Privacy-friendly Monero transaction signing on a hardware wallet, extended version

Dusan Klinec and Vashek Matyas

Masaryk University, Brno, Czech Republic

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 [Goo19; You17; Cim20]

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* [Alo18], 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 [Ped92]. 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 hardware wallet model T [Lab19] (Trezor for short) as it represents a class of hardware wallets with generally available processors used in the embedded devices, with around 100 kB of RAM. It runs Micropython[2], a Python version for resource-constrained

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

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

devices. A different hardware wallet class is one with a Secure Element that has 2-10 kB of RAM. We do not aim for the Secure Element class as the transaction signing protocol already exists [Mes17], having different security properties and performance stemming from the low available memory. Our protocol works on a higher abstraction level; it has a simple and concise interface that enables easier security analysis, the protocol is more stable over protocol changes, easy to maintain, and needs less message round trips. Moreover, we provide a privacy-friendly range proofs generation.

Contribution: This work builds on our previous technical report [Kli18]. 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 [Kli18] 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 [Bun+18], 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 [Yao82], 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 $\mathcal{T}^{\text{in}}_j$, $|\mathcal{T}^{\text{out}}| = p$ outputs $\mathcal{T}^{\text{out}}_t$ 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 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 messages to/from the HW.

Attacker's goals are: a) to learn new information not derivable from the input parameters, e.g., master keys, secrets, random masks that can be used later to prove some statements about the transaction to a third party, for example, the amount being transacted; b) to spend the user-owner coins, change the transaction address or amount without user knowing about it; c) cause damage or financial loss to the user; d) embed information in the created transaction, possibly compromising senders or recipient privacy.

1.3 Notation

We use a standard notation used in the related literature, such as [Alo18; Bun+18]. Due to the paper application domain in the Monero transactions, we use the elliptic curve (EC) *Ed25519* [Ber+12] 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 $x \overset{\$}{\leftarrow} \mathbb{Z}_l^*$ 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) [Noe15]. 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[3] 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.

The transaction contains large elements such as range proof and signatures for each $\mathcal{T}^{\text{in}}$ thus, it is not feasible to construct the transaction in the HW in one pass. Thus the building process has to be

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

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).

A *state offloading* is required to build the transaction incrementally as HWs are memory-constrained devices. Some parts of the state have to be 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. Let's denote the $\hat{x} = x \mid\mid \text{HMAC}(x)$ and $\widetilde{x} = \text{Enc}(x)$, respectively. The keys are systematically generated by a strategy described in the *Step 1* of the protocol.

The offloaded signing protocol is described in the following section. Due to space limitations, we do not present the original signing protocol as currently implemented in Monero. For a comprehensive description, please refer to the [Alo18]. 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. 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 some 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 is handled. Finally, the MLSAG is generated per $\mathcal{T}^{\text{in}}$.

Step 1: Host sends to the HW basic transaction parameters: TsxData: $\{$*version, unlock_ time,* $|\mathcal{T}^{in}|$, *n, fee, sub-address major index* $\ell\!\uparrow$, $\mathcal{T}^{out}\}$. The destination is defined as: $\mathcal{T}_i^{\text{out}} = \{v_i^o, K_i^{o,v}, K_i^{o,s}\}$. The HW allocates and initializes internal state. The HW generates a transaction secret and public keys: $r \overset{\$}{\leftarrow} \mathbb{Z}_l^*$ and $R = rG$. Then master offloading keys are generated in the following way: $k^{mst} = \text{KDF}(\text{TsxData}||r||x \overset{\$}{\leftarrow} \mathbb{Z}_l^*)$, $k^{enc} = \text{KDF}(\text{"enc"}||k^{mst})$, $k^{mac} = \text{KDF}(\text{"mac"}||k^{mst})$.

The HW displays a confirmation prompt to the user to confirm the transaction fee and destination address and amounts.

Key constuction scheme: Once user confirms the data validity, the HW generates HMACs for each $\mathcal{T}^{\text{out}}$: $\{\mathcal{T}_i^{\text{out}}, \text{HMAC}(\mathcal{T}_i^{\text{out}}, \text{key= KDF}(k^{mac}\ ||\text{"txdest"}||i))\}_i$. 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 the 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. In the remaining text, we omit the HMAC key construction from the notation for brevity, unless the HMAC key is constructed differently than described.

The essential part of the transaction construction is a serialization of transaction-related information to the blockchain format and hashing the serialized data to a *transaction prefix* hash, $\mathfrak{m}_p$ and *pre-MLSAG* message hash $\mathfrak{m}$. The hashes $\mathfrak{m}_p, \mathfrak{m}$ are used later in the MLSAG. As the $\mathfrak{m}_p$ and $\mathfrak{m}$ relate to all transaction information, they are constructed incrementally as the transaction data such as $\mathcal{T}_i^{\text{in}}$, $\mathcal{T}_j^{\text{out}}$ are being processed. Step 1 checks for additional conditions, such as the change address is correctly constructed, i.e., we can derive the spending keys and that the change address is among destinations if applicable. The step 1 returns the $\{\text{HMAC}(\mathcal{T}_i^{\text{out}})\}_i$.

Step 2: The host sends $\mathcal{T}_i^{\text{in}}$: $\{$*outputs*, π, R, $\boldsymbol{R^a}$, t, v^{in}, γ^{in}, $\ell\!\downarrow_i\}$, where *outputs*: $\{\text{idx}_{i,j}, K_{i,j}^o, C_{i,j}^{\text{in}}\}_{j\in[1,n]}$ is an array that constitutes a ring $\mathcal{R}_i$ for a given $\mathcal{T}_i^{\text{in}}$, the $\text{idx}_{i,j}$ is an absolute blockchain index of the $\mathcal{T}_{i,j}^{\text{in}}$, the $K_{i,j}^o$ is an output key the $\mathcal{T}_{i,j}^{\text{in}}$ has been sent to, the $C_{i,j}^{\text{in}}$ is an amount commitment. The π is an index of the $\mathcal{T}_{i,\pi}^{\text{in}}$ for which the transaction signer knows the private keys, other $\mathcal{T}_{i,j}^{\text{in}} : j \neq \pi$ in the ring are decoys. It holds that $C_{i,\pi}^{\text{in}} = \gamma^{\text{in}}G + v^{\text{in}}H$. The t is an index of the $\mathcal{T}_{i,\pi}^{\text{in}}$ in the transaction output array of the sending transaction. The t is required to correctly derive transaction spending keys. The

R and $\boldsymbol{R^a}$ are sending transaction public keys. The $\ell{\downarrow}_i$ is a sub-address minor index (explained below). The π is later omitted from the index tuple to simplify the notation when we talk about $\mathcal{T}^{\text{in}}_{i,\pi}$.

Sub-addresses is a mechanism that supports having multiple different wallet addresses associated with the same wallet secret so recipients can specify one-time recipient addresses for better payment tracking. The sub-address is specified by a tuple $(\ell{\uparrow}, \ell{\downarrow})$, i.e., the major and minor sub-address index, where by convention the $(0, 0)$ keys map to standard keys (k^v, k^s). Then $k^{s,(\ell\uparrow,\ell\downarrow)} = \mathcal{H}_s(\text{"}SubAddr\text{"}||k^v||\ell{\uparrow}||\ell{\downarrow}) + k^s$ and $k^{v,(\ell\uparrow,\ell\downarrow)} = k^v k^{s,(\ell\uparrow,\ell\downarrow)}$.

The *one-time address* K^o_t is a unique output public key that the sender generates for each transaction output instead of using the recipient address directly. Let's define recipient keys (k^v_B, K^v_B), (k^s_B, K^s_B), sender knows the public parts (K^v_B, K^s_B). Sender uses transaction private key r to generate one-time public key $K^o_t = \mathcal{H}(rK^v_B||t)G + K^s_B$, where t is the $\mathcal{T}^{\text{out}}$ index. The $R = rG$ is stored to the transaction data. The recipient can then compute the spending private key $k^o_t = \mathcal{H}(k^v_B rG||t) + k^s_B$.

When sending to a sub-address, the $rK^{s,(\ell\uparrow,\ell\downarrow)}_B$ needs to be added to a transaction data $\boldsymbol{R^a}$, so the receiving and spending works in the same way as with the normal addresses. Note that $k^v_B rK^{s,(\ell\uparrow,\ell\downarrow)}_B = rK^{v,(\ell\uparrow,\ell\downarrow)}_B$, thus the one-time keys are $K^o = \mathcal{H}_s(rK^{v,(\ell\uparrow,\ell\downarrow)}_B||t)G + K^{s,(\ell\uparrow,\ell\downarrow)}_B$ and the corresponding spending private key $k^o = \mathcal{H}_s(rK^{v,(\ell\uparrow,\ell\downarrow)}_B||t) + k^{s,(\ell\uparrow,\ell\downarrow)}_B = \mathcal{H}_s(k^v rK^{s,(\ell\uparrow,\ell\downarrow)}_B||t) + k^{s,(\ell\uparrow,\ell\downarrow)}_B$. All $\mathcal{T}^{\text{in}}$ being spent in the transaction being generated have to share the same major sub-address index $\ell{\uparrow}$ by design. The $\ell{\uparrow}$ is specified in the message of the step 1. We thus omit the $\ell{\uparrow}$ from the equations to simplify the notation.

The HW then computes an ECDH *derivation*: $d_i = k^v R$, if $(\ell{\uparrow} = 0 \wedge \ell{\downarrow}_i = 0)$, or $d_i = k^v R^a_t$ otherwise. The HW verifies that $K^{s,\ell\downarrow_i} = K^o_i - \mathcal{H}_s(d_i||t)G$. If the equation holds it means the $\mathcal{T}^{\text{in}}_{i,\pi}$ has been sent to the $K^{s,\ell\downarrow_i}$ and we can generate spending private key $k^o_i = \mathcal{H}(d_i||t) + k^{s,(\ell\uparrow,\ell\downarrow_i)}$ and the *key image* $\ddot{K}^o_i = k^o_i \mathcal{H}_p(k^o_i G)$.

The HW then generates a new commitment mask $\alpha_i \xleftarrow{\$} \mathbb{Z}^*_l$ and a *pseudo-output commitment* $C'_i = \alpha_i G + v^{\text{in}}_i H$. The pseudo-output is used later in the MLSAG scheme. The HW also generates a $\text{vin}_i : \{\text{indices}, \ddot{K}^o_i\}$, a blockchain-encoded message which contains blockchain indices for the ring keys and the key image. The $\{\text{vin}_i, C'_i\}$ go to the final transaction, thus they are offloaded in clear-text with HMAC, on the other hand, the $\{\alpha_i,\, k^o_i\}$ are sensitive thus offloaded encrypted. The offloading keys are domain separated and contain the $\mathcal{T}^{\text{in}}$ index. The step 2 returns: $\{\widehat{\text{vin}_i}, \widehat{C'_i}, \widetilde{\alpha}_i, \widetilde{k}^o_i, \text{HMAC}(\mathcal{T}^{\text{in}}_i)\}$.

Step 3: The Monero consensus rules require UTXOs to be ordered by key-image in a transaction. As the vin_i messages are part of the transaction prefix hash $\mathfrak{m}_p$, the host sends $\widehat{\text{vin}_i}$ sorted by key-image, one by one to the HW in step 3, so the HW can update the hashing context $\mathcal{H}_{\mathfrak{m}_p}$. As HMAC keys were generated by the original ordering prior to sorting, the host needs to send also the original index o^i. The correctness of the permutation is verified by checking the set size and a string ordering on key images (no duplicates).

Step 4: processes transaction destinations $\mathcal{T}^{\text{out}}_i$ one by one and is described by the pseudo-code:

```
function SignStep4($\widehat{\mathcal{T}^{\text{out}}_i}$)    ▷ $\mathcal{T}^{\text{out}}_i = \{\text{amount}^o_i, K^v_i, K^s_i\}$
    Check HMAC on $\mathcal{T}^{\text{out}}_i$    ▷ to make sure destination was confirmed by the user in the step 1
    $r^a_i \xleftarrow{\$} \mathbb{Z}^*_l$;  $R^a_i \leftarrow (r^a_i K^s_i)$ if isSubaddr($\mathcal{T}^{\text{out}}_i$) else $(r^a_i G)$
    $d_i \leftarrow (aR)$ if isChange($\mathcal{T}^{\text{out}}_i$) else $((r^a_i K^v_i)$ if isSubaddr($\mathcal{T}^{\text{out}}_i$) else $(rK^v_i))$
    $K^o_i \leftarrow \mathcal{H}_s(d_i||i)G + K^s_i$    ▷ Derive output one-time address
    $k^a_i \leftarrow \mathcal{H}_s(d_i||i)$;  $\gamma^o_i \leftarrow \mathcal{H}_s(\texttt{"commitment\_mask"}||k^a_i)$    ▷ Derive amount key and C mask
    $\text{ECDH}_i \leftarrow \{\text{amount}_i \oplus \mathcal{H}(\texttt{"amount"}||k^a_i)_{[:8]}\}$    ▷ Encrypt amount in the ECDH structure
    $\text{vout}_i \leftarrow \{K^o_i\}$;  $C^o_i \leftarrow \gamma^o_i G + \text{amount}^o_i H$    ▷ Serialize destination, compute commitment
    Handle range proof generation for $(\text{amount}^o_i, \gamma_i)$    ▷ Explained in the section 3
```

```
        $\mathcal{H}_{\mathfrak{m}_p}(\text{vout}_i)$; $\mathcal{H}^{\text{ecdh}}_{\mathfrak{m}}(\text{ECDH}_i)$; $\mathcal{H}^{\text{range}}_{\mathfrak{m}}(\text{rangeProofData}_i)$;        ▷ Incremental hashing
        return $\{\widehat{\text{vout}_i}, \text{rangeProofData}_i, C^o_i, \text{ECDH}_i\}$
```

Step 5: The *extra* field containing additional transaction information is serialized from the elements $\{R, \boldsymbol{R^a}\}$. The incremental hashing is finished as all required information has been computed. The structure of the message hashes is the following: $\mathfrak{m}_p = \mathcal{H}(\text{version, unlock\_time}, \{\text{vin}_i\}_i, \{\text{vout}_i\}_i, \text{extra})$, then the pre-MLSAG $\mathfrak{m} = \mathcal{H}(\mathfrak{m}_p \ || \ \mathcal{H}(\{\text{ECDH}_i\}_i \ || \ \{C^o_i\}_i) \ || \ \mathcal{H}(\{\text{rangeProofData}_i\}_i))$.

MLSAG: A separate MLSAG is generated per $\mathcal{T}^{\text{in}}_j$ message. The host then sends $\{\widehat{\mathcal{T}^{\text{in}}_j}, \widehat{C'_j}, \widetilde{\alpha_j}, \widetilde{k^o_j}\}$, one by one. The HW checks HMACs on received values and decrypts commitment mask α_j and the spending private key k^o_j.

Mask balancing: The following equation has to hold to prove that the transaction input amounts equal to the transaction outputs plus transaction fee: $\sum_j C'_j - \sum_t C^o_t - \text{fee}H = 0$. For that to hold, the commitment masks have to balance on each side, i.e., $\sum_j \alpha_j = \sum_t \gamma^o_t$. As the γ^o_t are generated deterministically in the step 5, the balancing has to be done on a pseudo-output commitment mask α_j. The α_j are generated randomly in the step 2 so to make the equation hold the last mask is recomputed in this step: $\alpha_m = \sum_t \gamma^o_t - \sum_j^{m-1} \alpha_j$.

The ring is then constructed as $\mathcal{R}_j = \{\{K^o_{j,i}, (C^{\text{in}}_{j,i} - C'_{j,\pi})\}_{i\in[1,n]}\}$. The signer can sign the ring in the MLSAG as he knows the private key for public keys at index π, namely k^o_j for $K^o_{j,\pi}$ and $z_j = (\gamma^{\text{in}}_\pi - \alpha_\pi)$ for $(C^{\text{in}}_{j,\pi} - C'_{j,\pi}) = (\gamma^{\text{in}}_\pi G + v^{\text{in}}_j H - \alpha_\pi G - v^{\text{in}}_j H)$. The *MLSAG* is then generated in the following way:

Algorithm 1 MLSAG generator

```
function MLSAG($\mathfrak{m}, \mathcal{R}, \pi, k^o, z$)
    $\ddot{K} \leftarrow k^o \mathcal{H}_p(K_\pi)$        ▷ Compute the key image
    $\alpha_1 \xleftarrow{\$} \mathbb{Z}^*_l$; $\alpha_2 \xleftarrow{\$} \mathbb{Z}^*_l$; $r_{i,j} \xleftarrow{\$} (\mathbb{Z}^*_l)^{2i}$        ▷ Generate random masks
    $c_{\pi+1} \leftarrow \mathcal{H}_s(\mathfrak{m} \ || \ (K^o_\pi \ || \ \alpha_1 G \ || \ \alpha_1 \mathcal{H}_p(K^o_\pi)) \ || \ (C^{\text{in}}_\pi - C'_\pi \ || \ \alpha_2 G))$
    for $i \in (\pi+1, \pi+2, \ldots, n, 1, 2, \ldots, \pi-1)$ do
        $c_{i+1} \leftarrow \mathcal{H}_s(\mathfrak{m} \ || \ (K^o_i \ || \ r_{i,1}G + c_i K^o_i \ || \ r_{i,1}\mathcal{H}_p(K^o_i) + c_i\ddot{K}) \ || \ (C^{\text{in}}_i - C'_\pi \ || \ r_{i,2}G + c_i(C^{\text{in}}_i - C'_\pi)))$
    $r_{\pi,1} \leftarrow \alpha_1 - c_\pi k^o$; $r_{\pi,2} \leftarrow \alpha_2 - c_\pi z$        ▷ Redefine $r_\pi$ as we know the privates for $K_\pi$
    return $\{\ddot{K}, c_1, r_{1,1}, r_{1,2}, \ldots, r_{m,2}\}$
```

Step 5 returns $\{\widetilde{\text{MLSAG}_t}, C^o_t\}$, i.e., the encrypted generated signature and recomputed commitment. The *verifier* then starts reconstructing commitments from c'_2 and checks if the final c'_1 matches the value from the signature. Thus $c'_{i+1} = \mathcal{H}_s(\mathfrak{m} \ || \ (K_{i,1} \ || \ r_{i,1}G + c_i K_{i,1} \ || \ r_{i,1}\mathcal{H}_p(K_{i,1}) + c_i\ddot{K}) \ || \ (K_{i,2} \ || \ r_{i,2}G + c_i K_{i,2}))$. The verification works because if $i \neq \pi$, the c'_{i+1} are generated in the same way as in the signer algorithm. If $i = \pi$ then it holds that $r_{\pi,j}G + c_\pi K_{\pi,j} = (\alpha_j - c_\pi k_j)G + c_\pi K_{\pi,j} = \alpha_j G$ and similarly $r_{\pi,1}\mathcal{H}_p(K^o_\pi) + c_\pi\ddot{K} = (\alpha_1 - c_\pi k^o)\mathcal{H}_p(K^o_\pi) + c_\pi\ddot{K} = \alpha_1\mathcal{H}_p(K^o_\pi) - c_\pi(k^o\mathcal{H}_p(K^o_\pi)) + c_\pi\ddot{K} = \alpha_1\mathcal{H}_p(K^o_\pi)$.

Extension: Currently, the MLSAG is generated in memory in one message step as the only memory-limiting factor is the size of the ring $n = |\mathcal{R}|$, that is fixed to 11 in the current Monero version, and it can fit in memory. However, the generation algorithm can be easily extended to generate MLSAG in chunks in order to support large ring sizes. Note that each iteration requires only the current ring row $\mathcal{R}_i$, the $r_{i,j}$ masks and c_i for a fixed i.

Finalization: Once all MLSAGs are generated, the HW returns a decryption keys for $\widetilde{\text{MLSAG}_t}$. The HW also generates transaction-dependent encryption key $k^t = KDF(salt \xleftarrow{\$} \mathbb{Z}^*_l \ || \ k^s \ || \mathfrak{m}_p)$ and returns encrypted transaction private keys $\text{Enc}(k^t, r, \boldsymbol{r^a})$ so other Monero features requiring the keys are supported, such as generating a ZK proof that transaction has been sent.

Analysis: The protocol is based on cold-signing protocol 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 hardware wallet, 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 (Step 1), 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 $(\mathcal{T}^{\text{in}}, \mathcal{T}^{\text{out}})$.

Moreover, final signatures can be decrypted after the whole protocol finishes successfully, as the decryption keys are sent as the last message. 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 $\mathcal{T}_j^{\text{in}}$. 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.

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 *state* maintained by the protocol contains: $\{k^v$, k^s, k^{enc}, k^{mac}, r, n, m, p, $\ell\uparrow$, fee, $\mathcal{T}^{\text{out}}_{\text{change}}$, $\boldsymbol{r^a}$, $\boldsymbol{R^a}$, extra, v_p^o, $\sum v_t^o$, $\sum v_j^{\text{in}}$, $\sum \alpha_j$, $\sum \gamma_t$, $\boldsymbol{C'}$, $\boldsymbol{v^o}, \boldsymbol{\gamma^o}$, current element index, automaton state, $\mathcal{H}_{\mathfrak{m}_p}$, $\mathcal{H}_{\mathfrak{m}} : \{\mathcal{H}_{\mathfrak{m}}^{\text{main}}, \mathcal{H}_{\mathfrak{m}}^{\text{ECDH}}, \mathcal{H}_{\mathfrak{m}}^{\text{range}}\}$, $\mathfrak{m}_p, \mathfrak{m}\}$. However, state entries do not exist all in the same time in the memory and entries are freed once they are not needed anymore, e.g, $\{\boldsymbol{r^a}$, $\boldsymbol{R^a}$, extra, $\boldsymbol{C'}$, $\mathcal{H}_{\mathfrak{m}_p}$, $\mathcal{H}_{\mathfrak{m}}\}$ can be deleted after step 4 as they are needed only for the hashing. Similarly, the $\{\mathcal{T}^{\text{out}}_{\text{change}}$, $\boldsymbol{v^o}, \boldsymbol{\gamma^o}\}$ can be deleted after all outputs have been processed (and range proofs were generated).

Performance: The transaction signing protocol implemented in Micropython for Trezor HW was tested with various input transactions. Refer to Table 1 for performance overview data.

3 Range proof

The 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. Range proofs do not contain any wallet secrets, so the offloading optimizes the signing protocol, making it feasible to run on resource-constrained devices.

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 and the discrete logarithm problem, it is infeasible to find a different v', γ', s.t. $\gamma' G + v' H = \gamma G + vH$.

Table 1. Performance of the transaction signing protocol on Trezor HW. The algorithm was tested on emulator and Trezor T HW. Configuration is a tuple (#inputs, #outputs, #ring size). By default, the ring size is 11 and the third element is omitted. he 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	2-2-24	2-2-48	16-2	32-2	64-2	128-2	2-16	16-16
Time Emu [s]	9.31	7.68	21.80	29.93	58.13	106.32	209.88	21.45	42.21
Time [s]	16.90	21.45	30.82	83.22	156.82	306.74	604.09	46.13	118.42
$\sum$ Steps	12.49	14.63	19.59	56.30	106.69	207.33	408.42	36.62	81.05
Rounds	14	14	14	56	104	200	392	28	70
RAM [B]	41 264	70 768	79 328	42 176	42 208	42 048	58 512	41 376	42 464
State min [B]	2 385	2 385	2 385	2 385	2 385	2 385	2 385	4 406	4 406
State real [B]	5 315	5 315	5 315	5 315	5 315	5 315	5 315	9 224	9 224

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* [Bun+18] (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 $M = 2^x$ statements. The BP proof has the following structure: points/scalars $A, S, T_1, T_2, \tau_x, \mu, a, b, t$; a vector of Pedersen commitments $\boldsymbol{V}$ of size M, vectors $\boldsymbol{L}, \boldsymbol{R}$ of points of size $6log(M)$. In order to understand the implementation and the offloading protocol for the Bulletproofs, the Algorithm 2 describes the in-memory prover. We implemented the memory-optimized version for the Trezor HW. As the whole computation is in-memory, the algorithm is secure in the malicious attacker model.

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. Vectors like $\boldsymbol{\zeta}, \boldsymbol{r_0}, \boldsymbol{r_1}$ are evaluated in a constant time with sequential access to the elements, i.e., with internal state to x_i the x_{i+1} can be evaluated in $O(1)$. The size of vectors is reduced by 2 with each iteration so overall memory consumption is determined by the first iteration. The values c_L, c_R, L_c, R_c, w can be computed in a constant memory with on-the-fly vector evaluation. The vectors $\boldsymbol{G'}, \boldsymbol{H'}, \boldsymbol{a'}, \boldsymbol{b'}$ folding on lines 36-39 require to store the results to the memory as on-the-fly evaluation of the new vectors would be infeasible due to large dependency complexity (mainly L_c, R_c). Thus the space complexity of the whole algorithm is $O(4\frac{MN}{2})$. This leaves a space for minor optimization: with the $O(2MN)$ memory we can memorize the $\boldsymbol{l}, \boldsymbol{r}$ vectors before the while-loop, compute the loop up to line 37 with $O(2MN)$ memory, but more effectively due to reduced on-the-fly evaluation costs. Before line 38, the memory consumption is $O(MN)$, after the line 39 it is $O(2MN)$.

Implementation: We implemented the in-memory Bulletproof prover as specified in the Algorithm 2 and the verifier in Micropython and tested on the Trezor HW. The size of the generated proof is $32(9 + M + 2log(NM)) = 672 + 32M + 64log(M)$, for $N = 64$ bits of amount and M statements. The memory usage is $32(128M + 12log(M)) + O(max(lg(N), M))$ B. The verification algorithm runs with $O(max(lg(N), M))$ memory. Table 2 shows the performance of the prover and the verifier implemented on the Trezor. 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.

[4] 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 2 Bulletproof prover. $N = 64, M = |\boldsymbol{v}|, \boldsymbol{v}$ is a vector of amount scalars

```
1  function BulletproofProver($\boldsymbol{v}, \boldsymbol{\gamma}$)    ▷ Amounts and masks are input parameters
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 constant memory up to this point.
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($\boldsymbol{v}, \boldsymbol{\gamma}$)    ▷ Amounts and masks are input parameters
9      $\mathring{G}_{i \in [0,\ldots,MN)} \leftarrow \mathcal{H}_p(\mathcal{H}(\text{"bulletproof"}||H||\text{varint}(2i+1)))$    ▷ varint encodes integer to bytes
10     $\mathring{H}_{i \in [0,\ldots,MN)} \leftarrow \mathcal{H}_p(\mathcal{H}(\text{"bulletproof"}||H||\text{varint}(2i)))$
11     $V_j \leftarrow \gamma_j G + v_j H$    ▷ Compute commitment vector $\boldsymbol{V}$ used on line 18

12     $\alpha \xleftarrow{\$} \mathbb{Z}_l^*$
13     $\boldsymbol{a_L} \leftarrow \text{expand}(\boldsymbol{v})$; $\boldsymbol{a_R} = \boldsymbol{a_L} - \boldsymbol{1}^{MN}$    ▷ expand($\boldsymbol{v}$)=$\boldsymbol{x}$: $\sum_{i=0}^{63} 2^i x_{64j+i} = v_j, j \in [0, |v|), x_i \in \{0, 1\}$
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[4]

15     $\rho, R \xleftarrow{\$} (\mathbb{Z}_l^*)^2$
16     $\boldsymbol{s_L} \leftarrow \text{randVct}(0), \boldsymbol{s_R} \leftarrow \text{randVct}(1)$    ▷ randVct($j$) $= \boldsymbol{x} : |\boldsymbol{x}| = MN, x_i = H_s(\text{"mask"}||R||i||j)$
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 and masks
19     $\boldsymbol{l_0} \leftarrow \boldsymbol{a_L} - z\boldsymbol{1}^{MN}$; $\boldsymbol{l_1} \leftarrow \boldsymbol{s_L}$    ▷ Vectors evaluated with constant memory, with $\boldsymbol{v}$
20     $\zeta_i \leftarrow z^{2+\lfloor i/N \rfloor} 2^{i\%N}$    ▷ Evaluated with constant memory and time with sequential access
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         $c_L \leftarrow a'_{[0,\ldots,n')} \cdot b'_{[n',\ldots,\bar{n})}$
32         $c_R \leftarrow a'_{[n',\ldots,\bar{n})} \cdot b'_{[0,\ldots,n')}$
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         $a'_{i \in [0,\ldots,n')} \leftarrow w \quad a'_i + w^{-1} a'_{i+n'}$    ▷ Scalar vector folding, reduces vector size by 2
37         $b'_{i \in [0,\ldots,n')} \leftarrow w^{-1} b'_i + w \quad b'_{i+n'}$
38         $G'_{i \in [0,\ldots,n')} \leftarrow w^{-1} G'_i + w \quad G'_{i+n'}$    ▷ Hadamard product (folding), reduces vector size by 2
39         $H'_{i \in [0,\ldots,n')} \leftarrow w \quad H'_i + w^{-1} H'_{i+n'}$

40     return $(\boldsymbol{L}, \boldsymbol{R}, a'_0, b'_0)$    ▷ vector $\boldsymbol{L}$ composed from $L_c$
```

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.

Table 2. BP performance on the Trezor T hardware wallet. Verifier is faster than prover and requires significantly lower memory. Prover time and space complexity increases linearly to the input size.

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

3.1 Offloaded Bulletproofs

Due to rising 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 protocol (MPC) to compute Bulletproofs jointly with the PC host and a hardware wallet with a constant memory on the hardware wallet side. We do not consider the time and memory requirements of the protocol running on the host in the following text.

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 can be computed incrementally as $t = \sum l_i r_i$. We need only sequential access to the vector elements thus the chunking method works well. 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.

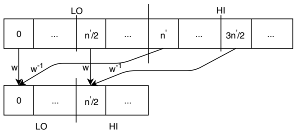
As the scalars are from $\mathbb{Z}_l^*$ and vector elements are essentially random, the attacker cannot infer $\boldsymbol{a'}$ from $\pi_{a'}\boldsymbol{a'}$. The $\pi_{a'}\boldsymbol{a'}_0 = z$ does not have a unique factorization, i.e., $\forall z, x\ \exists y : xy = z;\ y = zx^{-1}$. Thus, a blinded 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 $a'_{i \in (0,\ldots,n']} = wa'_i + w^{-1}a'_{i+n'}$, 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 Figure 3.1 and 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:



Vector folding

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

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 $\pi_{G'_{LO}}$.

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

$\boldsymbol{L_c}, \boldsymbol{R_c}$ offloading: Observe that L_c from line 33 can be computed from 3 independent components: $L_c = 8^{-1}\left(\left(\left(\sum_{i=0}^{n'} a'_i G'_{i+n'}\right) + \left(\sum_{i=0}^{n'} b'_{i+n'} H'_i\right)\right) + \left(c_L x'\right) H\right)$. 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.

Implementation: The offloaded Bulletproof prover as defined in Algorithm 3 was implemented in the Micropython for the Trezor and performance of the implementation was evaluated. Table 3 shows a total time and memory consumption for the emulator (running the same software, but with emulated CPU, running on the desktop) and the HW. The table also contains performance statistics for previous in-memory implementation for comparison. The HW part also contains a maximal state size during the computation and 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.

Algorithm 3 Bulletproof prover with offloading. b_{tch} is a number of elements to offload in one batch, n_{thr} is a n' threshold when to finish BP computation in-memory.

```
1  function BulletproofProverOffloaded($\boldsymbol{v}, \boldsymbol{\gamma}$)
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      $t \leftarrow 0$; $c \leftarrow -1$; $n' \leftarrow MN$; $\boldsymbol{\pi} \xleftarrow{\$} (\mathbb{Z}_l^*)^8$        ▷ New random blinding masks $\boldsymbol{\pi}$
4      for $i \in \left[0, \dots, \frac{MN}{b_{tch}}\right)$ do        ▷ Compute $t$, export blinded $\boldsymbol{l}, \boldsymbol{r}$ vectors
5          $\boldsymbol{l_c} \leftarrow l_{0,j} + xl_{1,j}$; $\boldsymbol{r_c} \leftarrow r_{0,j} + xr_{1,j},\ j \in [ib_{tch}, (i+1)b_{tch})$
6          $\bar{\boldsymbol{l_c}} \leftarrow \pi_{a'_{\delta(i)}} \boldsymbol{l_c}$; $\overline{\boldsymbol{r_c}} \leftarrow \pi_{b'_{\delta(i)}} \boldsymbol{r_c}$; $t \leftarrow t + \boldsymbol{l_c} \cdot \boldsymbol{r_c}$        ▷ $\delta(x) = x < n'$ ? LO : HI, $\bar{\boldsymbol{l_c}}$ means blinded $\boldsymbol{l_c}$
7          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
8      $w \leftarrow x' \leftarrow \mathcal{H}_s(x||x||\tau_x||\mu||t)$        ▷ $t = l \cdot r$
9      Send($y$)        ▷ The host needs $y$ to compute $\boldsymbol{H'}$ and $L_c, R_c$ related sums
10     while $n' > 1$ do        ▷ The $y$ is public, computable from the final proof
11         $\bar{n} \leftarrow n'$; $n' \leftarrow n'/2$; $c \leftarrow c + 1$
12         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 $L_c$
13         $L_c \leftarrow 8^{-1}\left(\pi^{-1}_{a'_{LO}} \pi^{-1}_{G'_{HI}} \overline{L_{cA}} + \pi^{-1}_{b'_{HI}} \pi^{-1}_{H'_{LO}} \overline{L_{cB}} + x'\pi^{-1}_{a'_{LO}} \pi^{-1}_{b'_{HI}} \overline{c_L} H\right)$
14         $R_c \leftarrow 8^{-1}\left(\pi^{-1}_{a'_{HI}} \pi^{-1}_{G'_{LO}} \overline{R_{cA}} + \pi^{-1}_{b'_{LO}} \pi^{-1}_{H'_{HI}} \overline{R_{cB}} + x'\pi^{-1}_{a'_{HI}} \pi^{-1}_{b'_{LO}} \overline{c_R} H\right)$
15         $w \leftarrow \mathcal{H}(w||L_c||R_c)$; $\boldsymbol{\theta} \xleftarrow{\$} (\mathbb{Z}_l^*)^8$        ▷ Compute $w$, generate blindings $\boldsymbol{\theta}$ for the next round
16         if $n' \leq n_{thr}$ then        ▷ Finish in-memory with original algorithm
17             Receive$\left(\bar{\boldsymbol{a}}', \bar{\boldsymbol{b}}', \overline{\boldsymbol{G}}', \overline{\boldsymbol{H}}'\right)$; Unblind to obtain $\{\boldsymbol{a'}, \boldsymbol{b'}, \boldsymbol{G'}, \boldsymbol{H'}\}$
18             $(\boldsymbol{L}, \boldsymbol{R}, a'_0, b'_0) \leftarrow$ BulletproofLoop($\boldsymbol{a'}, \boldsymbol{b'}, \boldsymbol{G'}, \boldsymbol{H'}, n', c, w, x'$)        ▷ Combine $\boldsymbol{L}, \boldsymbol{R}$
19             return $(\boldsymbol{V}, A, S, T1, T2, \tau_x, \mu, \boldsymbol{L}, \boldsymbol{R}, a'_0, b'_0, t)$
20         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 $\phi^{(c)}_{x,l}$ for host folding
21         if $c = 0$ then
22             Compute and send $\boldsymbol{G'}, \boldsymbol{H'}$ folding correction points as defined in paragraph 3.1, by chunks
23         $\boldsymbol{\pi} \leftarrow \boldsymbol{\theta}$        ▷ Update blinding masks for the next round
```

Attacker constraints: As we transformed the Algorithm 2 to the Algorithm 3 using offloading steps that preserve the privacy of the computation, the protocol remains secure in the honest-but-curious attacker model. The malicious attacker deviating from the protocol 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 cryptographic hash function on line 15. As a consequence, the network nodes reject the transaction due to the invalid proof, and the attack will be noticed by the user. Another precaution against malicious attackers is to run a Bulletproof verifier on the generated proofs and 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.

4 Related work

The work [Mes19] presents a signature protocol for a Ledger[5] 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

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

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.

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 size [B]	3 603	5 747	5 875	6 067	6 387
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
Messages	12	16	25	42	59
Sent [B]	6 528	8 576	8 768	8 960	9 152
Received [B]	7 395	13 635	26 563	51 843	1 018 27

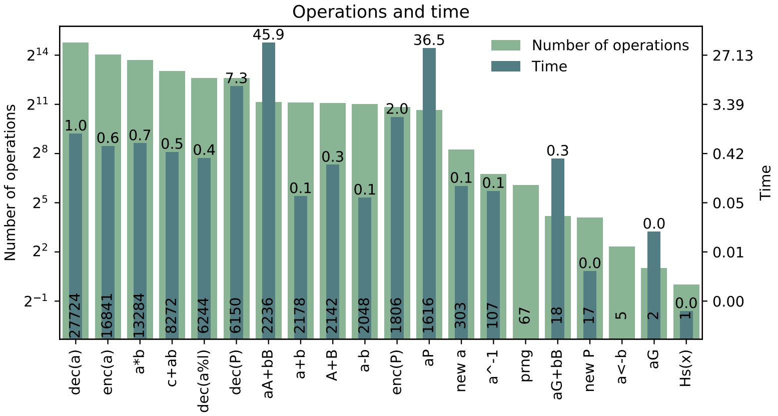


Figure 1. Privacy-preserving multi-party BP with $M = 16$ cryptographic operations and timing breakdown. It is evident the most expensive operations are aP and point recodings.

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

The protocol is tightly integrated into the Monero codebase. The tight coupling imposes maintainability challenges as a Monero algorithm change usually requires a HW signing protocol change. Technically, the Monero client implements a virtual device. If Ledger is used, the low-level cryptographic operations are computed in the device. 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. To the best of our knowledge, the overall protocol is not documented nor analyzed. The low-level commands are documented in [Mes19] 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, has very simple interface and thus reduced attack surface. 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 hardware wallets. We designed and analyzed the memory and time complexity of the zero-knowledge proofs (range-proofs, Bulletproofs [Bun+18]), 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.

References

[Yao82] Andrew C. Yao. “Protocols for secure computations.” In: *23rd Annual Symposium on Foundations of Computer Science*. IEEE, 1982, pp. 160–164. DOI: `10.1109/SFCS.1982.88`.

[Ped92] Torben P. Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing.” In: *Advances in Cryptology — CRYPTO ’91*. Springer Berlin Heidelberg, 1992, pp. 129–140. ISBN: 978-3-540-46766-3.

[Ber+12] Daniel J. Bernstein et al. “High-speed high-security signatures.” In: *Journal of Cryptographic Engineering* 2.2 (Sept. 2012), pp. 77–89. ISSN: 2190-8516. DOI: `10.1007/s13389-012-0027-1`.

[Noe15] Shen Noether. *Ring Signature Confidential Transactions for Monero*. Cryptology ePrint Archive, Report 2015/1098. `https://eprint.iacr.org/2015/1098`. 2015.

[Mes17] Cédric Mesnil. *Ledger device for Monero, v0.8*. GitHub/LedgerHQ/blue-app-monero. https://github.com/LedgerHQ/blue-app-monero/blob/master/doc/developer/blue-app-monero.pdf. 2017.

[You17] Joseph Young. *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. Accessed: 26. Feb 2020. 2017.

[Alo18] Kurt M. Alonso. *Zero to Monero: First Edition*. https://www.getmonero.org/library/Zero-to-Monero-1-0-0.pdf. Accessed: 20. Feb 2020. 2018.

[Bun+18] Benedikt Bunz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More." In: May 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020.

[Kli18] Dusan Klinec. *Monero wallet Trezor integration*. https://github.com/ph4r05/monero-trezor-doc. Accessed: 26. Feb 2020. 2018.

[Goo19] Dan Goodin. *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. Accessed: 26. Feb 2020. 2019.

[Lab19] Satoshi Labs. *Trezor Model T*. https://wiki.trezor.io/Trezor_Model_T. Accessed: 26. Feb 2020. 2019.

[Mes19] Cédric Mesnil. *Ledger device for Monero*. online. https://github.com/LedgerHQ/ledger-app-monero, Accessed: 20. Feb 2020. 2019.

[Cim20] Catalin Cimpanu. *Chrome extension caught stealing crypto-wallet private keys*. https://www.zdnet.com/article/chrome-extension-caught-stealing-crypto-wallet-private-keys/. Accessed: 26. Feb 2020. 2020.