefore, to maximize the blending with other transactions, we use transactions that are structurally identical to the most widely used shielded transactions in Zcash blockchain. We further looked into the transactions from Aug 2019, and they showed the same trend of observations from earlier months.

Cost & Bandwidth. *MoneyMorph* (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 (0.003 USD). *MoneyMorph* (ZEC) uses the ciphertext ct_i of a Zcash transaction, encoding 1148 bytes for the challenge and 1168 bytes for the response covertexts.

5.3 Encoding Scheme in Monero

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 vk and an extra random point R, from the payee's address using the Monero Stealth Address mechanism [65]. 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 note that while Stealth Addresses hide the intended receiver, they do not define a mechanism to encode censorable data. We refer the readers to [65] for further details.

A Monero transaction is divided into *inputs* and *outputs*. An input consists of a tuple $(\{vk_i\}, \{Com(x_i, r_i)\}, \{\Pi_i\})$, where $\{vk_i\}$ is a *ring* of one-time keys that have previously appeared in the blockchain, each $Com(x_i, r_i)$ is a cryptographic commitment of the amount of coins x_i locked in the corresponding public key vk_i, and each Π_i is 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 $((vk', R'), Com(\mathrm{x}', r'), \Pi')$, where each element is defined as aforementioned. Finally, a transaction contains a signature σ_{ring}, created following the *linkable ring signature scheme* [64]. A Monero transaction is valid if the following conditions hold. First, σ_{ring} shows that the sender knows the signing key sk^* associated to a verification key within the set $\{vk_i\}$ and such 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.

Possibilities for Encoding Data. As the information 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 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. 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.

Further, we can encode extra data within the signature (see Appendix C). In particular, we observe that the *LRS.Sign* algorithm samples $n-1$ random values from $\mathbb{Z}_p$. Our insight is that instead of sampling random numbers, we use the corresponding bytes from a ciphertexts as random numbers. As values $s_1, \ldots, s_{n-1}$ are included in the signature of the transaction, they allow to increase the bandwidth. Currently, Monero establishes that the rings must contain 11 public keys and thus 10 random numbers can encode data.

Implementation Details. Here we detail the encoding of covertexts into Monero transactions.

$\{\boldsymbol{ctx}, \perp\} \leftarrow \boldsymbol{TxEnc}_f(\boldsymbol{cd}, \boldsymbol{ca})$. Parse $vk_u, H(vk_p), ct \leftarrow cd$ and parse $sk_u \leftarrow ca$. Create a transaction tx_1 as shown below. The ciphertext ct is split in chunks of 32 bytes and each chunk is included as a value s_i in the signature.

tx_1	
Inputs	$(\{\mathit{vk}_1^i\}, \{\mathit{Com}(\mathrm{x}_1^i, r_1^i)\}, \{\Pi_1^i\})$, $(\{\mathit{vk}_2^i\}, \{\mathit{Com}(\mathrm{x}_2^i, r_2^i)\}, \{\Pi_2^i\})$
Outputs	$((\mathit{vk}_u, R_1'), \mathit{Com}(\mathrm{x}_1', r_1'), \Pi_1')$, $((\mathit{vk}_p, R_2'), \mathit{Com}(\mathrm{x}_2', r_2'), \Pi_2')$
Sign.	$\sigma_{ring} := (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I})$, $\sigma'_{ring} := (s'_0, s'_1, \ldots, s'_{n-1}, h'_0, \mathcal{I}')$

$\{\boldsymbol{rtx}, \perp\} \leftarrow \boldsymbol{TxEnc}_b(\boldsymbol{rd}, \boldsymbol{ra})$. Parse $ct, vk_p \leftarrow rd$ and parse $sk_p \leftarrow ra$. Create a Monero transaction tx_2 as described below.

tx_2	
Inputs	$(\{\mathit{vk}_1^i\} \cup \mathit{vk}_p, \{\mathit{Com}(\mathrm{x}_1^i, r_1^i)\}, \{\Pi_1^i\})$, $(\{\mathit{vk}_2^i\}, \{\mathit{Com}(\mathrm{x}_2^i, r_2^i)\}, \{\Pi_2^i\})$
Outputs	$((\mathit{vk}_1', R_1'), \mathit{Com}(\mathrm{x}_1', r_1'), \Pi_1')$, $((\mathit{vk}_2', R_2'), \mathit{Com}(\mathrm{x}_2', r_2'), \Pi_2')$
Sign.	$\sigma_{ring} := (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I})$, $\sigma'_{ring} := (s'_0, s'_1, \ldots, s'_{n-1}, h'_0, \mathcal{I}')$

$\{\boldsymbol{cd}, \perp\} \leftarrow \boldsymbol{TxDec}_f(\boldsymbol{ctx})$. Extract the ciphertext ct by concatenating the values in the signature, and the pair vk_u, vk_p from each of the outputs. Return the tuple $cd := (vk_u, H(vk_p), ct)$.

$\{\boldsymbol{rd}, \perp\} \leftarrow \boldsymbol{TxDec}_b(\boldsymbol{rtx})$. Extract the ciphertext ct from the values included in the signature and extract vk_p from the input ring. Return $rd := (vk_p, ct)$.

System Discussion.

Sibling Transactions. We downloaded a snapshot of the Monero blockchain that contains blocks 1,890,000 to 1,940,000 (July-Oct 2019), with more than 200 thousand transactions. From this dataset, we have extracted the distribution of the number of inputs and outputs used by the transactions. We observe that transactions with one and two inputs, each consist of around 48% and 42% of the transactions. Furthermore, around 89% of them have two outputs. Therefore, to maximize our sibling transactions and bandwidth goals, we opt for transactions with two inputs and two outputs.

Cost & Bandwidth. MoneyMorph (XMR) requires to pay only two transaction fees. The price of the transaction fee is 0.0005 XMR (0.03 USD). Each of the outputs provides us with ten fields of 32 bytes, totaling the bandwidth of 640 bytes.

Traditional Privacy-Preserving Currencies vs. *MoneyMorph*. Privacy-preserving currencies already support encrypted messaging; however, the straightforward use of private payments is disadvantageous with respect to *MoneyMorph*. Our approach for Zcash enables more than twice the bandwidth compared to the traditional use case. (i.e., the user pays to herself and memo must not include payment information for the receiver). The 32 bytes of the payment-id in Monero are not enough to send a key (i.e., already 32 bytes) and some extra query data (e.g., proxy type). Instead, *MoneyMorph* provides ten times the bandwidth provided by the straightforward use of Monero.

5.4 Encoding Scheme in Ethereum

In Ethereum, there exist two types of addresses: *external addresses* and *contracts*. An external address is formatted as in Bitcoin and holds a certain amount of ETH, the Ethereum native coin. They, however, differ in that unlike Bitcoin and Zcash, addresses in Ethereum can be used multiple times. A contract has associated a piece of software that implements a certain business logic. A contract invocation requires *data* as input for the contract execution. In this work, we focus on the use of contracts to encode steganographic data.

We have downloaded a snapshot of the Ethereum blockchain and extracted the transactions invoking contracts. We observed that among them, the contract *Etherdelta_2* (an exchange contract) [12] is the most invoked (8% of all transactions) and thus we use it in our encoding mechanism. Nevertheless, the approach described here can be easily extended if any other contract is invoked. The *testTrade* is a method in this contract that checks whether a trade between two different addresses can take place. We will use this sample method to encode our data.
The signature of this method is as follow:

```
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 address is 160 bits and the uint values are 256 bits. To minimize the suspiciousness of the censor we only encode data in the *amountGet*, *amountGive*, *expires*, *amount* fields and use the lower 32 bits (to simulate realistic amounts). We get a total bandwidth of 20 bytes which is enough to bootstrap the censored user.

We denote $H(vk)$ by ADDRESS(vk). We denote by $Extract(tx, tag)$ a function that returns the value of the field tag, e.g. $Extract(tx, Receiver) = H(vk_2)$.

$\{\boldsymbol{ctx}, \perp\} \leftarrow \boldsymbol{TxEnc}_f(\boldsymbol{cd}, \boldsymbol{ca})$**.** Parse $vk_u, H(vk_p), ct \leftarrow cd$ and $sk_u \leftarrow ca$. If $|ct| > 20$ 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 0.001 ETH (0.15 USD), which is equivalent to one transaction fee (γ') for calling the contract. The transaction fee (γ) for transferring Ether to external accounts is about 0.0002 ETH (0.03 USD).

tx_0 (Fund one-time key)

Field	Receiver	Amount	Fee	Signature	Data
Value	$H(vk_p)$	X ETH	γ ETH	$\sigma(sk_u), vk_u$	—

Next, create an Ethereum transaction tx_1. Split the ciphertext ct in chunks of 5 bytes. Each chunk is included as a value v_i in the low bit order of the *amountGet*, *amountGive*, *expires*, *amount* fields of the *data* field of a contract call. Rest of the fields are not changed and will contain proper addresses and values as aforementioned. Moreover, we assume that ct has been padded with pseudorandom bytes so that $|ct| = 20$ bytes. $H(vk_e)$ denotes the address of the *Etherdelta_2* contract. The value is set to zero and the only cost will be the transaction fee. Return $tx_0||tx_1$.

tx_1

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

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

$\{\boldsymbol{rtx}, \perp\} \leftarrow \boldsymbol{TxEnc}_b(\boldsymbol{rd}, \boldsymbol{ra})$**.** Parse $ct, vk_p \leftarrow rd$ and $sk_p \leftarrow ra$. If $|ct| > 20$ 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 $|ct| = 20$ bytes.

tx_2

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

$\{\boldsymbol{rd}, \perp\} \leftarrow \boldsymbol{TxDec}_b(\boldsymbol{rtx})$**.** If rtx does not have $H(vk_e)$ 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 $data \leftarrow Extract(tx_2, data)$. Compute $\sigma(sk_p), vk_p \leftarrow Extract(tx_2, signature)$. Return the tuple $rd := (vk_p, ct)$.

5.5 Summary of Our Findings

We compare the feasibility of different cryptocurrencies as rendezvous in Table 1. We observe that shielded Zcash transactions provide the most bandwidth with 1168 bytes at a low cost of less than 0.01 USD. The downside is that only 11% of the transactions within Zcash are shielded, however, in the past month (Oct 2019) this number has increased to 19% [25]. Moreover, Zcash currently has the lowest market capitalization and exerts the lowest economic impact for a censor if it decides to ban it when compared to the other cryptocurrencies in this work.

Bitcoin (presented in Section 5.1) 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 32% of the Bitcoin transactions, therefore hindering the censor's task. However, the fees in Bitcoin are the largest among all, 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. These transparent transactions have the lowest fees among all of the cryptocurrencies, with only 0.003 USD.

After shielded Zcash, Monero (Section 5.3) provides the most bandwidth with 640 bytes. Interestingly, the fee associated with the Monero transactions are the second lowest among the four cryptocurrencies. The type of transactions we consider in Monero blends in with 42% of all Monero transactions, making it difficult for the censor to block all such transactions. Ethereum provides the least amount of bandwidth (only 20 bytes in each direction according to the encoding scheme in Section 5.4) and a moderate cost of 0.18 USD compared to the other cryptocurrencies.

		Bitcoin	Z(Tr)	Z(Sh)	Monero	Ethereum
Chal	Bandwidth	20	20	1148	640	20
	Tx fee	$0.34	$0.003	$0.003	$0.03	$0.18
	Lost coins	$0.18	$0.01	—	—	—
Resp	Bandwidth	40	40	1168	640	20
	Tx fee	$0.34	$0.003	$0.003	$0.03	$0.15
	Lost coins	$0.36	$0.02	—	—	—
	Sibling txs	32%	81%	19%	42%	> 8%
	Market cap $	136B	230M	230M	950M	16B

Table 1. Comparison of the different rendezvous. Here, we consider the coins market value [7] at the time of writing (Nov. 27th 2019). We denote Zcash *transparent* by Z(Tr) and *shielded* by Z(Sh). Similar to Zcash(Tr), results for Bitcoin can be applied to Altcoins following the Bitcoin transaction patterns.

6 Implementation and Evaluation

We have developed a prototypical python implementation [15] to show the feasibility and practicality of *MoneyMorph*. We divide the implementation into two separate tasks: i) cryptographic operations and ii) cryptocurrency specific transaction encoding. In the remaining of this section, we detail each of the tasks.

6.1 Cryptographic Operations

As shown in Fig. 2, the SB scheme requires two main cryptographic operations. First is the symmetric key derivation required by $SBEnc_f$ and $SBDec_f$, where each party (a user or the decoder) uses its private key along with the public key of the other party to perform the Diffie-Hellman key exchange. Next, using this shared key, it obtains symmetric keys for the encryption and decryption of the challenge and response messages. For the implementation, we leveraged the "ecdsa" [11] and "cryptography" [8] libraries to perform the Diffie-Hellman key exchange and derive the mentioned keys. The process of creating a fresh pair of keys takes the censored user about 120 milliseconds and deriving the shared key and symmetric keys for the encryption about 41 milliseconds on average.

Second, we have the encryption and decryption of the challenge and response messages. The timing of the encryption and decryption is dependent on the length of the messages. In our experiment, both encryption and decryption take less than 1 milliseconds. The challenge message sent by the user was chosen as *tor—obfs3* with the tag *00000000*. The hexadecimal representation of the message, key, and cipher is presented below.

```
Message: 3030303030303030746f722d2d2d2d6f62667333
Enc Key: 399b8178571ea29a8094562e2200f6ad1b024f8f
Cipher:  09abb148672e92aaf4fb24030f2ddbc279643cbc
```

For the operations mentioned above, we used a personal commodity machine with an Intel Core i7, 2.2 GHz processor, and 16 GB RAM. Note that the blocks in all cryptocurrencies are generated with some specified creation rate, resulting in a time gap between the sequential blocks. We observe that a commodity machine, such as the one that we have used, will have more than enough capabilities to serve the users in multiple cryptocurrencies. Using higher performance machines will only improve the timings of the mentioned operations; However, the end result remains the same as the results will not be immediate for the users due to the time gap

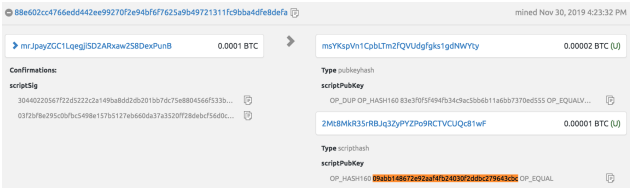

Fig. 4. Snapshot of the challenge transaction on Bitcoin testnet. The highlighted hexadecimals are showing the Cipher mentioned above inside the scriptpubkey (SPKEY).

between the blocks. We further investigated the possibility of hosting the decoder on cloud platforms and observed that the price of a simple machine with 1TB of storage for multiple blockchains would be less than 100 USD per month [2, 9].

6.2 Transaction Encoding

After encrypting the challenge and response messages, the parties need to encode them into one of the cryptocurrencies' transactions. To show the feasibility and practicality of the instantiations mentioned in Section 5, we implemented the encoding scheme for three of the cryptocurrencies and deployed the challenge and response transactions to the corresponding test network.

Bitcoin. The implementation details of the encoding scheme in Bitcoin are provided in Section 5.1. In summary, using the "bitcoin-utils" library [6], we constructed and deployed two transactions on to the Bitcoin's blockchain. The first transaction, tx_1 (https://tinyurl.com/wvmlyyz) sent by the user, includes two outputs of the type Pay2PKeyHash and Pay2ScriptHash. The first output is the paying address that the decoder uses to send the response to the user. The second output is the 20 byte ciphertext mentioned in Section 6.1 (`09abb148672e92aaf4fb24030f2ddbc279643cbc`) and shown in Fig. 4. Similarly, the decoder encodes the response message into tx_2 using the first output of tx_1 (https://tinyurl.com/qsh5a7y). We further analyzed the time it takes the decoder to process the transactions in a block. We observed that it takes less than 100 milliseconds to retrieve a block that only contains the hashes of its transaction. It takes an additional 2-5 millisecond for each of the transactions to be fetched and decoded. Overall, a block of Bitcoin containing about $2,000$ transactions can be examined by the decoder to find challenge messages in less than 15 seconds.

Ethereum. The implementation details of the encoding scheme in Ethereum are provided in Section 5.4. In summary, using the "web3" library, we encoded the challenge and response messages into transactions that were sent to the *etherdelta2* contract. However, since we were working on the testnet (the Rinkeby network), first we needed to deploy the contract (presented in https://tinyurl.com/s7w9sxe). Next, the censored user funds the paying address (vk_p) using tx_0 (https://tinyurl.com/rqy4ns9) and sends the challenge covertext via tx_1 (https://tinyurl.com/vq6b3qn). Similar to tx_1, the decoder used the funds in vk_p (obtained from tx_0) and sends the response covertext through tx_2 (https://tinyurl.com/uol385r). Keep in mind that the challenge and response ciphertexts are fragmented into 4 pieces and placed as input values to the *testTrade* function. Similar to Bitcoin, we analyzed the time it takes the decoder to process the transactions in an Ethereum block. Overall, a block of Ethereum containing on average 100 transactions can be examined by the decoder to find challenge messages in less than 1 second.

Zcash. The implementation details of the encoding scheme in Zcash are provided in Section 5.2. We observed that, the most common transaction has one transparent input and two shielded outputs. This allows parties to send funds to themselves without loosing any coins (as the values and addresses of the shielded outputs are hidden) other than the transaction fee[1]. For the implementation we used the "zcash-cli" (command line client) to construct, send and receive transactions. The challenge and response data were included in the memo field of tx_1 (https://tinyurl.com/wtku9ge) and tx_2 (https://tinyurl.com/vnleyhb). We note that our tx_1 includes an extra transparent output for retrieving the remainder of the coins for testing purposes and similar to tx_2 (that has no transparent outputs) it can easily be omitted. Zcash, compared to the other two cryptocurrencies, has a small number of transactions per block. With an average of 5 transactions per block, it takes a decoder less than a second to process a block and look for challenge messages.

Transaction Finality. Previously, we saw that the time it takes to process a block to decode the challenge and response covertexts in all the three cryptocurrencies is insignificant. However, the procedure for sending the challenge and response covertexts is bounded by the

1 This feature allows the users to send the shielded output directly to the shielded address of the decoder known by all.

transaction finality. We observe that transaction finality introduces a latency proportional to the rate of cryptocurrency blocks creation. Among the three cryptocurrencies, Bitcoin has the slowest block creation time at 10 minutes on average. Next is Zcash with an average block creation rate of 2.5 minutes and lastly Ethereum with a fast creation rate of 15 seconds. These results show that the throughput for any instantiation of *MoneyMorph* is only limited by the blocks generation rate. Fortunately, bootstrapping of censorship-resistance credentials does not have to be real-time; instead, it is carried out infrequently by each user (e.g., emails are currently used to find Tor bridges).

Simple Payment Verification. Although we recommend the censored users to run a full node for the cryptocurrency used as rendezvous if storage and computation overhead is prohibitive, they can run the Simple Payment Verification (SPV) client software to query expected blocks from available full nodes (multiple nodes for redundancy and security). As explained in Section 5, *MoneyMorph* ensures that clients know what stealth address should to query to retrieve the response, thereby reducing the number of required blocks. While improving the usability, this approach comes, however, with a tradeoff in security guarantees. First, SPV clients are more susceptible to eclipse attacks similar to those in [28, 40]. Second, the selective query of transactions (e.g., using Bloom filters) could facilitate the detectability task of the censor, similar to [44]. Recent advances aim to mitigate those detectability issues in Bitcoin [49, 53] and Zcash [68], and similar techniques could be applied for Ethereum and Monero.

7 Related Work

The traditional censorship circumvention systems such as VPNs [55, 57], Dynaweb [10], Ultrasurf [24], Lantern [13], Tor [32], and others [33] 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 a cat-and-mouse game is between Tor [32] and the Great Firewall of China, which has resulted in introducing mirrors [23], bridges [21], and secret entry servers [22] in the Tor system. At the same time, multiple attacks (e.g., active probing and insider attacks) have been proposed to discover the Tor bridges [34, 51, 67]. In recent years domain fronting [14, 36] 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 [18]. To reduce the cost, we can benefit from the use of content delivery networks (CDNs) namely CDNBrowsing [41, 72]. CDN's disadvantage is the unblocking of limited censored contents [72]. Moreover, as a central authority controls these services, their support for censorship circumvention is not reliable [1].

The most recent line of work in censor circumvention is the decoy routing approach [38, 45, 46, 54, 69, 70]. 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. Other anti-censorship mechanisms available in the literature leverage blog pings as communication medium [52] or hinder the harvesting attack by the censor relying on proof-of-work [50].

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

Stealth Addresses (SA) is a cryptographic technique that allows detaching public keys from the intended receiver's identity, thus providing anonymity. However, SA does not define any data encoding mechanism. In this work, we not only ensure the anonymity of the receiver but also investigate how we can use different parts of the most used types of transactions in each of the cryptocurrencies to encode bootstrapping information.

Concurrently to this work, Tithonus [59] contributed a censorship-resistant communication tool that leverages the Bitcoin blockchain and the Bitcoin P2P network to enable communication between the censored and uncensored areas. While Tithonus only considers Bitcoin as the communication channel, we instead define how to leverage other cryptocurrencies for rendezvous. We include a more detailed comparison in Appendix A.

8 Conclusions and Future Work

Despite the many academic and practical alternatives for censorship resistance, censorship remains an important problem that hinders numerous people from freely accessing and communicating information. 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 communication channels offering interesting tradeoffs between bandwidth, costs and censorship resistance. In particular, we describe for the first time communication channels fully compatible not only with Bitcoin but also with Zcash, Monero, and Ethereum that allow censored users to get bootstrapping credentials.

Nevertheless, this work only scratches the tip of the iceberg. The different permissionless blockchains are in continuous development, and new features are being added continuously that may come with yet unexplored possibilities to build a communication channel. For instance, the deployment of off-chain payment channels [31, 58] adds a new locking mechanism to Bitcoin and the alike cryptocurrencies with extra fields that can potentially be used to encode extra covertext bytes of the communication between censored user and decoder. This work sets the grounds for future research works exploring the use of blockchain for censorship resistance communications.

Acknowledgments

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, and the anonymous reviewers for their helpful comments. This work has been partially supported by the National Science Foundation under grant CNS-1846316, and by the FWF Austrian funding agency through the Lise Meitner program.

References

[1] Amazon Web Services starts blocking domain-fronting, following Google's lead. https://www.theverge.com/2018/4/30/17304782/amazon-domain-fronting-google-discontinued.

[2] Amazon workspaces pricing. https://aws.amazon.com/workspaces/pricing/.

[3] Bitcoin in Iran. https://localbitcoins.com/country/IR.

[4] Bitcoin Purchase in China. https://paxful.com/china/buy-bitcoin.

[5] Bitcoin transaction fee estimator. https://www.buybitcoinworldwide.com/fee-calculator/.

[6] bitcoin-utils Python Package. https://pypi.org/project/bitcoin-utils/.

[7] Cryptocurrency market capitalizations. https://coinmarketcap.com/.

[8] Cryptography Python Package. https://pypi.org/project/cryptography/.

[9] Digital Ocean Pricing. https://www.digitalocean.com/pricing.

[10] Dynaweb. http://us.dongtaiwang.com/loc/download.en.php.

[11] ECDSA Python Package. https://github.com/warner/python-ecdsa.

[12] Etherdelta source code. https://etherscan.io/address/0x8d12a197cb00d4747a1fe03395095ce2a5cc6819#code.

[13] Lantern. https://getlantern.org.

[14] Meek transport. https://trac.torproject.org/projects/tor/wiki/doc/meek.

[15] MoneyMorph Python Prototype. https://github.com/moneymorph.

[16] Psiphon. https://www.psiphon3.com/en/index.html.

[17] Script - Bitcoin Wiki. https://en.bitcoin.it/wiki/Script.

[18] Summary of meek's costs, july 2016. https://lists.torproject.org/pipermail/tor-project/2016-August/000690.html.

[19] Talk crypto blog OP_RETURN 40 to 80 bytes. http://www.talkcrypto.org/blog/2016/12/30/op_return-40-to-80-bytes/.

[20] The Unexpected Fallout of Iran's Telegram Ban. https://www.wired.com/story/iran-telegram-ban/.

[21] Tor:bridges. https://www.torproject.org/docs/bridges.

[22] Tor:Hidden Service. https://www.torproject.org/docs/hidden-services.

[23] Tor:mirrors. https://www.torproject.org/getinvolved/mirrors.html.en.

[24] Ultrasurf. https://ultrasurf.us/.

[25] Zcash Usage. https://explorer.zcha.in/statistics/usage.

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

[27] Backes, M., and Cachin, C. Public-Key Steganography with Active Attacks. In *TCC* (2005), pp. 210–226.

[28] Biryukov, A., Khovratovich, D., and Pustogarov, I. Deanonymisation of clients in bitcoin P2P network. In *ACM Conference on Computer and Communications Security* (2014), pp. 15–29.

[29] Buterin, V., and Foundation, E. A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper.

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

[31] Decker, C., and Wattenhofer, R. A fast and scalable payment network with bitcoin duplex micropayment channels. In *(SSS)* (2015), pp. 3–18.

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

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

[34] Ensafi, R., Winter, P., Mueen, A., and Crandall, J. R. Analyzing the great firewall of china over space and time. *PoPETs*, 1 (2015), 61–76.

[35] Fazio, N., Nicolosi, A., and Perera, I. M. Broadcast Steganography. In *Topics in Cryptology - CT-RSA* (2014).

[36] Fifield, D., Lan, C., Hynes, R., Wegmann, P., and Paxson, V. Blocking-resistant communication through domain fronting. *PoPETs*, 2 (2015), 46–64.

[37] Freire, E. S. V., Hofheinz, D., Kiltz, E., and Paterson, K. G. Non-interactive key exchange. In *PKC* (2013).

[38] 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 *FOCI* (2017).

[39] Fujisaki, E., and Okamoto, T. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology 26*, 1 (Jan 2013).

[40] Heilman, E., Kendler, A., Zohar, A., and Goldberg, S. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security Symposium* (2015), pp. 129–144.

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

[42] Hopper, N. On Steganographic Chosen Covertext Security. In *ICALP* (2005).

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

[44] Houmansadr, A., Brubaker, C., and Shmatikov, V. The Parrot Is Dead: Observing Unobservable Network Communications. In *S&P* (2013), pp. 65–79.

[45] Houmansadr, A., Nguyen, G. T. K., Caesar, M., and Borisov, N. Cirripede: Circumvention infrastructure using router redirection with plausible deniability. In *CCS* (2011).

[46] 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 *FOCI* (2011).

[47] Khattak, S., Elahi, T., Simon, L., Swanson, C. M., Murdoch, S. J., and Goldberg, I. Sok: Making sense of censorship resistance systems. *PoPETs 2016*, 4 (2016).

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

[49] Le, D. V., Hurtado, L. T., Ahmad, A., Minaei, M., Lee, B., and Kate, A. A tale of two trees: One writes, and other reads. optimized oblivious accesses to bitcoin and other utxo-based blockchains. *Proceedings on Privacy Enhancing Technologies* (2020), 519–536.

[50] 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).

[51] 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.

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

[53] Matetic, S., Wüst, K., Schneider, M., Kostiainen, K., Karame, G., and Capkun, S. BITE: bitcoin lightweight client privacy using trusted execution. *IACR Cryptology ePrint Archive 2018* (2018), 803.

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

[55] Nobori, D., and Shinjo, Y. Vpn gate: A volunteer-organized public vpn relay system with blocking resistance for bypassing government censorship firewalls. In *NSDI* (2014).

[56] Parker, E. Can china contain bitcoin? MIT Technology Review Blog Post. https://www.technologyreview.com/s/609320/can-china-contain-bitcoin, 2017.

[57] 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. *PoPETs*, 1 (2015), 77–91.

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

[59] Recabarren, R., and Carbunar, B. Tithonus: A bitcoin based censorship resilient system. *Proceedings on Privacy Enhancing Technologies 2019*, 1 (2019), 68–86.

[60] Sasson, E. B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *S&P* (2014).

[61] Sward, A., Vecna, I., and Stonedahl, F. Data insertion in Bitcoin's Blockchain. *Ledger 3* (2018).

[62] Tschantz, M. C., Afroz, S., Anonymous, and Paxson, V. SoK: Towards Grounding Censorship Circumvention in Empiricism. In *(S&P)* (2016), pp. 914–933.

[63] van Saberhagen, N. Cryptonote v 2.0. https://cryptonote.org/whitepaper.pdf, 2013.

[64] van Saberhagen, N., Meier, J., and Juarez, A. M. CryptoNote Signatures. https://cryptonote.org/cns/cns001.txt, 2011.

[65] van Saberhagen, N., null, S., Meier, J., and Lem, R. CryptoNote One-Time Keys. https://cryptonote.org/cns/cns006.txt, 2012.

[66] von Ahn, L., and Hopper, N. J. Public-Key Steganography. In *EUROCRYPT* (2004).

[67] Winter, P., and Lindskog, S. How the Great Firewall of China is Blocking Tor. In *Free and Open Communications on the Internet, FOCI* (2012).

[68] Wust, K., Matetic, S., Schneider, M., Miers, I., Kostiainen, K., and Capkun, S. Zlite: Lightweight clients for shielded zcash transactions using trusted execution. Cryptology ePrint Archive, Report 2018/1024, 2018.

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

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

[71] Zhang, F., Daian, P., Bentov, I., and Juels, A. Paralysis proofs: Safe access-structure updates for cryptocurrencies and more. In *Financial Crypto* (2018).

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

A Comparison with Tithonus

While pursuing the same goal, Tithonus and *MoneyMorph* offer a tradeoff in several aspects. We review them in this section. We remark that *MoneyMorph* can work over different encoding schemes, each of them leveraging a different cryptocurrency. We thus refer here only to the Bitcoin-based encoding unless otherwise specified.

Number of transactions. Tithonus leverages staged transactions, an encoding technique that exploits the input part of the Script language for the encoding. In a nutshell, this technique requires two transactions for a single transmission (e.g., single communication from the censored user to the decoder): the preparing transaction and the redeeming transaction [59, Fig. 1]. The preparing transaction creates a Pay2ScriptHash output. The redeeming transaction spends the Pay2ScriptHash output by providing a $\langle redeem\_script \rangle$ that encodes the censored data. Instead, *MoneyMorph* encodes the censored data directly in an output script, requiring thus a single transaction for a single transmission.

Bandwidth. Although Tithonus and *MoneyMorph* define different encoding schemes, the bandwidth per transaction is determined by the same factor: the actual Bitcoin script used in $\langle redeem\_script \rangle$ for Tithonus or in the encoded output for *MoneyMorph*. While arbitrary scripts can potentially be encoded in the $\langle redeem\_script \rangle$, only standard and widely used scripts can be leveraged in Tithonus to avoid trivial attacks against undetectability, as also mentioned by the authors. Thus, both Tithonus and *MoneyMorph* provide the same bandwidth per transaction: the one allowed by the widely used Bitcoin scripts. In this work, we show that higher bandwidth is possible in a single transaction by leveraging other cryptocurrencies today such as Zcash and Monero. Tithonus also supports a mode for arbitrary length communication that provides higher bandwidth at the cost of using several transactions.

Undetectability. The encoding scheme in Tithonus opens the door to two potential sources of leakage to the censor. First, the period of time between the transmission of the pair spending-redeeming transactions could be different from time elapsed between a traditional payment based on Pay2ScriptHash and the time where the receiver decides to redeem such payment. This potential source of leakage does not exist in *MoneyMorph* as it requires a single transaction. Second, the actual Bitcoin script included in the $\langle redeem\_script \rangle$ by Tithonus should be widely used by non-censored users. This is also the case in *MoneyMorph*. A possible mitigation studied in this work (and also pointed out in Tithonus) consists on constantly monitoring the usage of different Bitcoin script types and dynamically leverage the most widespread script type. Alternatively, in the case of *MoneyMorph*, it is possible to leverage the encoding scheme in other cryptocurrencies. For instance, Monero defines only one type of transactions aiming to achieve fungibility and thus avoids the aforementioned leakage to the censor by definition.

Economic cost. Encoding the censored data directly in an output of the type other than Pay2Multisig as in *MoneyMorph* presents two disadvantages. First, the output becomes unspendable as it is cryptographically hard to find the preimage of a random hash value. Second, the sender of such transaction (e.g., the censored user in the case of *MoneyMorph*) incurs an economic cost as such output must be funded with an amount that does not make the output easy to detect for the censor. These issues do not appear in Tithonus because the preparing transaction is spendable by the same user that funds it and the encoded data is encrypted in a Pay2Multisig script, allowing to encode a valid public key to recover the coins and yet use the other keys to encode the censored data. This issue can be seamlessly be mitigated in *MoneyMorph*. First, *MoneyMorph* can encode the censored data in a Pay2Multisig output as specified in Section 5.1 (and also described in Tithonus). Second, *MoneyMorph* can leverage the encoding schemes in Zcash and Monero that allows the censored user to retrieve the invested coins while still allowing the steganographic transmission of data to the decoder.

Payment obfuscation. One challenge inherent to blockchains as rendezvous for censorship resistance communication is that the censored user needs to transfer funds to an address, allowing thereby the decoder to redeem those funds. However, the naive approach of transferring those funds to an address that is publicly associated to the decoder would pave the way for the censor task. In Tithonus, authors suggest to transfer the funds to an exchange instead of the receiver. Those funds should then be traded to several different cryptocurrencies before being transferred to the actual receiver. Arguably, this puts a non-trivial burden in the usability of the system and adds an economic cost as each of these transactions has an associated fee. In *MoneyMorph*, we solve this issue cryptographically: The censored user transfer funds to a public key whose private key is known only to the decoder. Moreover,

the censored user can non-interactively create many of such public keys and yet the censor cannot link any of these public keys to the decoder. The censored user can thereby transfer the funds directly to the decoder, in a single transaction, totally avoiding to rely on non-colluding third-party exchanges.

B Security Proofs

B.1 Security Proof of Theorem 1

We start by showing that condition 1 holds. In particular, given a pair $((cc,\sigma),k) \leftarrow SBEnc_f(vk_d, cm, \tau)$, we need to show that $SBDec_f(sk_d,(cc,\sigma),\tau)$ returns a pair (cm',k') such that $cm' = cm$ and $k' = k$.

By correctness of Π, $vk'_u = vk_u$, $H(vk'_p) = H(vk_p)$ 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 $SBDec_f$ does not return $\perp$, this proves that both functionalities output the same symmetric key. Now, we show that $SBDec_f$ 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 $SBDec_f$ returns a tuple (cm', k'_d).

The condition 2 holds following similar arguments.

B.2 Security Proof of Theorem 2

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

Let $vk'_u, H(vk'_p), ct'_c$ be the tuple extracted by $TxDec_f((cc,\sigma))$. 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 $SBDec_f$. 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 $\Pr[SBDec_b(vk_d,k,(rc,\sigma')) \neq \perp \mid (rc,\sigma') \leftarrow_\$ \mathcal{C}_r] \leq \epsilon_2(\lambda)$ where k is part of the pair $(cc,k) \leftarrow SBEnc_f(vk_d, cm, \tau)$.

Let $vk_p := g^{sk_s}$ the public key encoded in (cc,σ) after executing $SBEnc_f$. Note that, the same vk_p is generated in $SBDec_b$ 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,σ').

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

B.3 Security Proof of Theorem 3

Assume by contradiction that *MoneyMorph* 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 $|\Pr[\mathsf{Exp}_B^{NIKE}(\lambda) = b] - 1/2| > \epsilon_2(\lambda)$ [37]. We define B as follow:

- On input $(\lambda, params)$:
  - Query challenger with $register(ID(vk_d))$ and get vk_d.
  - Query challenger with $extract(ID(vk_d))$ and get sk_d.
  - Compute $vk'_d, sk'_d, \tau \leftarrow SBSet(\lambda)$. 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 challenger with $register(ID(vk_u))$ and get vk_u.
  - Query challenger with $extract(ID(vk_u))$ and get sk_u.
  - Query challenger with $reveal(ID(vk_u), ID(vk_d))$. Get k_d.
  - Compute $sk_s||k_c||k_r \leftarrow HKDF(k_d, \lambda + l_c + l_r)$
  - Compute $vk_p \leftarrow vk_d^{sk_s}$ and set $ct_c := (\tau||cm) \oplus k_c$
  - Compute $(cc,\sigma) \leftarrow TxEnc_f((vk_u, H(vk_p), ct_c), sk_u)$
  - Compute $sk_p \leftarrow sk_d \cdot sk_s$ and set $ct_r := rm \oplus k_r$
  - Compute $(rc,\sigma) \leftarrow TxEnc_b((ct_r, vk_p), sk_p)$
  - If $cc = \perp$ or $rc = \perp$, return $\perp$. Else, return $((cc,\sigma), k_d, (rc,\sigma'))$

  Due to the correctness of *NIKE* and *HKDF* there is no need to run $SBDec_f$ as the symmetric key generated in this function is equal to the one in $SBEnc_f$. Similarly, the *HKDF* function in $SBEnc_b$ is not necessary to be computed.
- B simulates the oracle O_1^{dec} as follows. On input $((cc,\sigma), k_d, (rc,\sigma'))$:
  - Compute $cd \leftarrow TxDec_f((cc,\sigma))$
  - If $cd = \perp$, return $\perp$. Otherwise:
    - Parse $vk_u, H(vk_p), ct_c \leftarrow cd$
    - Compute $sk_s||k_c||k_r \leftarrow HKDF(k_d, \lambda + l_c + l_r)$

Lock's name	Script	Description of unlocking conditions
Pay2PKey	⟨*<pubKey> OP_CHECKSIG*⟩ ⟨*<sig>*⟩	Including a signature ⟨*<sig>*⟩ of the Bitcoin transaction verifiable using the verification key ⟨*<pubKey>*⟩.
Pay2PKeyHash	⟨$OP\_DUPOP\_HASH160 < pubKeyHash > OP\_EQUALVERIFYOP\_CHECKSIG$⟩ ⟨*<sig> <pubKey>*⟩	Including a verification key ⟨*<pubKey>*⟩ such that ⟨*<pubKeyHash>*⟩ $= H(\langle$*<pubKey>*$\rangle)$ and a signature ⟨*<sig>*⟩ of the Bitcoin transaction verifiable using the verification key ⟨*<pubKey>*⟩
Pay2ScriptHash	⟨*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`.
Pay2Null	⟨*OP_RETURN [data]*⟩ ⟨*<empty>*⟩	Coins can never be unlocked. Data can contain up to 80 bytes [19].
Pay2Multisig	⟨$OP\_M < pubkey1 > \ldots < pubkeyn >$ $OP\_NOP\_CHECKMULTISIG$⟩ ⟨*<sig1> … <sigm>*⟩	Including M signatures ⟨*<sig1> … <sigm>*⟩ of the Bitcoin transaction, verifiable using the corresponding verification keys ⟨*<pubkey1> … <pubkeyn>*⟩

Table 2. Description of the Script excerpts used in the Bitcoin transactions. Text in blue denotes SPKEY and orange denotes the corresponding SSIG.

  - Compute $vk_d \leftarrow g^{sk_d}$ and $vk'_p \leftarrow vk_d^{sk_s}$
  - Set $m := ct_c \oplus k_c$ and parse $(\tau' || cm) \leftarrow m$
  - Set $b := (\tau' = \tau) \wedge (H(vk'_p) = H(vk_p))$
  - If $b = 0$, return $\perp$.
- Compute $rd \leftarrow TxDec_b((rc, \sigma'))$
- If $rd = \perp$, return $\perp$. Otherwise: parse $ct_r, vk_p \leftarrow rd$ and set $rm := ct_r \oplus k_r$
- If $b = 0$ or $vk_p \neq vk_d^{sk_s}$, return $\perp$. Else, return (cm, rm)

  Due to the correctness of *HKDF* there is no need to run $SBEnc_b$ as the key generated in this function is equal to the one in $SBEnc_f$.

◦ At some point A outputs the challenge messages (cm^*, rm^*). Then B proceeds as follows and passes the returned message to A:
- Query challenger with $register(ID(vk_u^*))$ and get vk_u^*.
- Query challenger with $extract(ID(sk_u^*))$ and get sk_u^*.
- Query the challenger with $test(ID(sk_u^*), ID(vk_d))$ and retrieve k_d^*.
- Compute $sk_s^* || k_c^* || k_r^* \leftarrow HKDF(k_d^*, \lambda + l_c + l_r)$
- Compute $vk_p^* \leftarrow vk_d^{sk_s^*}$ and set $ct_c := (\tau || cm^*) \oplus k_c^*$
- Compute $(cc^*, \sigma^*) \leftarrow TxEnc_f((vk_u^*, H(vk_p^*), ct_c^*), sk_u^*)$
- Compute $sk_p^* \leftarrow sk_d \cdot sk_s^*$ and set $ct_r^* := rm^* \oplus k_r^*$
- Compute $(rc^*, \sigma'^*) \leftarrow TxEnc_b((ct_r^*, vk_p^*), sk_p^*)$
- If $cc^* = \perp$ or $rc^* = \perp$, return $\perp$. Else, return $((cc^*, \sigma^*), k_d^*, (rc^*, \sigma'^*))$

◦ 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 generate by the protocol is returned, otherwise ($b = 1$) a randomly generated 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 $((cc^*, \sigma^*), k'', (rc^*, \sigma'^*))$, for any symmetric key k''.

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 *MoneyMorph* must be secure against chosen-covertext attacks.

B.4 Security Proof of Theorem 4

Theorem 4 (Bitcoin Encoding Scheme Correctness)**.** *Bitcoin encoding scheme is correct according to Def. 6.*

Proof. We start by showing that condition 1 holds. In particular, given $ctx \leftarrow TxEnc_f(cd, ca)$ we show that $TxDec_f(ctx)$ return cd^* such that $cd = cd^*$.

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

we show that $TxDec_f$ does not return $\perp$. Since $TxEnc_f$ 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, $TxDec_f$ checks the number of inputs and outputs. tx_1 has exactly one input and two outputs, along with sufficient funds, therefore $TxDec_f$ will not return $\perp$. The condition 2 holds following similar arguments. □

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

Proof. We start by showing that condition 1 holds. In particular, given $ctx \leftarrow TxEnc_f(cd, ca)$ we show that $TxDec_f(ctx)$ return cd^* such that $cd = cd^*$.

By the correctness of *Extract* function $ct[0:563]^* = ct[0:563]$, $ct[563:1147]^* = ct[564:1147]$, $vk_u^* = vk_u$ and $H(vk_p^*) = H(vk_p)$. Moreover, as H is collision-resistant, $vk_p^* = vk_p$. If $TxDec_f$ does not return $\perp$, this proves that both functionalities are correct. Now we show that $TxDec_f$ does not return $\perp$.

Since $TxEnc_f$ has not returned $\perp$ it means that $|ct|$ is exactly 1148 bytes (after padding) and sufficient fund has been associated to the accounts. Upon receiving a transaction, $TxDec_f$ checks to have a transparent input and shielded output, along the fee associated to the transparent output. tx_1 has exactly one transparent input and a shielded output, along with sufficient funds in the transparent output, therefore $TxDec_f$ will not return $\perp$.

The condition 2 holds following similar arguments. □

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

Proof. We start by showing that condition 1 holds. In particular, given $ctx \leftarrow TxEnc_f(cd, ca)$ we need to show that $TxDec_f(ctx)$ return cd^* such that $cd = cd^*$.

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

Since $TxEnc_f$ has not returned $\perp$ it means that $|ct|$ is exactly 640 bytes (after padding) and sufficient fund has been associated to the accounts. Upon receiving a transaction, $TxDec_f$ checks the number of inputs and outputs. tx_1 has exactly one input and two outputs, therefore $TxDec_f$ will not return $\perp$.

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

Theorem 7 (Ethereum Encoding Scheme Correctness)**.** *Ethereum encoding scheme is correct based on Definition 6.*

Proof. We start by showing that condition 1 holds. In particular, given $ctx \leftarrow TxEnc_f(cd, ca)$ we show that $TxDec_f(ctx)$ return cd^* such that $cd = cd^*$.

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

Since $TxEnc_f$ 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 $tx_0||tx_1$, $TxDec_f$ checks the recipient of the transaction tx_1 to be $H(vk_e)$. tx_1, along with sufficient funds in address $H(vk_p)$, therefore $TxDec_f$ will not return $\perp$.

The condition 2 holds following similar arguments. □

C Monero LRS Scheme

In this section, we present the Linkable Ring Signature (LRS) Scheme used in Monero.

Let λ be the security parameter, and let $\mathbb{Z}_p$ be a group of prime 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 ($LRS.KeyGen$, $LRS.Sign$, $LRS.Verify$) is defined as:

- $vk, sk \leftarrow LRS.KeyGen(\lambda)$: Sample $sk \leftarrow_\$ \mathbb{Z}_p$ and compute $vk := g^{sk}$.
- $\sigma \leftarrow LRS.Sign((vk_1, \ldots, vk_{n-1}, vk_n), sk_n, m)$: Sample $r \leftarrow_\$ \mathbb{Z}_p$. Compute $\mathcal{I} := sk_n \cdot H(vk_n)$ and $h_0 \leftarrow H(m||g^r||H(vk_n)^r)$. Then, sample $s_1, \ldots s_{n-1} \leftarrow_\$ \mathbb{Z}_p^{n-1}$ and compute the following series:
$$h_i := H(m||g^{s_i} \cdot vk_i^{h_{i-1}}||H(vk_i)^{s_i} \cdot \mathcal{I}^{h_{i-1}})$$
Now, solve s_0 such that $H(m||g^{s_0} \cdot vk_n^{h_{n-1}}||H(vk_n)^{s_0} \cdot \mathcal{I}^{h_{n-1}} = h_0$. 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 $\sigma := (s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I})$.
- $\{\top, \perp\} \leftarrow LRS.Verify((vk_1, \ldots, vk_n), m, \sigma)$: Parse $(s_0, s_1, \ldots, s_{n-1}, h_0, \mathcal{I}) \leftarrow \sigma$ and compute the series:
$$h_i := H(m||g^{s_i} \cdot vk_i^{h_{i-1}}||H(vk_i)^{s_i} \cdot \mathcal{I}^{h_{i-1}})$$
Return $\top$ if $h_0 = h_n$. Otherwise, return $\perp$.