Privacy-Friendly Monero Transaction Signing on a Hardware Wallet

Dusan Klinec, Vashek Matyas

Privacy-friendly Monero transaction signing on a hardware wallet

Dusan Klinec and Vashek Matyas

Masaryk University, Brno, Czech Republic,
xklinec@fi.muni.cz, matyas@fi.muni.cz

Abstract Keeping cryptocurrency spending keys safe and being able to use them when signing a transaction is a well-known problem, addressed by hardware wallets. Our work focuses on a transaction signing process for privacy-centric cryptocurrency Monero, in the hardware wallets. We designed, implemented, and analyzed a privacy-preserving transaction signing protocol that runs on a hardware wallet and protects the spending keys. Moreover, we also implemented a privacy-preserving multi-party version of the Bulletproof zero-knowledge prover algorithm, which runs on a hardware wallet with constant memory. We present the protocols and evaluate their performance on a real hardware wallet.

Keywords: Monero · transaction signing · Bulletproofs · zero-knowledge system · multi-party computation · hardware wallets.

1 Introduction

Cryptocurrencies gained popularity and increased adoption by general public in the recent years. They became a valuable asset worth protecting. In the vast majority of the cryptocurrency designs, the only thing needed to transact (spend) the coins is a cryptographic key (master key). Recently, we have seen several attacks on the software wallets storing the master keys and leading to coin thefts [4,11].

Software wallets are inherently vulnerable to malware threats, so users seek better ways to protect their cryptographic assets. One option is to use a dedicated hardware device, the hardware wallet, that stores the master key securely and performs the signature on the transactions specified by the user. The device can be equipped with a display to show the transaction details to the user (e.g., destination and the amount) and buttons to confirm the transaction information. As the hardware wallet (HW) is a special-purpose device, it has a much smaller attack surface than a PC. The HW is limited and usually needs a PC client for operation, e.g., transaction construction, transaction scanning.

Bitcoin, the first massively used cryptocurrency, does not provide much privacy to its users. i.e., the whole transaction history is stored in a public blockchain (append-only ledger), it contains source, destination addresses (cryptographic keys, pseudonymous identifiers), and transacted amounts in a clear form so the attacker can mount several chain analysis techniques to trace the

financial flow between the users. On the other hand, creating the signature and implementing the logic in the HW is usually straightforward as it requires just transaction serialization logic and ECDSA signature over the data.

We focus on a privacy-centric cryptocurrency, *Monero* [1], which is the most used privacy-centric cryptocurrency[1]. In Monero, all destination addresses are unique, and the amounts being transacted are hidden using Pedersen commitments [9]. The secure implementation of the Monero transaction signing is thus a more challenging task. Moreover, HWs are resource-constrained devices with limited memory and computational power.

For the practical testing, we choose a Trezor HW model T[2] (Trezor for short) as it represents a class of HWs with generally available processors used in the embedded devices, with around 100 kB of RAM. It runs Micropython[3], a Python version for resource-constrained devices.

Contribution: This work builds on our previous technical report [5] and for an extended version of this paper please refer to the [6]. We designed and implemented a Monero transaction signing protocol for HWs. The protocol is simple, which helps with the security analysis. Moreover, it mimics the already deployed cold-signing protocol [5] used with offline Monero wallets. We implemented the protocol to Trezor and Monero codebases, and it is being used in practice. Moreover, we designed and implemented a secure multi-party protocol (MPC) version of a zero-knowledge proving system called Bulletproof [3], which uses constant memory and works on HWs. The MPC version protects private values, from PC-based attacker. To the best of our knowledge, this is the first MPC implementation of the Bulletproof runnable on HWs.

General methods: As the HW is a resource-limited device, the general methods for converting arbitrary protocols into MPC, such as Garbled circuits [10], are not applicable, as those usually require significantly more memory and running time than we have available. We thus resort to a more effective protocol offloading design tailored specifically to the application domain, to preserve the practical usability of the resulting protocol.

1.1 Cryptocurrency primer

The elementary operation of transacting a particular amount from a sender to a receiver is called *transaction*. The transaction is an atomic state transition. Transactions are stored in the blocks, the block contains a hash of a previously generated block, thus forming a ledger of blocks, a blockchain. A new block is created every 10 minutes.

Transaction has $|\mathcal{T}^{\text{in}}| = m$ inputs $\mathcal{T}_j^{\text{in}}$, $|\mathcal{T}^{\text{out}}| = p$ outputs $\mathcal{T}_t^{\text{out}}$ and a fee. The amounts values in the input and the output side have to be equal, i.e., $\sum_{j=1}^{m} v_j^{\text{in}} = \sum_{t=1}^{p} v_t^o + \text{fee}$, so the transaction is a valid state transition (and no value is lost or created). Let's denote transaction outputs TXOs. Transaction inputs are also

[1] By the market value `https://coinmarketcap.com`, accessed on 5. 1. 2020.

[2] `https://wiki.trezor.io/Trezor_Model_T`

[3] `https://micropython.org`

called unspent transaction outputs (UTXOs). UTXOs are addresses that have a non-zero balance that can be spent. A balance can be trivially computed by replaying all transactions recorded in the blockchain. Blockchain clients usually track all UTXOs and update the state with each new block.

Transaction construction is controlled from the PC client as it scans the blockchain for UTXOs that can be spent. A user enters the transaction recipient addresses and amounts on the PC. The client then performs the transaction signing protocol with an HW to obtain a signed transaction. The signed transaction is then broadcasted to the cryptocurrency network and eventually added to a block.

The Monero user wallet has two key-pairs, (k^v, K^v) and (k^s, K^s), the *view-key*, and *spend-key*, respectively. The view-key is used to scan the blockchain for incoming transactions to the user wallet; the spend-key is required to create a signature for spending the incoming coins.

1.2 Attacker model

HWs are considered as trusted in all attacker models in this paper, i.e., HW securely stores master keys, and an attacker can gain no knowledge by observing and tampering the HW device. We only focus on an attacker controlling the PC client. The *honest-but-curious* attacker model is defined by an attacker that obeys the protocol precisely but tries to learn new information observing the protocol transcripts. The *malicious* attacker model is stronger, an attacker can arbitrarily deviate from the protocol. He can start multiple instances of protocols, interleave protocol runs, send, replay, delay, or drop any protocol messages.

1.3 Notation

We use a standard notation used in the related literature, such as [1,3]. Due to the paper application domain in the Monero transactions, we use the elliptic curve (EC) *Ed25519* [2] as a specialization of the cyclic group $\mathbb{G}$ of the prime order l, let's denote $\mathbb{Z}_l$ the ring of integers modulo l. The $\mathbb{Z}_l^*$ denotes $\mathbb{Z}_l \setminus \{0\}$. The $x \xleftarrow{\$} \mathbb{Z}_l^*$ denotes a uniform sampling of an element from $\mathbb{Z}_l^*$. Capital letters represent points on the curve $\mathbb{G}$, lower-case letters represent scalars from $\mathbb{Z}_l^*$ unless said otherwise. We use the EC additive notation for $\mathbb{G}$, e.g., $P+Q$ is a point addition, $aP = (P + \cdots + P)$ is a scalar multiplication, $0P = O$, i.e., neutral element, point in infinity. Let $\mathbb{F}^n$ denote a vector space over $\mathbb{F}$, the $\boldsymbol{a} \in \mathbb{F}^n$ is a vector from the vector space with elements $a_0, \ldots, a_{n-1} \in \mathbb{F}$. The $\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=0}^{n-1} a_i b_i$ denotes a dot-product of $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^n$, the $\boldsymbol{a} \circ \boldsymbol{b} = \{(a_i b_i)\}_i$ is element product. We also use Python notation for vector slicing, i.e., $\boldsymbol{a}_{[:l]} = (a_0, \ldots, a_{l-1})$, $\boldsymbol{a}_{[l:]} = (a_l, \ldots, a_n)$. For $k \in \mathbb{Z}_l^*$, the $\boldsymbol{k}^n \in \mathbb{F}^n$ denotes the vector $k_{|i|} = k^i$. The G is a generator of the $\mathbb{G}$, i.e., a base point. Let's define a cryptographic hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^{256}$, the $\mathcal{H}_p : \{0,1\}^* \to \mathbb{G}$ is a cryptographic hash function to curve points, $\mathcal{H}_s : \{0,1\}^* \to \mathbb{Z}_l^*$ is a cryptographic hash function to scalars. Moreover, let's define $H = \mathcal{H}_p(G)$, a point of unknown logarithm. Binary format of the scalars

and points is 256 bits long. Let's denote the binary concatenation as $||$ and a key derivation function as $\mathrm{KDF}(x) = \mathcal{H}(\mathcal{H}(x))$.

2 Transaction signing protocol

Ring signatures are signatures generated by a single private key k_π corresponding to the public key K_π which is in the ring of unrelated public keys $\mathcal{R} = \{K_0, \ldots, K_\pi, \ldots, K_n\}$. The verifier is not able to tell which $K_i \in \mathcal{R}$ generated the signature. This provides n-anonymity for the signer. Keys $K_i, i \neq \pi$ are called decoy keys. Let's define $n = |\mathcal{R}|$, i.e., the ring size.

Monero uses Schnorr-style multilayered linkable spontaneous anonymous group signatures (MLSAG) [8]. The linkability is a property that links signatures generated with the same private keys. The linked signatures have the same *key image* (explained later). Signatures with already seen key images are considered as invalid to protect from double-spending the same UTXO.

Monero uses the *Pedersen commitment* to conceal the transacted amounts and to prove that transaction input amounts are equal to transaction output amounts. A Monero *range proof* is a zero-knowledge proof that the TXO amount encoded in the scalar value $v \in \mathbb{Z}_l^*$ lies[4] in the interval of allowed values $[0, 2^N], N = 64$. The range proof is an essential part of the confidential transactions as it protects from overflows and new coin generation.

The *transaction generator* takes $\mathcal{T}^{\text{in}}$ and set of destination addresses and amounts $\mathcal{T}^{\text{out}}$ and produces a transaction with signature. The value $|\mathcal{T}^{\text{in}}|$ can be quite high and is limited by the fee the user is willing to spend for the transaction to be added to the blockchain and the current block size. The current Monero protocol version (0.15.0.1), i.e. hard-fork, specifies that for a valid transaction it holds that $2 \leq |\mathcal{T}^{\text{out}}| \leq 16$. Thus the $|\mathcal{T}^{\text{in}}|$ is the main limiting factor for the transaction generator with respect to the memory. The ring size n is fixed to 11 at the current version, but it is likely to increase in the future.

It is not feasible to construct a transaction in the HW in one pass. Thus the building process has to be separated into several steps so it can be computed with limited memory. We designed, implemented, and tested the transaction generation protocol that runs in the HW with unlimited $\mathcal{T}^{\text{in}}$ (the protocol).

Transaction signing protocol with state offloading: *The offloaded signing protocol* is described fully in the extended paper [6]. Here we explain only key ideas of the protocol construction.

A *state offloading* is required to build the transaction incrementally. Some parts of the state are sent to the host for later retrieval during the transaction construction. The protocol uses HMAC to protect the public state parts, e.g., parts of the final transaction, and an authenticated encryption (Chacha20Poly1305) for private state information, e.g., $\mathcal{T}^{\text{in}}$ private spending keys.

[4] v in Pedersen commitment $\gamma G + vH$, $\gamma \xleftarrow{\$} \mathbb{Z}_l^*$. Refer to Section 3 for more details.

All MLSAG signatures are generated in the HW, the k^s never leaves the device, while the k^v is exported to the host so it can scan all blockchain transactions to determine whether the funds were sent to the recipient wallet keys. Performing the blockchain scanning with the device would be inefficient.

The transaction is built on the host incrementally, from the information provided by the HW. In general, the host sends initial transaction information to the HW. The user is asked to confirm transaction details on the HW screen, such as destination address, amounts, and transaction fee. If confirmed, the HW generates HMACs for transaction input elements so they cannot be changed later (commitment to values).

Then all $\mathcal{T}^{\text{in}}$ are sent to the HW one by one; the HW derives required signing keys, serializes $\mathcal{T}^{\text{in}}$ to blockchain format, incrementally hashes information required for later MLSAG construction. Then destination information is sent one by one; the HW generates $\mathcal{T}^{\text{out}}$ related information, serializes $\mathcal{T}^{\text{out}}$ to the blockchain format, range proof generation (see Section 3) is handled. Finally, the MLSAG is generated per $\mathcal{T}^{\text{in}}$.

Key constuction scheme: Let's describe HMAC key construction on the $\mathcal{T}^{\text{out}}$ example, generated after user confirmation. HW returns to the host $\mathcal{T}^{\text{out}}$: $\{\mathcal{T}_i^{\text{out}}$, HMAC($\mathcal{T}_i^{\text{out}}$, key= KDF($k^{mac}$ ||"txdest"||i))$\}_i$, where k^{mac} is a random HMAC master key generated per-transaction. The HMAC keys construction prevents from changing destination specification later in the protocol by an attacker. All the offloading keys used in the protocol are generated correspondingly, i.e., the keys are unique per offloaded element to make the protocol strictly commit to the offloaded values and their ordering. The key is derived from the master keys, the domain separator, e.g., "txdest", and the item index.

Encrypt-then-reveal: In order to limit the attacker's reactivity, MLSAG signatures are returned encrypted to the host. The encryption key is returned to the host after the protocol finishes successfully.

Analysis: The protocol is based on the cold-signing protocol [5] implemented in Monero codebase, which takes all transaction inputs $\mathcal{T}^{\text{in}}$, transaction outputs $\mathcal{T}^{\text{out}}$, asks the user to confirm transaction outputs, and a fee and generates a valid Monero transaction. Cold-signing protocol is trivially secure as it is evaluated in a secure environment (offline Monero wallet), and the user confirms all transaction outputs. Our offloaded protocol mimics the cold-signing protocol. It is evaluated in a HW, which is considered a secure environment. The transaction is constructed incrementally, from the basic input blocks $\mathcal{T}^{\text{in}}$, $\mathcal{T}^{\text{out}}$ sent one by one to the HW. The user confirms the $\mathcal{T}^{\text{out}}$ on the HW, as it has a display and touch screen. After the confirmation is done, the HW generates HMAC for confirmed $\mathcal{T}^{\text{out}}$. Thus it cannot be later modified by an attacker.

Each call is guarded by a state automaton, set up in step 1 by the parameters of the transaction. This prevents from calling protocol methods in a different than expected order. Moreover, due to HMAC and encryption key construction, it is not possible to modify, reorder, reply, or drop the offloaded state elements. Protocol aborts if invalid input is provided.

The only place where the k^s is used is during spend key computation, during $\mathcal{T}^{\text{in}}$ construction. The result of the computation is offloaded in an encrypted form, which could only be used as input in the last protocol step, during MLSAG generation per each $\mathcal{T}^{\text{in}}$. MLSAG signature is generated over hash commitment $\mathfrak{m}$, which hashes the entire transaction specification $(\mathcal{T}^{\text{in}}, \mathcal{T}^{\text{out}})$. The protocol is thus secure in the malicious attacker model as cheating in each protocol step is detected by HMAC, auth tag, or state transition failure.

The only information the attacker can obtain from the protocol runs (without a need to finish the protocol) is key images corresponding to the $\mathcal{T}^{\text{in}}_j$. The key images are part of the constructed transaction. As we need the host to sort key images so some kind of order-preserving encryption would have to be used to protect key images from leaking before the protocol finish. However, we do not consider this as a required measure as the key images are computed during the blockchain scanning once the transaction sent to our wallet has been found.

Performance and space complexity: The space complexity is determined by $O(n+p)$, i.e., by a ring size and the number of the transaction outputs, i.e., $n = 11$ and $p \leq 16$ in the current Monero version. The whole ring is needed only for the MLSAG signature, which can be easily extended to support large rings. If the p is increased later, the protocol can be easily changed to offload all output-related values. The transaction signing protocol implemented in Micropython for Trezor HW was tested with various input transactions. Refer to Table 1 for performance overview data.

Table 1: Performance of the transaction signing protocol on Trezor HW. The algorithm was tested on the emulator and Trezor T HW. Configuration is a tuple (#inputs, #outputs). The first metric, "Time emu", is a runtime in an emulator, other statistics are from runs on the real hardware. "$\sum$ Steps" is protocol computation time without communication overhead, "rounds" is a total number of message round-trips. Rows with "State" show a maximal state size over the protocol, where "real" is the real size measured in the implementation, "min" is the minimal space required, without Micropython objects overhead. Note that the range-proof is not included in the statistics as it is measured separately in section 3. It is visible that the state size is constant, and timing is linear to the number of inputs.

Configuration	2-2	16-2	32-2	64-2	128-2	2-16
Time Emu [s]	9.31	29.93	58.13	106.32	209.88	21.45
Time [s]	16.90	83.22	156.82	306.74	604.09	46.13
$\sum$ Steps	12.49	56.30	106.69	207.33	408.42	36.62
Rounds	14	56	104	200	392	28
RAM [B]	41 264	42 176	42 208	42 048	58 512	41 376
State min [B]	2 385	2 385	2 385	2 385	2 385	4 406
State real [B]	5 315	5 315	5 315	5 315	5 315	9 224

3 Range proof

A range proof is a zero-knowledge proof that the amount encoded in a scalar $v \in \mathbb{Z}_l^*$ (256-bit number for Ed25519), lies in the interval $[0, 2^{64})$, without revealing the amount value. Range proof computations are the most resource expensive operations in the transaction construction (time and memory). Thus it makes sense to offload the computation to the host.

The range proofs make use of commitments $V = \gamma G + vH$, where $\gamma \xleftarrow{\$} \mathbb{Z}_l^*$ is a mask, and the V is part of the publicly stored information. If the attacker generates the masks in a special way, he can exfiltrate information about the keys or the transaction. From the binding property of the commitment scheme which relies on the discrete logarithm problem, it is infeasible to find a different $v', \gamma',$ s.t. $\gamma' G + v' H = \gamma G + vH$. The attacker already knows the amount as he observes the transaction construction, but knowing the masks enables the attacker to prove the amount to a third party (e.g., court). This poses a privacy risk as the attacker can prove that he has seen the transaction construction or knows the amount keys.

The *Bulletproof* [3] (BP) is the range proof system used in Monero. The proof size increases logarithmically with respect to the number of statements (transaction outputs). BP can prove statement of the form $M = 2^x$. We implemented the memory-optimized prover version for the Trezor HW, described in Algorithm 1.

Space complexity: Up to the while-loop on line 29, all vectors can be evaluated on-the-fly with a constant memory and low CPU overhead with just $\boldsymbol{v}, \boldsymbol{\gamma}$ stored. Vector foldings on lines 36-39 dominate the space complexity. Overall the space complexity is $O(MN)$. Please refer to full paper [6] for detailed analysis.

Implementation: We implemented the in-memory Bulletproof prover as specified in the Algorithm 1 and the verifier in Micropython and tested on the Trezor HW. The memory usage is $32(128M + 12log(M)) + O(max(log(N), M))$ B. The verification algorithm runs with $O(max(log(N), M))$ memory. Table 2 shows the performance of the prover and the verifier implemented on the Trezor.

3.1 Offloaded Bulletproofs

Due to increasing space complexity, it is not possible to generate BPs with $M \geq 4$ on the Trezor. We thus designed a new privacy-preserving secure multi-party computation protocol (MPC) to compute Bulletproofs jointly with the PC host and an HW with a constant memory on the HW side. We do not consider the time and memory requirements of the protocol running on the host in the following.

A naïve offloading protocol computes all vector-related operations by chunking, i.e., exporting encrypted vectors to a host, then asks for vector chunks to compute the intermediate results. The vectors $\boldsymbol{l}, \boldsymbol{r}$ and dot-products c_L, c_R, t

[5] Multiplying by 8^{-1} protects from small subgroup addition `https://www.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html`

Algorithm 1 Bulletproof prover. $N = 64, M = |\boldsymbol{v}|, \boldsymbol{v}$ is a vector of amounts

1 **function** BULLETPROOFPROVER($\boldsymbol{v}, \boldsymbol{\gamma}$) ▷ Input: Amounts and masks
2 $(\boldsymbol{V}, A, S, T1, T2, \tau_x, \mu, x, h, l_{0,i}, l_{1,i}, r_{0,i}, r_{1,i}) \leftarrow$ BULLETPROOFPREFIX($\boldsymbol{v}, \boldsymbol{\gamma}$)
3 $l_i \leftarrow l_{0,i} + x l_{1,i}$ ▷ Vector $\boldsymbol{l}$
4 $r_i \leftarrow r_{0,i} + x r_{1,i}$ ▷ Vector $\boldsymbol{r}$
5 $t \leftarrow \boldsymbol{l} \cdot \boldsymbol{r}$; $x' \leftarrow \mathcal{H}_s(x||x||\tau_x||\mu||t)$ ▷ Evaluated with const. memory up to here
6 $(\boldsymbol{L}, \boldsymbol{R}, a'_0, b'_0) \leftarrow$ BULLETPROOFLOOP($\boldsymbol{l}, \boldsymbol{r}, \mathring{\boldsymbol{G}}, \boldsymbol{y}^{-|\mathring{H}|} \circ \mathring{\boldsymbol{H}}, MN, -1, x', x'$)
7 **return** $(\boldsymbol{V}, A, S, T1, T2, \tau_x, \mu, \boldsymbol{L}, \boldsymbol{R}, a'_0, b'_0, t)$

8 **function** BULLETPROOFPREFIX($\boldsymbol{v}, \boldsymbol{\gamma}$) ▷ varint encodes integer to bytes
9 $\mathring{G}_{i \in [0,\dots,MN)} \leftarrow \mathcal{H}_p(\mathcal{H}(\text{"bulletproof"}||H||\text{varint}(2i+1)))$
10 $\mathring{H}_{i \in [0,\dots,MN)} \leftarrow \mathcal{H}_p(\mathcal{H}(\text{"bulletproof"}||H||\text{varint}(2i)))$
11 $V_j \leftarrow \gamma_j G + v_j H$ ▷ Compute commitment vector $\boldsymbol{V}$ used on line 18
12 $\alpha \xleftarrow{\$} \mathbb{Z}_l^*$ ▷ expand($\boldsymbol{v}$)=$\boldsymbol{x}$: $\sum_{i=0}^{63} 2^i x_{64j+i} = v_j, j \in [0, |v|), x_i \in \{0, 1\}$
13 $\boldsymbol{a_L} \leftarrow \text{expand}(\boldsymbol{v})$; $\boldsymbol{a_R} = \boldsymbol{a_L} - \boldsymbol{1}^{MN}$
14 $A \leftarrow 8^{-1} \left(\alpha G + \sum_{i=0}^{MN-1} a_{L,i} \mathring{G}_i + a_{R,i} \mathring{H}_i\right)$ ▷ Commitment over values[5]
15 $\rho, R \xleftarrow{\$} (\mathbb{Z}_l^*)^2$ ▷ randVct(j) = $\boldsymbol{x} : |\boldsymbol{x}| = MN, x_i = H_s(\text{"mask"}||R||i||j)$
16 $\boldsymbol{s_L} \leftarrow \text{randVct}(0), \boldsymbol{s_R} \leftarrow \text{randVct}(1)$
17 $S \leftarrow 8^{-1} \left(\rho G + \sum_{i=0}^{MN-1} s_{L,i} \mathring{G}_i + s_{R,i} \mathring{H}_i\right)$ ▷ Commitment over random masks
18 $y \leftarrow \mathcal{H}_s(\mathcal{H}_s(\boldsymbol{V})||A||S)$; $z \leftarrow \mathcal{H}_s(y)$ ▷ Compute commitments over inputs
19 $\boldsymbol{l_0} \leftarrow \boldsymbol{a_L} - z\boldsymbol{1}^{MN}$; $\boldsymbol{l_1} \leftarrow \boldsymbol{s_L}$ ▷ Evaluated with const. memory, with $\boldsymbol{v}$
20 $\zeta_i \leftarrow z^{2+\lfloor i/N \rfloor} 2^{i\%N}$ ▷ Evaluated with const. memory and time
21 $\boldsymbol{r_0} \leftarrow ((\boldsymbol{a_R} + z) \circ \boldsymbol{y}^{MN}) + \boldsymbol{\zeta}$; $\boldsymbol{r_1} \leftarrow \boldsymbol{s_R} \circ \boldsymbol{y}^{MN}$
22 $t_1 \leftarrow \boldsymbol{l_0} \cdot \boldsymbol{r_1} + \boldsymbol{l_1} \cdot \boldsymbol{r_0}$; $t_2 \leftarrow \boldsymbol{l_1} \cdot \boldsymbol{r_1}$
23 $\tau_1, \tau_2 \xleftarrow{\$} (\mathbb{Z}_l^*)^2$
24 $T_1 = 8^{-1} (\tau_1 G + t_1 H)$; $T_2 = 8^{-1} (\tau_2 G + t_2 H)$
25 $x \leftarrow \mathcal{H}_s(z||z||T1||T2)$
26 $\tau_x \leftarrow \tau_1 x + \tau_2 x^2 + \sum_{i=0}^{M} \gamma_i z^{i+2}$; $\mu \leftarrow \rho x + \alpha$
27 **return** $(\boldsymbol{V}, A, S, T1, T2, \tau_x, \mu, x, h, l_{0,i}, l_{1,i}, r_{0,i}, r_{1,i})$

28 **function** BULLETPROOFLOOP($\boldsymbol{a'}, \boldsymbol{b'}, \boldsymbol{G'}, \boldsymbol{H'}, n', c, w, x'$)
29 **while** $n' > 1$ **do**
30 $\bar{n} \leftarrow n'$; $n' \leftarrow n'/2$; $c \leftarrow c + 1$
31 $c_L \leftarrow a'_{[0,\dots,n')} \cdot b'_{[n',\dots,\bar{n})}$
32 $c_R \leftarrow a'_{[n',\dots,\bar{n})} \cdot b'_{[0,\dots,n')}$
33 $L_c \leftarrow 8^{-1} \left(\left(\sum_{i=0}^{n'} a'_i \quad G'_{i+n'} + b'_{i+n'} H'_i\right) + (c_L x') H\right)$
34 $R_c \leftarrow 8^{-1} \left(\left(\sum_{i=0}^{n'} a'_{i+n'} G'_i \quad + b'_i \quad H'_{i+n'}\right) + (c_R x') H\right)$
35 $w \leftarrow \mathcal{H}(w, L_c, R_c)$
36 $a'_{i \in [0,\dots,n')} \leftarrow w \quad a'_i + w^{-1} a'_{i+n'}$ ▷ Scalar vector folding
37 $b'_{i \in [0,\dots,n')} \leftarrow w^{-1} b'_i + w \quad b'_{i+n'}$ ▷ Folding reduces vector size by 2
38 $G'_{i \in [0,\dots,n')} \leftarrow w^{-1} G'_i + w \quad G'_{i+n'}$ ▷ Hadamard folding
39 $H'_{i \in [0,\dots,n')} \leftarrow w \quad H'_i + w^{-1} H'_{i+n'}$ ▷ Folding reduces vector size by 2
40 **return** $(\boldsymbol{L}, \boldsymbol{R}, a'_0, b'_0)$ ▷ vector $\boldsymbol{L}$ composed from L_c

Table 2: BP performance on the Trezor T HW. Verifier is faster than prover and requires significantly lower memory. Prover time and space complexity increase linearly to the input size. The *RAM min* shows the minimal amount of RAM required to cover the BP generation w.r.t. dominating cost - vectors, excluding the constants and state required for on-the-fly evaluation. The difference between Total RAM and minimal RAM is due to memory handling mechanisms in Micropython and constant memory overhead for on-the-fly vector evaluation.

Prover outputs	1	2	4	8	16
Time [s]	16.88	30.55	57.40	121.21	246.57
RAM min [B]	4 480	8 640	16 896	33 344	66 176
RAM total [B]	9 648	13 536	22 224	39 696	74 400
Verifier outputs	1	2	4	8	16
Time [s]	5.12	9.58	18.56	39.00	80.40
RAM total [B]	5 664	6 256	6 912	7 616	8 512
Proof size [B]	704	800	928	1120	1440

can be computed incrementally as $t = \sum l_i r_i$. This yields a constant memory protocol, but with high communication overhead.

We present basic offloading techniques in the following paragraphs, which are used to transform the in-memory prover to the privacy-preserving MPC prover with constant memory.

Dot-product offloading: We can evaluate dot-products and foldings on the host privately, using homomorphic property of a *blinding*. We export the vectors $\pi_{a'}\boldsymbol{a'}, \pi_{b'}\boldsymbol{b'}$ to the host, where $\pi_{a'}, \pi_{b'} \xleftarrow{\$} \mathbb{Z}_l^{*2}$ are random blinding scalars known only to the HW. The host computes the dot-product of blinded vectors $\bar{r} = \pi_{a'}\boldsymbol{a'} \cdot \pi_{b'}\boldsymbol{b'} = \sum \pi_{a'} a'_i \pi_{b'} b'_i = \pi_{a'}\pi_{b'} \sum a'_i b'_i$ and returns $\bar{r}$, i.e., blinded value r, to the HW. The HW then unblinds the $\bar{r}$ as $\pi_{a'}^{-1}\pi_{b'}^{-1}\bar{r} = r = \boldsymbol{a'} \cdot \boldsymbol{b'}$ to get the dot-product.

As the scalars are from $\mathbb{Z}_l^*$ and vector elements are essentially random, the attacker cannot infer $\boldsymbol{a'}$ from $\pi_{a'}\boldsymbol{a'}$. The $\pi_{a'}\boldsymbol{a'_0} = z$ does not have a unique factorization, i.e., $\forall z, x\ \exists y : xy = z;\ y = zx^{-1}$. Thus, a blinded vector is indistinguishable from an unblinded one for an attacker. Moreover, each element is divisor of 1 in $\mathbb{Z}_l^*$ so we cannot extract the blinding masks by $\mathrm{GCD}(\pi l_0, \pi l_1)$ as it is undefined.

Folding offloading: A vector folding is defined as $a'_{i\in(0,\dots,n']} = wa'_i + w^{-1}a'_{i+n'}$, before folding it holds $|\boldsymbol{a'}| = 2n'$, after the fold $|\boldsymbol{a'}| = n'$. Computing the folding on the host saves CPU and communication round-trips. Only the w is needed for the host to compute the folding, but it is desired to keep internal constants secret to preserve the privacy-preserving property of the offloading. Thus $\{w, w^{-1}\}$ are incorporated into blinding masks.

We have two distinct blinding constants for one vector. One for the lower half (LO), the other for the higher half (HI): $\{\pi_{a'_{LO}}, \pi_{a'_{HI}}\}$. The folding is then computed in two parts, as we preserve the LO/HI blinding also for the folding result vector, as shown in equation 1, so this blinding scheme is composable.

Let's thus define $\boldsymbol{x_{LO}} = \boldsymbol{x}_{[:\frac{n}{2}]}$ and $\boldsymbol{x_{HI}} = \boldsymbol{x}_{[\frac{n}{2}:]}$ for vector $\boldsymbol{x}$ of length n, $Lh(0)$=LO, $Lh(1)$=HI, then define folding constants as $\phi_{\boldsymbol{x},j}, \boldsymbol{x} \in \{\boldsymbol{a'}, \boldsymbol{b'}, \boldsymbol{G'}, \boldsymbol{H'}\}$, $j \in [0,3]$, $\phi_{\boldsymbol{x},j} = \theta_{\boldsymbol{x}_{Lh(\lfloor j/2 \rfloor)}} w^{1-(2j\%4)} \pi_{\boldsymbol{x}_{Lh(j\%2)}}$, e.g., $\phi_{\boldsymbol{a'},0} = \theta_{\boldsymbol{a'_{LO}}} w \pi^{-1}_{\boldsymbol{a'_{LO}}}$, where $\boldsymbol{\theta}$ are randomly generated blinding masks from $\mathbb{Z}_l^*$ for the next round. The ϕ is constructed so it cancels the blinding mask π, multiplies by $w^{\{1,-1\}}$, and multiplies by a new blinding mask θ. It is also easy to observe that folding offloading is compatible with the dot-product offloading, as $c_L = \boldsymbol{a'_{LO}} \cdot \boldsymbol{b'_{HI}}, c_R = \boldsymbol{a'_{HI}} \cdot \boldsymbol{b'_{LO}}$.

The blinding technique differs from the dot-product offloading due to constants $\{w, w^{-1}\}$ being used. We need to have distinct blinding masks for each term in the folding sum, so an attacker cannot extract the w from the blinding masks. The folding offloading works in the following way:

$$
\begin{aligned}
\theta_{a'_{LO}} \boldsymbol{a'}_{\left[:\frac{n'}{2}\right]} &\leftarrow \overbrace{\left(\theta_{a'_{LO}} w \pi^{-1}_{a'_{LO}}\right)}^{\phi_{\boldsymbol{a'},0}} \left(\pi_{a'_{LO}} \boldsymbol{a'}_{[:n']}\right) + \overbrace{\left(\theta_{a'_{LO}} w^{-1} \pi^{-1}_{a'_{HI}}\right)}^{\phi_{\boldsymbol{a'},1}} \left(\pi_{a'_{HI}} \boldsymbol{a'}_{[n':]}\right) \\
\underbrace{\theta_{a'_{HI}} \boldsymbol{a'}_{\left[\frac{n'}{2}:\right]}}_{\substack{\text{High half of new } \boldsymbol{a'} \\ \text{blinded by } \theta_{a'_{HI}}}} &\leftarrow \underbrace{\left(\theta_{a'_{HI}} w \pi^{-1}_{a'_{LO}}\right)}_{\phi_{\boldsymbol{a'},2}} \underbrace{\left(\pi_{a'_{LO}} \boldsymbol{a'}_{[:n']}\right)}_{\substack{\text{Low half of } \boldsymbol{a'} \\ \text{blinded by } \pi_{a'_{LO}}}} + \underbrace{\left(\theta_{a'_{HI}} w^{-1} \pi^{-1}_{a'_{HI}}\right)}_{\phi_{\boldsymbol{a'},3}} \underbrace{\left(\pi_{a'_{HI}} \boldsymbol{a'}_{[n':]}\right)}_{\substack{\text{High half of } \boldsymbol{a'} \\ \text{blinded by } \pi_{a'_{HI}}}}
\end{aligned}
\tag{1}
$$

Initial $\boldsymbol{G'}, \boldsymbol{H'}$ folding: The folding of the $\boldsymbol{G'}, \boldsymbol{H'}$ cannot be performed as defined above as the vectors are protocol constants in the first round (known to attacker), i.e., $\pi_{G'} = \pi_{H'} = 1$. Thus the attacker could extract w from the $\boldsymbol{\phi}$ and unblind the folded vectors. We define folding constants $\boldsymbol{\phi^{(0)}}$ for the first round as in the equation 2: $\phi^{(0)}_{\boldsymbol{x},j}, \boldsymbol{x} \in \{\boldsymbol{G'}, \boldsymbol{H'}\}, j \in [0,3]$, $\phi^{(0)}_{\boldsymbol{x},j} = \theta_{\boldsymbol{x}_{Lh(\lfloor j/2 \rfloor)}} w^{(2j\%4)-1} + (1 - j\%2)\pi_{x_{LO}}$, e.g., $\phi^{(0)}_{\boldsymbol{G'},0} = \theta_{\boldsymbol{G'_{LO}}} w^{-1} + \pi_{\boldsymbol{G'_{LO}}}$.

The host computes the folding with the $\boldsymbol{G'}, \boldsymbol{H'}, \boldsymbol{\phi^{(0)}}$, the HW then generates a vector of correction points $\pi_{G'_{LO}} \boldsymbol{G'_{LO}}$ and returns it to the host so the host can remove extraneous component caused by the additive blinding mask $\pi_{G'_{LO}}$.

$$
\begin{aligned}
\theta_{G'_{LO}} \boldsymbol{G'}_{[:\frac{n'}{2}]} + \pi_{G'_{LO}} \boldsymbol{G'_{LO}} &\leftarrow \left(\theta_{G'_{LO}} w^{-1} + \pi_{G'_{LO}}\right) \boldsymbol{G'}_{[:n']} + \left(\theta_{G'_{LO}} w\right) \boldsymbol{G'}_{[n':]} \\
\theta_{G'_{HI}} \boldsymbol{G'}_{[\frac{n'}{2}:]} + \pi_{G'_{LO}} \boldsymbol{G'_{LO}} &\leftarrow \left(\theta_{G'_{HI}} w^{-1} + \pi_{G'_{LO}}\right) \boldsymbol{G'}_{[:n']} + \left(\theta_{G'_{HI}} w\right) \boldsymbol{G'}_{[n':]}
\end{aligned}
\tag{2}
$$

$\boldsymbol{L_c}, \boldsymbol{R_c}$ offloading: Observe that L_c from line 33 contains 3 independent components: $L_c = 8^{-1} \left(\left(\left(\sum_{i=0}^{n'} a'_i G'_{i+n'}\right) + \left(\sum_{i=0}^{n'} b'_{i+n'} H'_i\right)\right) + (c_L x') H\right)$. Each component can be computed by the host with blinded vectors. The c_L is offloaded dot-product, host returns $\pi_{a'_{LO}} \pi_{b'_{HI}} c_L$. The sum $\sum_{i=0}^{n'} a'_i G'_{i+n'}$ is computed from the blinded vectors in a similar way, the host returns: $\overline{L_{cA}} = \pi_{a'_{LO}} \pi_{G'_{HI}} \sum_{i=0}^{n'} a'_i G'_{i+n'}$, the other sum is analogical. The HW unblinds the components and computes L_c.

Analysis: The offloaded Bulletproof prover as defined in Algorithm 2 was implemented in the Micropython for the Trezor and performance of the implementation was evaluated. Please refer to Table 3 and Fig 1 for performance metrics. *Attacker constraints:* As we transformed the Algorithm 1 to the Algorithm 2 using offloading steps that preserve the privacy of the computation, the protocol remains secure in the honest-but-curious attacker model. The malicious attacker could tamper the intermediate results to learn new information or break the security properties of the protocol. However, such manipulation leads to an invalid proof with overwhelming probability due to the use of a cryptographic hash function on line 16. Rejection of an invalid proof leads to attack being detected. We could also run a Bulletproof verifier on the generated proofs, abort the transaction signing, and alert the user on the invalid proof. As in-memory verifier runs in constant memory, the protocol becomes malicious attacker resistant; however, the running time increases. There might be more effective methods to verify intermediate results or catch attacker cheating with high probability. Such extensions are left for future work.

Table 3: Offloaded BP performance on the Trezor emulator and Trezor, parameters: $b_{tch} = 32, n_{thr} = 32$. Offloaded version is faster and requires only constant memory compared to in-memory prover. The table shows a total time and memory consumption for the HW PC-based emulator and the HW. The performance statistics for previous in-memory implementation is included for comparison. The HW part contains maximum RAM needed for all steps of the algorithm, which gives minimal RAM needed for the offloaded algorithm. Real total RAM usage is higher due to Micropython memory management, message recoding, serialization, etc.

Prover outputs	1	2	4	8	16
Time Emu [s]	6.97	9.69	14.75	25.48	44.81
Total RAM Emu [B]	24 896	25 056	25 120	25 408	25 632
Time HW [s]	25.10	37.49	59.03	99.96	184.02
Total RAM HW [B]	8 768	8 928	9 488	10 656	12 816
Max state RAM used [B]	5 576	7 720	7 848	8 040	8 360
Time HW in-mem [s]	16.88	30.55	57.40	121.21	246.57
RAM HW in-mem [B]	9 648	13 536	22 224	39 696	74 400

4 Related work

The work in [7] presents a signature protocol for a Ledger[6] HW (Ledger for short). Ledger is an HW with a secure element (SE). Using the SE and the overall architecture limits the usable RAM to a few tens kB. Thus they had to

[6] `https://www.ledger.com`

Algorithm 2 Bulletproof prover with offloading. b_{tch} is a number of elements to offload in one batch, n_{thr} is a n' threshold for in-memory finish

function BulletproofProverOffloaded($\boldsymbol{v}, \boldsymbol{\gamma}$)
  $(\boldsymbol{V}, A, S, T1, T2, \tau_x, \mu, x, h, l_{0,i}, l_{1,i}, r_{0,i}, r_{1,i}) \leftarrow$ BulletproofPrefix($\boldsymbol{v}, \boldsymbol{\gamma}$)
  $t \leftarrow 0$; $c \leftarrow -1$; $n' \leftarrow MN$; $\boldsymbol{\pi} \xleftarrow{\$} (\mathbb{Z}_l^*)^8$ ▷ New random blinding masks $\boldsymbol{\pi}$
  for $i \in \left[0, \ldots, \frac{MN}{b_{tch}}\right)$ **do** ▷ Compute t, export blinded $\boldsymbol{l}, \boldsymbol{r}$ vectors
    $\boldsymbol{l_c} \leftarrow l_{0,j} + x l_{1,j}$; $\boldsymbol{r_c} \leftarrow r_{0,j} + x r_{1,j},\ j \in [i b_{tch}, (i+1) b_{tch})$
    $\bar{\boldsymbol{l_c}} \leftarrow \pi_{a'_{\delta(i)}} \boldsymbol{l_c}$ ▷ $\delta(x) = x < n'$? LO : HI, $\bar{\boldsymbol{l_c}}$ means blinded $\boldsymbol{l_c}$
    $\overline{\boldsymbol{r_c}} \leftarrow \pi_{b'_{\delta(i)}} \boldsymbol{r_c}$; $t \leftarrow t + \boldsymbol{l_c} \cdot \boldsymbol{r_c}$
    Send($\bar{\boldsymbol{l_c}}, \overline{\boldsymbol{r_c}}$) ▷ $\bar{\boldsymbol{l_c}}, \overline{\boldsymbol{r_c}}$ are $\bar{\boldsymbol{a'}}, \bar{\boldsymbol{b'}}$ for the first while iteration
  $w \leftarrow x' \leftarrow \mathcal{H}_s(x||x||\tau_x||\mu||t)$ ▷ $t = l \cdot r$
  Send(y) ▷ The host needs y to compute $\boldsymbol{H'}$ and L_c, R_c related sums
  while $n' > 1$ **do** ▷ The y is public, computable from the final proof
    $\bar{n} \leftarrow n'$; $n' \leftarrow n'/2$; $c \leftarrow c + 1$
    Receive($\overline{cL}, \overline{cR}, \overline{L_{cA}}, \overline{L_{cB}}, \overline{R_{cA}}, \overline{R_{cB}}$) ▷ $\overline{L_{cA}}$ is first blinded sum from the L_c
    $L_c \leftarrow 8^{-1} \left(\pi^{-1}_{a'_{LO}} \pi^{-1}_{G'_{HI}} \overline{L_{cA}} + \pi^{-1}_{b'_{HI}} \pi^{-1}_{H'_{LO}} \overline{L_{cB}} + x' \pi^{-1}_{a'_{LO}} \pi^{-1}_{b'_{HI}} \overline{c_L} H\right)$
    $R_c \leftarrow 8^{-1} \left(\pi^{-1}_{a'_{HI}} \pi^{-1}_{G'_{LO}} \overline{R_{cA}} + \pi^{-1}_{b'_{LO}} \pi^{-1}_{H'_{HI}} \overline{R_{cB}} + x' \pi^{-1}_{a'_{HI}} \pi^{-1}_{b'_{LO}} \overline{c_R} H\right)$
    $w \leftarrow \mathcal{H}(w||L_c||R_c)$; $\boldsymbol{\theta} \xleftarrow{\$} (\mathbb{Z}_l^*)^8$ ▷ Compute w, generate blindings $\boldsymbol{\theta}$
    if $n' \leq n_{thr}$ **then** ▷ Finish in-memory with original algorithm
      Receive$(\bar{\boldsymbol{a'}}, \bar{\boldsymbol{b'}}, \overline{\boldsymbol{G'}}, \overline{\boldsymbol{H'}})$; Unblind to obtain $\{\boldsymbol{a'}, \boldsymbol{b'}, \boldsymbol{G'}, \boldsymbol{H'}\}$
      $(\boldsymbol{L}, \boldsymbol{R}, a'_0, b'_0) \leftarrow$ BulletproofLoop($\boldsymbol{a'}, \boldsymbol{b'}, \boldsymbol{G'}, \boldsymbol{H'}, n', c, w, x'$)
      return $(\boldsymbol{V}, A, S, T1, T2, \tau_x, \mu, \boldsymbol{L}, \boldsymbol{R}, a'_0, b'_0, t)$ ▷ Combine $\boldsymbol{L}, \boldsymbol{R}$
    Send$\left(\phi^{(c)}_{x,l} : x \in \{\boldsymbol{a'}, \boldsymbol{b'}, \boldsymbol{G'}, \boldsymbol{H'}\}, l \in [0, 3]\right)$ ▷ Compute and send blindings
    if $c = 0$ **then**
      Compute and send folding correction points for $\boldsymbol{G'}, \boldsymbol{H'}$, by chunks
    $\boldsymbol{\pi} \leftarrow \boldsymbol{\theta}$ ▷ Update blinding masks for the next round

implement a more low-level protocol with basic operations such as: generate key image, $\mathcal{H}_s(x)$, xP, get sub-address secret key, etc.

Low-level cryptographic operations are computed in the device, acting as crypto proxy. The protocol is tightly integrated into the Monero codebase, and it imposes maintainability challenges as a Monero algorithm change usually requires an HW signing protocol change. The low-level protocol design makes the security analysis difficult as the information flow is quite complicated, and the attacker can call several methods in an arbitrary order, which can lead to information leak and potential vulnerability.

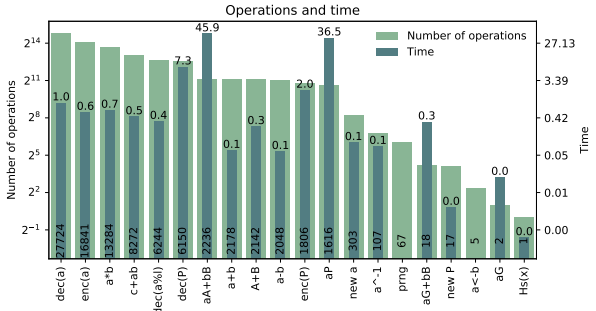

Fig. 1: Privacy-preserving multi-party BP with $M = 16$ cryptographic operations and timing breakdown. The most expensive operations are aP and point recodings.

To the best of our knowledge, the overall protocol is not documented nor analyzed. The low-level commands are documented in [7] only. The protocol does not use a state machine to guard the command calls.

To the best of our knowledge, there is no other Monero transaction signing protocol published nor used. Our protocol addresses issues with the security analysis, works on higher abstraction level, has very simple interface and thus reduced attack surface, it is more stable over protocol changes, easy to maintain, and needs less message round trips. Moreover, we compute the Bulletproofs in HW thus protecting blinding masks.

5 Conclusion

We designed, implemented, and tested a secure Monero transaction signing protocol for HWs. We designed and analyzed the memory and time complexity of the zero-knowledge proofs (range-proofs, Bulletproofs [3]), algorithms focused on low-memory consumption so they can be computed in HW. The memory consumption is linear in the number of inputs / UTXOs. The results can be easily applied to other protocols based on ring signatures, Pedersen commitments, and Bulletproof range proofs.

We also designed, implemented, and tested a privacy-preserving two-party Bulletproofs computation protocol with constant memory, enabling the computation of large input instances securely and in a reasonable time. This is the first privacy-preserving Bulletproof prover implementation running on an HW in constant memory. Techniques used in the protocol are applicable to similar protocols like Bulletproof, and Bulletproof also has several applications outside of Monero.

The implemented protocols are practically usable. The transaction signing protocol has been deployed since Nov. 7, 2018, integrated both to Trezor and

Monero codebases. All implemented sources are available online under a permissive open source license at: `https://github.com/ph4r05/monero-tx-paper`.

Acknowledgement: We thank our colleagues Petr Švenda and Marek Sýs, who provided valuable insights and ideas that helped to improve the protocols. Thanks also go to SatoshiLabs employees, Tomáš Sušánka, Jan Pochyla and Ondřej Vejpustek who did the security review of the design and implementation and helped significantly with simplifying the protocol implementation. We also thank the anonymous reviewers for their feedback and suggestions for improvement, and to Daniel Slamanig for shepherding the final revisions of our submission. This work was partly supported by the Czech Science Foundation project 20-03426S. For an extended paper please refer to the [6].

References

1. Alonso, K.M.: Zero to monero: First edition. `https://www.getmonero.org/library/Zero-to-Monero-1-0-0.pdf` (2018), accessed: 20. Feb 2020
2. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering **2**(2), 77–89 (Sep 2012). https://doi.org/10.1007/s13389-012-0027-1
3. Bunz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. pp. 315–334 (05 2018). https://doi.org/10.1109/SP.2018.00020
4. Goodin, D.: Official monero website is hacked to deliver currency-stealing malware. `https://arstechnica.com/information-technology/2019/11/official-monero-website-is-hacked-to-deliver-currency-stealing-malware` (2019), accessed: 26. Feb 2020
5. Klinec, D.: Monero wallet trezor integration. `https://github.com/ph4r05/monero-trezor-doc` (2018), accessed: 26. Feb 2020
6. Klinec, D., Matyas, V.: Privacy-friendly monero transaction signing on a hardware wallet, extended version. Cryptology ePrint Archive, Report 2020/281 (2020), `https://ia.cr/2020/281`
7. Mesnil, C.: Ledger device for Monero. online (2019), `https://github.com/LedgerHQ/ledger-app-monero`, Accessed: 20. Feb 2020
8. Noether, S.: Ring signature confidential transactions for monero. Cryptology ePrint Archive, Report 2015/1098 (2015), `https://eprint.iacr.org/2015/1098`
9. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Advances in Cryptology, CRYPTO '91. pp. 129–140. Springer Berlin Heidelberg (1992)
10. Yao, A.C.: Protocols for secure computations. In: 23rd Annual Symposium on Foundations of Computer Science. pp. 160–164. IEEE (1982). https://doi.org/10.1109/SFCS.1982.88
11. Young, J.: Malware Steals User Funds & Bitcoin Wallet Keys From PCs. `https://cointelegraph.com/news/malware-steals-user-funds-bitcoin-wallet-keys-from-pcs-bitcoin-altcoins-targeted` (2017), accessed: 26. Feb 2020