Multisig2 Formal Verification Report

Prepared by Pruvendo at 09/02/22

Executive summary

The current report validates the smart contract called “Multisig2” (located here) and confirms no bugs that can lead to losing or freezing of any funds, as well as to its improper distribution were discovered. The claim above is based on mathematical proving of things rather than on manual inspection that minimizes the chance of missing the potential threats.

Thus, the verifier recommends the Multisig2 smart contract for moving into production.

Any questions or comments regarding the present report are to be requested by email info@pruvendo.com or by Telegram(SergeyEgorovSPb or andruiman).

Brief project description

The smart contract being verified is an Everscale cryptocurrency wallet that can require multiple signatures to make any payment. The list of eligible “custodians” as well as a number of signatures required can be altered exclusively by upgrading the full contract code that is allowed by the approval of the strong majority of custodians.

Intention of the current release and verification

While the original version of *Multisig* was released in 2020 and was fully formally verified, including elements of bytecode formal verification that is temporarily currently not available, one of the critical bugs in bytecode verification was discovered.

Bug description

The bug was related to that fact that the destination of the signed external message was not a part of the signature that made the following attack possible:

- There are two physical users: Alice and Bob
- There are two *Miltisig* wallets: Alpha and Beta, where Alice and Bob are the only custodians for both of them]
- Both the wallets require two signatures for sending any transaction
- Alice creates, at the same time, two payment requests:
    - To Alpha, that is totally acceptable for Bob
    - To Beta, that is unacceptable to Bob
- The both transactions have the same ID (with a significant probability)
- Bob signs the transaction for Alpha, the assets are sent
- Alice catches the transaction and put it into the queue for Beta
- As the destination was not signed, the contract considers the transaction as valid, signs it by Bob and sends the assets

Important notes

The bug described above is not a bug of the smart contract itself but rather a bug of the compiler. **It does not question the verification of 2020** where the full verification was provided for the sources of the smart contract itself only. The partial verification of the bytecode was implemented that time as well, but it was an experimental feature that proved to have limited capability and overcomplicated, so for now the verification of smart code is suspended indefinitely as an activity not ready for commercial usage.

Intention for the reverification

Some minor improvements were introduced into the code that automatically makes the code unreliable. While the modern *Pruvendo* technologies would require a minor verification efforts to revalidate the smart contract, the very old and obsolete 2020 techniques can not be repeated or reproduced. Even more, while the original 2020 verification took as much as ~25 man/months, the new reverification from scratch required ~1 man/month that effectively made the restoration of old technologies useless.

Methodology

Being different from other audit approaches where the smart contracts are validated by manual inspection, Pruvendo uses the formal methods where verification is performed by mathematical methods (roughly speaking, a code correctness is proved as a theorem). Some basic details are described in Appendix I while further information can be provided by request.

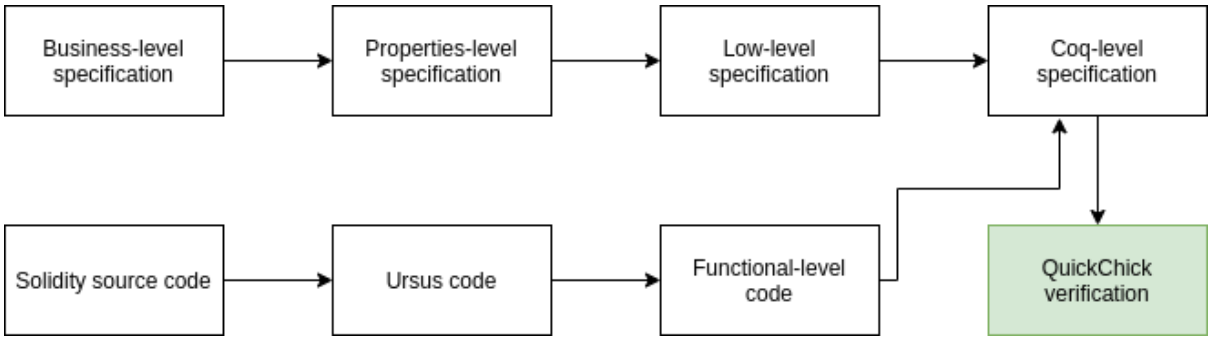
Briefly, the verification process can be splitted into the following phases:

- Specification - before any verification it's needed to state what it's planned to verify[1]. This part can be divided into the following sections:
    - Business-level specification (high-level description intended for the end-users of the smart contract system)
    - Property-level specification (description of all the properties of the system in a natural[2] language)
    - Intermediate-level specification (description of all properties in a language understandable by software developers[3])
    - Low-level specification (description of all properties with Coq[4]-specific predicates, intended for verifiers only). This activity is typically done when the next part - *Translation* - is completed
- Translation as conversion of the Solidity code into a set of functions with provable properties. The process has two stages:
    - Translation from Solidity into Coq-based DSL called *Ursus*[5]
    - Translation from the intermediate representation mentioned above into a set of functions
- Verification as supplying evidence of correctness of low-level specification towards the set of functions received above is conducted by QuickChick toolkit. This approach supplies less confidence than deductive software verification[6] but is considered as sufficient for the simple smart contract system being discussed.

The process can be described by the following diagram.

---

[1] Important to mention that the statement "The program works correctly" is meaningless because "correctness" fully depends on context. The correct statement is "The program works strictly according to the specification".
[2] For example, English.
[3] Specification of this intermediate language called FeLiz currently is available by request only. However, it follows Java-like notations understandable by most software developers.
[4] The framework used for verification. Some basic information about it is available in Appendix I .
[5] The specification is available by request.
[6] If needed, the complete deductive verification can be conducted upon a separate request.



The verification process is described in the following sections.

Specification

Business-level description

The purpose and basic properties

The multisig wallet serves two purposes. First, it increases the entrance security of a wallet, protects it from ill-tempered or unapproved actions of a custodiam. Also it serves as an additional protection in case one of the custodians is hacked.

The list of custodians and the number of required signatures is immutable unless the code is upgraded (see below).

The wallet can process several transactions simultaneously. A transaction can be deleted for two reasons: either it is confirmed and processed, or the time interval for confirmation has expired. There is no real-time and direct notification of a submitter about the expiration.

Each custodian can submit a transaction, but the predefined number of additional signatures (can be zero) are required to finish the operation. There is an upper limit for active transactions initiated by one custodian.

The code as well as all the parameters of the contract can be updated by the supermajority of the custodians.

Mechanics

If the custodian is the single one she can simply send a transaction to any recipient with some amount and payload.

For multiple custodians the mechanics is very straightforward as well: submit, confirm, send:

- Any custodians submits a transaction (having less than a predefined number of active transactions)
- Any other custodian can confirm a transaction
- If the required number of signatures is reached the transaction is automatically sent

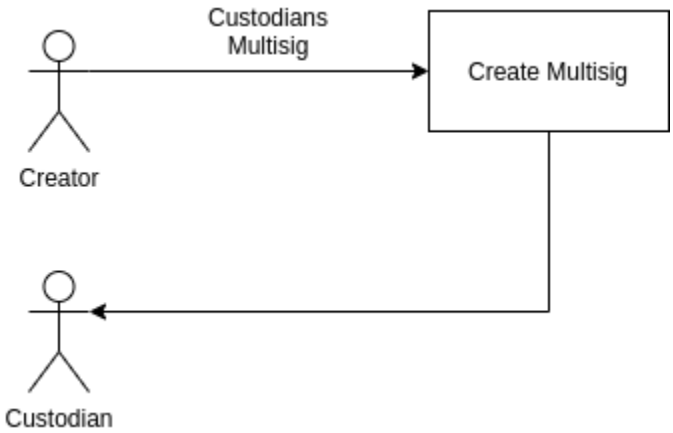
In case of code update the mechanics is rather similar:

- Any custodian can send the request to update the code
- Any other custodian can confirm the code update
- If the required number of signatures is reached any custodian can update the code while its hash equals to the hash code of the initial request

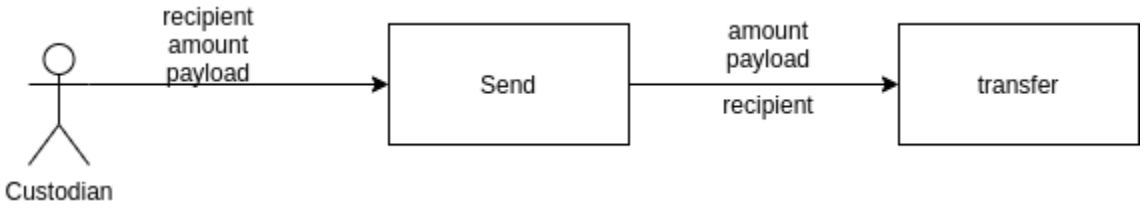
Scenarios

The following scenarios are in place.

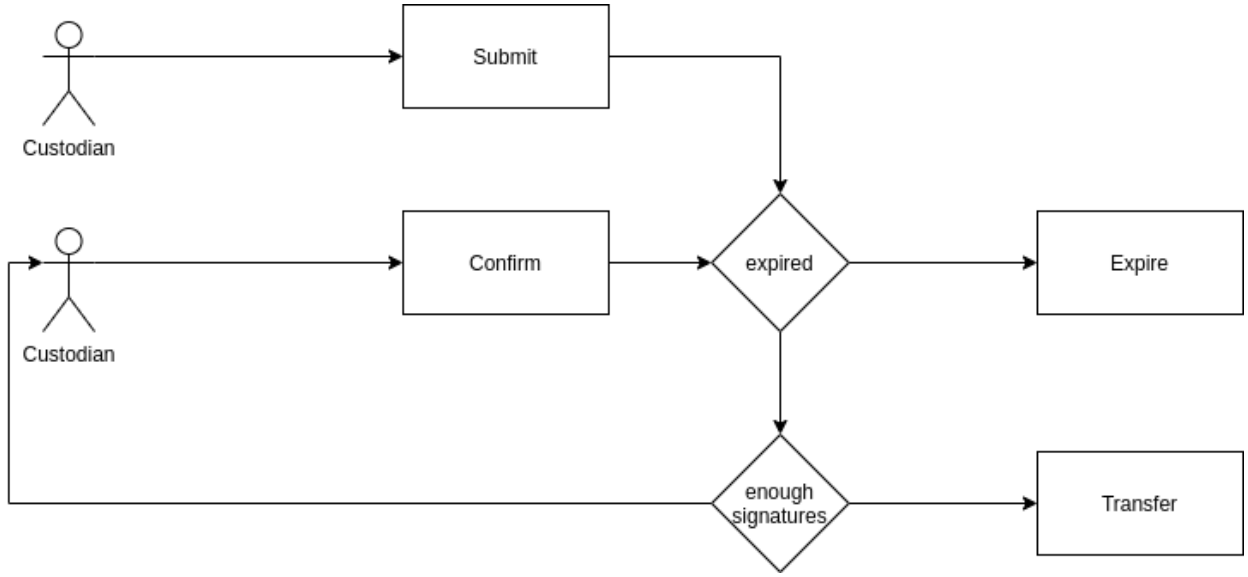
Create the wallet



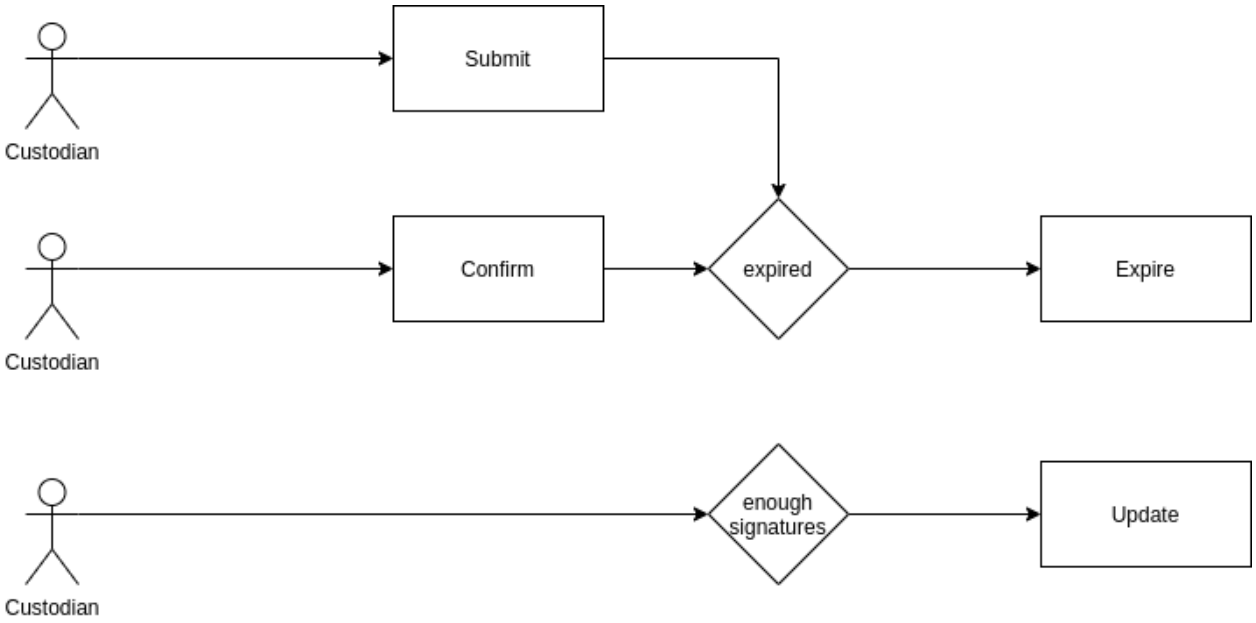
Send a transaction for a single custodian



Submit and send a transaction for multiple custodians



Update code


Custodian
Submit
Custodian
Confirm
expired
Expire
Custodian
enough
signatures
Update

Initialization features

No.	Natural language	FELIZ
INT.1	At least one *Custodian* must exist, otherwise Exception must be raised	∀ params : eval(constructor(params)) = ok() → params.owners.size > 0
INT.2	A number of *Custodians* should not exceed MAX_CUSTODIANS, otherwise Exception must be raised	∀ params : eval(constructor(params)) = ok() → params.owners.size ≤ MAX_CUSTODIANS
INT.3	*Custodians* are defined at the time of contract creation and can not be altered later (but code upgrade)	∀ f, params : f ∈ MiltisigFunctions → params.this ∈ Multisig → f ≠ constructor → f ≠ executeUpdate → (params.this.m_custodians = exec(f(params)).this.m_custodians ∧ params.this.m_custodianCount = exec(f(params)).this.m_custodianCount ∧ params.this.m_ownerKey = exec(f(params)).this.m_ownerKey)
INT.4	*A number of signatures* is defined at the time of contract creation, can not be altered later (but code upgrade) and calculated as minimum between a number of *Custodians* and *a number of signatures*	∀ params : eval(constructor(params)) = ok() → constructor(params).this.m_defaultRequiredConfirmations = min (params.owners.size, params.reqConfirms)
		∀ f, params : f ∈ MiltisigFunctions → params.this ∈ Multisig → f ≠ constructor → f ≠ executeUpdate → params.this.defaultRequiredConfirmations =

		`exec(f(params)).this.defaultRequiredConfirmations`
`INT.5`	Constructor can be called by *Creator* only	`∀ params : eval(constructor(params)) = ok() → params.sender.pubkey = params.this.creator.pubkey`
`INT.6`	In case of out of gas or any exception nothing changed	`∀ params : eval(constructor(params)) ≠ ok() → params.this = constructor(params).this`
`INT.7`	If all the following conditions are met, the contract must be created with *Custodians* and *A number of signatures* provided: ● At least one *Custodian* exists ● A number of *Custodians* not exceeds MAX_CUSTODIANS ● No out of gas Exception raised ● Constructor is called by *Creator*	`∀ params : params.this ∈ Multisig → params.owners.size > 0 → params.owners.size ≤ MAX_CUSTODIANS → eval(constructor(params) ≠ Exception("out of gas") → params.sender.pubkey = params.this.creator.pubkey → let result = constructor(params).this in (result.initialized = true ∧ result.m_custodians.size ≤ params.owners.size ∧ (∀ i : i ≥ 0 → i < result.m_custodians.size → (exists c : c In result.m_custodians.keys ∧ result.m_custodians[c] = Some(i))) ∧ (∀ i : i ≥ 0 → i < params.owners.size → (exists j : result.m_custodians[params.owners[i]] = Some(j))) ∧ (∀ c1, c2, v : → result.m_custodians[c1] = result.m_custodians[c2] → result.m_custodians[c1] = Some(v) → c1 = c2) ∧ result.m_defaultRequiredConfirmations = min (result.this.m_custodians.sizde, params.reqConfirms) ∧ result.m_ownerKey = params.this.owners[0] ∧ (∀ i : i ≥ 0 → i < 32 →`

		`result.m_requestsMask[i] = false) ∧ result.m_transactions = {} ∧ result.m_updateRequests = {} ∧ (∀ i : i ≥ 0 → i < 32 → result.m_updateRequestsMask[i] = false))`

Single custodian transaction sending

No.	Natural language	FELIZ
STS.1	If a number of *Custodians* is not equal to 1, the Single custodian transaction can not be sent (Exception must be raised)	`∀ params : eval(sendTransaction(params)) = ok() → params.this.m_custodians.size = 1`
STS.2	If the *sender* is not the only *Custodian*, the Exception must be raised	`∀ params : eval(sendTransaction(params)) = ok() → exists i : params.this.m_custodians[params.sender.pubkey] = Some(i)`
STS.3	If the sender is the only Custodian, the request to transfer the amount provided with the parameters provided must be placed with the additional `FLAG_IGNORE_ERRORS` flag	`∀ params : params.this.m_custodians.size = 1 → (exists i : params.this.m_custodians[params.sender.pubkey] = Some(i)) → eval(sendTransactions(params) ≠ Exception("out of gas") → (let transfer = sendTransaction(params) in eval(transfer) = ok() ∧ transfer.this = params.this[balance = transfer.this.balance] ∧ transfer.out.messages.size = 1 ∧ (let msg = transfer.out.messages.head in msg.name =`

		"transfer" ∧ msg.params.dest = params.dest ∧ msg.params.value = params.value ∧ msg.params.bounce = params.bounce ∧ msg.params.payload = params.payload ∧ FLAG_IGNORE_ERRORS In msg.params.flags ∧ (∀ f : (f In params.flags ∧ ~(f = FLAG_IGNORE_ERRORS)) ↔ (f In msg.params.flags ∧ ~(f = FLAG_IGNORE_ERRORS)))))

Multiple custodian transaction submission

Story: Expired transactions removal

No.	Natural language	FELIZ
ETR.1	Any transaction must be removed from the queue if and only if all the following conditions are met: ● The transaction was submitted before the EXPIRATION_TIME of seconds from NOW ● Less than MAX_CLEANUP_TXNS transactions submitted before the current one are in the queue	∀ params, transaction : transaction In params.this.m_transactions → (transaction.id >> 32) + EXPIRATION_TIME ≤ NOW → params.this.m_transactions.filter(t -> t.index = transaction.index ∧ t.createdAt < transaction.createdAt).count < MAX_CLEANUP_TXNS ↔ (transaction In params.this.m_transactions ∧ transaction Not In _removeExpiredTransactions(params).this.m_transactions)
	Amount of pending transactions for each	∀ f, params, i : f ∈ MiltisigFunctions ∨ f =

	custodian is updated accordingly	`_removeExpiredTransactions → params.this ∈ Multisig → f ≠ constructor → f ≠ executeUpdate → params.this.m_custodians[params.sender.pubkey] = Some(i) → params.this.m_transactions.filter(t -> t.index = i).fold("+", 0) = params.this.m_requestsMask[i] → f(params).this.m_transactions.filter(t -> t.index = i).fold("+", 0) = f(params).this.m_requestsMask[i]`

Main part

No.	Natural language	FELIZ
MTS.1	Only *Custodian* can submit a transaction, otherwise Exception must be raised	`∀ params : eval(submitTransaction(params)) = ok() → exists i : params.this.m_custodians[params.sender.pubkey] = Some(i)`
MTS.2	If the *sender* is a *Custodian* and there is enough gas all the expired transactions must be removed from the queue. Such an operation must be committed to ensure the state is changed even in case of further Exception	`∀ params, i : params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(submitTransactions(params) ≠ Exception("out of gas") → eval(submitTransaction(params)) = Exception() → exec(submitTransaction(params)[∀ transaction : ETR1]`
MTS.3	If the *sender* is a *Custodian* and there is enough	`∀ params, i, requestMask:`

	gas then, after the removal the expired transactions, the number of pending transactions related to this Custodian should not exceed `MAX_QUEUED_REQUESTS`, otherwise Exception must be raised	`eval(submitTransaction(params)) = ok() → params.this.m_custodians[params.sender.pubkey] = Some(i) → m_requestsMask = ETR1 → m_requestMask[i] < MAX_QUEUED_REQUESTS`
`MTS.4`	If: ● *sender* is a *Custodian*, and ● there is enough gas and ● after the removal the expired transactions, the number of pending transactions does not exceed `MAX_CLEANUP_TXNS`, and ● *A number of signatures* is less than 2 ● the *allBalance* input parameter is TRUE then the request to transfer the all the amount with the parameters provided must be placed with the additional `FLAG_IGNORE_ERRORS` flag	`∀ params, transactions : exists i : params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(submitTransactions(params) ≠ Exception("out of gas") → transactions, requestsMask = ETR1 → requestMask[i] < MAX_QUEUED_REQUESTS → params.m_defaultRequiredConfirmations < 2 → params.allBalance = true → (let transfer = submitTransaction(params) in eval(transfer) = ok() ∧ transfer.this = params.this[balance = transfer.this.balance, m_transactions = transactions] ∧ transfer.out.messages.size = 1 ∧ (let msg = transfer.out.messages.head in msg.name = "transfer" ∧ msg.params.dest = params.dest ∧ msg.params.bounce = params.bounce ∧ msg.params.payload = params.payload ∧ msg.params.value = 0 ∧ msg.params.flags = FLAG_IGNORE_ERRORS | FLAG_SEND_ALL_REMAINING))`
`MTS.5`	If: ● *sender* is a *Custodian*, and ● there is enough gas and ● after the removal the expired transactions, the number of pending transactions does	`∀ params, transactions : exists i : params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(submitTransactions(params) ≠ Exception("out of gas") → transactions, requestMask = ETR1 → requestMask[i] <`

	not exceed `MAX_CLEANUP_TXNS`, and ● *A number of signatures* is less than 2 ● the *allBalance* input parameter is FALSE then the request to transfer the the amount provided with the parameters provided must be placed with the additional `FLAG_IGNORE_ERRORS` flag	`MAX_QUEUED_REQUESTS → params.m_defaultRequiredConfirmations < 2 → params.allBalance = false → (let transfer = submitTransaction(params) in eval(transfer) = ok() ∧ transfer.this = params.this[balance = transfer.this.balance, m_transactions = transactions] ∧ transfer.out.messages.size = 1 ∧ (let msg = transfer.out.messages.head in msg.name = "transfer" ∧ msg.params.dest = params.dest ∧ msg.params.value = params.value ∧ msg.params.bounce = params.bounce ∧ msg.params.payload = params.payload ∧ msg.params.flags = FLAG_IGNORE_ERRORS | FLAG_PAY_FWD_FEE_FROM_BALANCE))`
`MTS.6`	If: ● *sender* is a *Custodian*, and ● there is enough gas and ● after the removal the expired transactions, the number of pending transactions does not exceed `MAX_CLEANUP_TXNS`, and ● *A number of signatures* is more than 1 then: ● The transaction with the following attributes must be put into the queue: ○ **Unique** identifier ○ Empty list of signers ○ *A number of signatures* ○ A number of received signatures (**must be synchronized with the list of signers**) ○ *sender*	`∀ f, params : f ∈ MiltisigFunctions → params.this ∈ Multisig → f ≠ constructor → f ≠ executeUpdate → (∀ t1, t2 : t1 ≠ t2 → t1 In params.this.m_transactions → t2 In params.this.m_transactions → t1.id ≠ t2.id) → (let result = f(params) in (∀ t1, t2 : t1 ≠ t2 → t1 In result.this.m_transactions → t2 In result.this.m_transactions → t1.id ≠ t2.id))`  `∀ f, params : f ∈ MiltisigFunctions → params.this ∈ Multisig → f ≠ constructor → f ≠ executeUpdate → (∀ t: t In params.this.m_transactions → t.signsReceived = t.confirmationMask.count) → (∀ t: t In f(params).this.m_transactions → t.signsReceived = t.confirmationMask.filter(m -> m).count)`

	○ *destination* ○ *value :* if *allBalance* ■ then, 0 ■ else, value provided ○ flags : `FLAG_IGNORE_ERRORS | allBalance ? FLAG_SEND_ALL_REMAINING : FLAG_PAY_FWD_FEE_FROM_BALANCE` ○ Payload as provided ○ Bounce as provided • The transaction must be signed by the sender that means: ○ The sender is added to the list of signers ○ A number of received signatures is increased	`∀ params, transactions : exists i : params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(submitTransactions(params) ≠ Exception("out of gas") → transactions, requestMask = ETR1 → requestMask[i] < MAX_QUEUED_REQUESTS → params.this.m_defaultRequiredConfirmations > 1 → (exists t : t.id Not In params.this.m_transactions → t.id In submitTransactions(params).this.m_transactions → ((∀ j : j ≥ 0 → j < MAX_CUSTODIAN_COUNT → params.this.m_custodians[params.sender.pubkey] ≠ Some(j) → t.confirmationsMask [j] = false) ∧ t.creator = params.sender.pubkey ∧ (∀ j : j ≥ 0 → j < MAX_CUSTODIAN_COUNT → params.this.m_custodians[params.sender.pubkey] = Some(j) → t.confirmationsMask [j] = true) ∧ params.this.m_custodians[t.index] = params.sender.pubkey ∧ t.dest = params.dest ∧ params.payload = t.payload ∧ params.bounce = t.bounce ∧ (params.allBalance ∧ t.value = 0 ∧ t.flags = FLAG_IGNORE_ERRORS | FLAG_SEND_ALL_REMAINING ∨ ~params.allBalance ∧ t.value = params.value ∧ t.flags = FLAG_IGNORE_ERRORS | FLAG_PAY_FWD_FEE_FROM_BALANCE)))`
MTS.7	In case of any Exception no state changes but removal the expired transactions can occur	`∀ params, exception : eval(submitTransaction(params)) =`

		`Exception(exception) → exec(submitTransaction(params)).this[m_transactions = ETR1] = params.this ∨ exec(submitTransaction(params)).this = params.this`

Multiple custodian transaction confirmation

No.	Natural language	FELIZ
`MTC.1`	Only *Custodian* can confirm a transaction, otherwise Exception must be raised	`∀ params : eval(confirmTransaction(params)) = ok() → exists i : params.this.m_custodians[params.sender.pubkey] = Some(i)`
`MTC.2`	If: • *sender* is a *Custodian*, and • there is enough gas then : the expired transactions must be removed and committed	`∀ params : exists i : params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(confirmTransactions(params) ≠ Exception("out of gas") → exec(submitTransaction(params)[∀ transaction : ETR1]`
`MTC.3`	If The *Custodian* already signed the Transaction then Exception must be raised	`∀ params, i: eval(confirmTransactions(params) = ok() → (let transaction = params.this.m_transactions[params.transactionId] in → params.this.m_custodians[params.sender.pubkey] = Some(i) → transaction.confirmationMask[i] =`

		false)
MTC.4	If the Transaction with the specified id is not in the queue then Exception must be raised	∀ params : eval(confirmTransactions(params) = ok() → (exists transaction : params.this.m_transactions[params.transactionId] = Some(transaction) /\ transaction.id = params.transactionId)
MTC.5	If: ● *sender* is a *Custodian*, and ● there is enough gas ● The *Custodian* has not signed the Transaction yet ● The Transaction exists in the queue after removal the expired transactions ● A number of signatures received + 1 is less than *a number of signatures* then : ● The sender is added to the list of signers ● A number of received signatures is increased	∀ params, i, transaction : params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(confirmTransactions(params) ≠ Exception("out of gas") → transaction In ETR1 → transaction.confirmationMask[i] = false → transaction.id = params.transactionId → params.this.m_defaultRequiredConfirmations > transaction.signsReceived + 1 → (exist t : t In confirmTransaction(params).this.m_transaction → t.id = params.transactionId → t.confirmationMask[i] = true → t.signsReceived = transaction.signsReceived + 1 → (∀ j : j ≥ 0 → j < params.this.m_custodians.size → i ≠ j → t.confirmationMask[j] = transaction.confirmationMask[i]) → (∀ t2 : t2 ≠ transaction → t2 In params.this.m_transactions → t2 In confirmTransaction(params).this.m_transactions))
MTC.6	If: ● *sender* is a *Custodian*, and ● there is enough gas ● The *Custodian* has not signed the Transaction yet	∀ params, i, transaction : params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(confirmTransactions(params) ≠ Exception("out of gas") → transaction In ETR1 →

	• The Transaction exists in the queue after removal the expired transactions • A number of signatures received + 1 is equal or greater than *a number of signatures* then : • The Transaction is removed from the queue • The request to transfer the the amount saved in the Transaction with the parameters saved in the Transaction must be placed	`transaction.confirmationMask[i] = false → transaction.id = params.transactionId → params.this.m_defaultRequiredConfirmations ≤ transaction.signsReceived + 1 → ((∀ t : t In confirmTransaction(params).this.m_transaction → t.id ≠ params.transactionId) ∧ (∀ t : t In params.this.m_transaction → t.id ≠ params.transactionId) → t In confirmTransaction(params).this.m_transaction) ∧ (let transfer = confirmTransaction(params) in eval(transfer) = ok() ∧ transfer.out.messages.size = 1 ∧ (let msg = transfer.out.messages.head in msg.name = "transfer" ∧ msg.params.dest = transaction.dest ∧ msg.params.value = transaction.value ∧ msg.params.bounce = transaction.bounce ∧ msg.params.payload = transaction.payload ∧ msg.params.flags = transaction.sendFlags)))`

Update request submission

Story: Removal of expired update requests

No.	Natural language	FELIZ
REU.1	Any update request must be removed from the update request list if and only if the update request	`∀ params, u : u In params.this.m_updateRequests → (u.id >> 32) + EXPIRATION_TIME ≤ NOW ↔ (u In`

	was submitted before the EXPIRATION_TIME of seconds from NOW	`params.this.m_updateRequests ∧ u Not In _removeExpiredUpdateRequests(params).this.m_updateRequests ∧ _removeExpiredUpdateRequests(params).this.m_updateRequestsMask[u.index] = false)`

Main part

No.	Natural language	FELIZ
CUR.1	Only *Custodian* can send a code update request, otherwise Exception must be raised	`∀ params : eval(submitUpdate(params)) = ok() → exists i : params.this.m_custodians[params.sender.pubkey] = Some(i)`
CUR.2	The new list of *Custodians* must have at least one *Custodian*, otherwise Exception must be raised	`∀ params : eval(submitUpdate(params)) = ok() → params.owners.size > 0`
CUR.3	The new number of *Custodians* should not exceed MAX_CUSTODIANS, otherwise Exception must be raised	`∀ params : eval(submitUpdate(params)) = ok() → params.owners.size ≤ MAX_CUSTODIANS`
CUR.4	If: • the *sender* is a *Custodian*, and • there is enough gas, and • the new list of *Custodians* has at least one member • the new list of Custodians has not more than MAX_CUSTODIANS elements	`∀ params : exists i : params.this.m_custodians[params.sender.pubkey] = Some(i) → params.owners.size > 0 → params.owners.size ≤ MAX_CUSTODIANS → eval(submitUpdate(params) ≠ Exception("out of gas") → exec(submitUpdate(params)[∀ transaction : REU1]`

	then all the expired update transactions must be removed from the queue. Such an operation must be committed to ensure the state is changed even in case of further Exception	
CUR.5	If: • the *sender* is a *Custodian*, and • there is enough gas, and • the new list of *Custodians* has at least one member • the new list of Custodians has not more than MAX_CUSTODIANS elements then : after the removal the expired update transactions, the *sender* should not have its own update requests in the queue, otherwise Exception must be raised	∀ params, i : eval(submitUpdate(params)) = ok() → params.this.m_custodians[params.sender.pubkey] = Some(i) eval(submitUpdate(params) ≠ Exception("out of gas") → params.owners.size > 0 → params.owners.size ≤ MAX_CUSTODIANS → u = REU1 → u.m_updateRequestsMask[i] = false
CUR.6	If: • the *sender* is a *Custodian*, and • there is enough gas, and • the new list of *Custodians* has at least one member • the new list of Custodians has not more than MAX_CUSTODIANS elements • after the removal the expired update transactions, the *sender* does not have its own update requests in the queue then : • A new update request must be put into the update request queue with the following attributes:	∀ f, params : f ∈ MiltisigFunctions → params.this ∈ Multisig → f ≠ constructor → f ≠ executeUpdate → (∀ u1, u2 : u1 ≠ u2 → tu In params.this.m_updateRequests → t2 In params.this.m_updateRequests → t1.id ≠ t2.id) → (let result = f(params) in (∀ u1, u2 : u1 ≠ u2 → t1 In result.this.m_updateRequests → u2 In result.this.m_updateRequests → u1.id ≠ u2.id)) ∀ params, u, i : → params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(submitUpdate(params) ≠ Exception("out of gas") → params.owners.size > 0 → params.owners.size ≤ MAX_CUSTODIANS → u,that = REU1 → that.m_updateRequestsMask[i] = false →

	○ **Unique** identifier ○ Custodian index ○ Zero number of signs ○ Empty list of signers ○ *sender* ○ New code hash ○ New list of *Custodians* ○ New suggested *number of signatures* ● The update request must be signed by the *sender* that means: ○ The *sender* is added to the list of signers ○ A number of received signatures is increased	`(eval(submitUpdate(params) = ok()) ∧ (∀ ur : ur In u → ur In submitUpdate(params).this.m_updateRequests) ∧ submitUpdate(params).this.m_updateRequestsMask[i] = true ∧ (exists ur : (∀ ur2 : ur2 ≠ ur → ur2 In submitUpdate(params).this.m_updateRequests) → ur In REU1.this.m_updateRequests →) ∧ (exists nur : nur Not In u ∧ nur In submitUpdate(params).this.m_updateRequests ∧ nur.id = i ∧ nur.signs = 1 ∧ nur.confirmationsMask[i] = true ∧ (∀ j : j ≥ 0 → j < params.owners.size → i ≠ j → nur.confirmationsMask[j] = false) ∧ (∀ nur2 : nur2 In REU1.this.m_updateRequests → nur2 ≠ nur → nur2 In REU1.this.m_updateRequests → nur2 In params.this.m_updateRequests) ∧ nur.index = i ∧ nur.signs = 1 ∧ nur.creator = params.sender.pubkey ∧ nur.codeHash = params.codeHash ∧ nur.owners = params.owners ∧ nur.reqConfirms = params.reqConfirms))`
CUR.7	In case of any Exception no state changes but removal the expired transactions can occur	`∀ params, exception : eval(submitUpdate(params)) = Exception(exception) → exec(submitUpdate(params)).this[m_updateTransactions = REU1] = params.this ∨ exec(submitUpdate(params)).this = params.this`

Code update confirmation

No.	Natural language	FELIZ
CUC.1	Only *Custodian* can confirm an update request, otherwise Exception must be raised	`∀ params : eval(executeUpdate(params)) = ok() → exists i : params.this.m_custodians[params.sender.pubkey] = Some(i)`
CUC.2	If: ● *sender* is a *Custodian*, and ● there is enough gas then : the expired update requests must be removed and committed	`∀ params : exists i : params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(executeUpdate(params) ≠ Exception("out of gas") → exec(confirmUpdate(params)[∀ transaction : REU1]`
CUC.3	If The *Custodian* already signed the update request, then Exception must be raised	`∀ params, i, u : eval(executeUpdate(params)) = ok() → params.this.m_custodians[params.sender.pubkey] = Some(i) → u In REU1.this.m_updateRequests → u.id = params.transactionId → u.confirmationMask[i] = false`
CUC.4	If the update request with the specified id is not in the queue after removal the expired update requests then Exception must be raised	`∀ params : eval(executeUpdate(params)) = ok() → exists u : u In REU1.this.m_updateRequests → u.id = params.tranactionId`
CUC.5	If: ● *sender* is a *Custodian*, and ● there is enough gas ● The *Custodian* has not signed the update request yet ● The update request exists in the queue after removal the expired transactions then :	`∀ params, u, i, ur : → params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(confirmUpdate(params) ≠ Exception("out of gas") → u = REU1 → ur In u → ur.transactionId = params.transactionId → u.m_confirmationMask[i] = false → (eval(confirmUpdate(params) = ok() ∧ (let res =`

	• The sender is added to the list of signers • A number of received signatures is increased	`exec(confirmUpdate(params)).this.m_updateRequests in (∀ u2 : u2 ≠ ur → u2 In u → u2 In res) ∧ u.size = res.size ∧ ur[m_confirmationMask[i] = true, signsReceived = signsReceived + 1] In res))`

Code update execution

No.	Natural language	FELIZ
CUE.1	Only *Custodian* can submit an update request, otherwise Exception must be raised	`∀ params : eval(executeUpdate(params)) = ok() → exists i : params.this.m_custodians[params.sender.pubkey] = Some(i)`
CUE.2	If: • *sender* is a *Custodian*, and • there is enough gas then : the expired update requests must be removed and committed	`∀ params : exists i : params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(executeUpdate(params) ≠ Exception("out of gas") → exec(confirmUpdate(params)[∀ transaction : REU1]`
CUE.3	If the update request with the specified id is not in the queue after removal the expired update requests then Exception must be raised	`∀ params : eval(executeUpdate(params)) = ok() → exists u : u In REU1.this.m_updateRequests → u.id = params.tranactionId`
CUE.4	If the update request has less than $\frac{2}{3}$ of the number of *Custodians* signed, the Exception must be raised	`∀ params, u : eval(executeUpdate(params)) = ok() → u In In REU1.this.m_updateRequests → u.id = params.tranactionId → u.signs ≥`

		`params.this.m_custodians.size * 2 / 3`
CUE.5	If the hash code of the Cell provided does not equal to the hash code stored in the update request, the Exception must be raised	`∀ params, u : eval(executeUpdate(params)) = ok() → u In In REU1.this.m_updateRequests → u.id = params.transactionId → u.hashCode = tvm.hash(params.code)`
CUE.6	If: • *sender* is a *Custodian*, and • there is enough gas • the update request exists in the queue after the expired transaction's removal • the update request has at least $\frac{2}{3}$ of the number of *Custodians* signed then the code is upgraded with the parameters provided in the upgrade request	`∀ params, u, i, ur : → params.this.m_custodians[params.sender.pubkey] = Some(i) → eval(executeUpdate(params) ≠ Exception("out of gas") → u = REU1 → ur In u → ur.transactionId = params.transactionId → ur.signs ≥ params.this.m_custodians.size * 2 / 3 → ur.hashCode = tvm.hash(params.code) → (eval(executeUpdate(params) = ok() ∧ (let res = exec(executeUpdate(params)).this.m_updateRequests in (∀ u2 : u2 ≠ ur → u2 In u → u2 In res) ∧ u.size - 1 = res.size ∧ (∀ u2 In res : u2.transactionId ≠ params.transactionId)) ∧ executeUpdate(params)).code = params.code ∧ executeUpdate(params)).currentCode = params.code ∧ let result = executeUpdate(params).this in (result.m_custodians.size ≤ params.owners.size ∧ (∀ i : i ≥ 0 → i < result.m_custodians.size → (exists c : c In result.m_custodians.keys ∧ result.m_custodians[c] = Some(i))) ∧ (∀ i : i ≥ 0 → i < params.owners.size → (exists j : result.m_custodians[params.owners[i]] = Some(j))) ∧ (∀ c1, c2, v : → result.m_custodians[c1] = result.m_custodians[c2] → result.m_custodians[c1]`

		= Some(v) → c1 = c2) ∧ result.m_defaultRequiredConfirmations = min (result.this.m_custodians.sizde, params.reqConfirms) ∧ result.m_ownerKey = params.this.owners[0] ∧ (∀ i : i ≥ 0 → i < 32 → result.m_requestsMask[i] = false) ∧ result.m_transactions = {} ∧ result.m_updateRequests = {} ∧ (∀ i : i ≥ 0 → i < 32 → result.m_updateRequestsMask[i] = false))

Coq-level specification

With low-level properties defined in the previous section the next level was to translate them into native *QuickChick* statements. This activity has been after completion of translation and the results are available by request.

Translation and Verification

Translation

At the first stage the Solidity source code was transformed into *Ursus* representation. The resulting Ursus files are available by request. This activity was made by the proprietary fully-automated translator from *Solidity* to *Ursus* developed by Pruvendo.

Then the conversion from *Ursus* into functional-level code was performed by the semi-automated[7] *Generator* tool also developed by Pruvendo. The results are available by request.

Verification

As a result of the previous stages the following results were obtained:

- The list of the required properties in a Coq-friendly representation
- The code to be verified against these properties (in a Coq-friendly manner as well)

Then, the two options were considered:

- Perform the manual full-scale mathematically strict (deductive) verification
- Perform the lighter version of the verification using QuickChick[8] tool

It was decided that for this simple contract system that does not have a non-trivial logic the usage of the former approach is redundant (deductive verification is very expensive and time-consuming), so the QuickChick approach was chosen.

The corresponding environment has been created and the tool was successfully executed.

[7] It's planned to make it fully automated in the near future.
[8] Some basic information about QuickChick can be found in Appendix I.

Original bug audit

While the original bug is out of the scope of the regular formal verification process (as it was mentioned above it's not a bug of the smart contract itself, but rather a bug of the compiler, that is generally out of scope of existing technologies), the manual audit of the fix was performed.

During the audit it was discovered that the preselector of the internal messaged was added (the Solidity compiler version 0.64) with the following set of primitives:

```
DUP
MYADDR
NEWC
STSLICE
STSLICE
ENDC
HASHCU
```

This set of primitives, according to the manual audit, adds the destination address into the hash and eliminates the risk discovered originally.

Issues found

Throughout the contract the only issue found was as follows:

In case of a very large number of transactions a transaction that was expired can be considered as valid and later executed.

This issue is considered as MINOR as it does not break the underlying business logic and does not lead to any valuable consequenties.

Outcome

The ultimate result of the verification is as follows : it was found the implementation, indeed, has all the declared properties, that means the overall outcome of the formal verification **IS POSITIVE**.

Appendix I. Behind the scene

This appendix provides some more information about the verification process. It may be rather difficult to read and require some advanced skills and knowledge to understand it. The authors, however, tried to be as simple as possible, avoiding hard stuff.

If you, by another hand, would like to know more, feel free to contact us using the means of communications provided on the top of the present document.

In mathematics, there are such software products as Proof Assistants, one of them is Coq. These products are able to automatically check if a theorem was proved correctly or not (and also provide some assistance with proving). Surprisingly, according to Curry-Howard correspondence, the software programs are isomorphic to the mathematical proofs, which gives an opportunity to use Proof Assistants for the proving of software.

However, the Curry-Howard correspondence considers a computer program as written in a typed declarative programming language with a property: it must halt at some time point. The Everscale smart contract developers use typed imperative programming languages - such as *Solidity* or *C++*, that are Turing-complete that means that, generally speaking, it's not possible to check if they halt at some moment or not.[9]

So the first goal of the verifiers was to translate the imperative Turing-complete code into the functional code that always terminates.

To achieve this goal the *Ursus* language has been invented. It's an imperative language with syntax very close to *Everscale Solidity*, but at the same time it's a DSL on top of declarative Coq-environment. Also a correct Ursus program always terminates, which means that some Solidity programs can not be translated into Ursus (or require some refactoring). Fortunately, it is an extremely rare case in the real world and usually implies a poor design of original smart contracts.

While an *Ursus* program can be already handled by Coq Proof Assistant, its imperative structure makes this activity extremely difficult so one more big step is required: convert imperative *Ursus* DSL code into a set of functions.

Each original function is converted into two:

- *eval* - that represents the return value of the function

[9] Due to the halting problem.

- *exec* - that represents the state of the machine after the execution of the function

Both functions must use the current state of the machine as one of the input parameters.

When this task is completed the verification becomes much easier. The program becomes a set of functions that can be used to define properties to be verified (this kind of definition was referenced as Coq-level specification throughout the present document). And nothing prevents us any more from proving these properties using the powerful Coq Proof Assistant capabilities.

Just one important notice. While the approach described above is fully valid and should be used in many cases, its serious drawback is high-cost of the proving as well as very high requirements for the qualification of executors.

While the systems of smart contracts with complicated business logic and/or implementation leave no other options, for simple systems a lighter approach with straightforward logic can be used. The idea is to use *QuickChick* randomized property-based Coq plugin that verifies the properties not by strict mathematical proving but by providing random input data checking.. It's worth mentioning that this approach, while inferior to deductive verification, is still much more powerful than traditional random testing, as predicates allow to automatically discover classes of equivalence while the classic approach fully relies on the operator's expertise.

Appendix II. Project structure

This information can be useful only in case the reader requested additional information from the verifier and wishes to deeply dive into the project.

The verification project is located at https://github.com/Pruvendo/vesting-pool and has the following structure.

File/Directory	Description
`README.md`	A copy of the executive summary of the present document
`dune-project`	Description file for the build system for OCaml[10] - dune
`/ref/multisig/multisig.col`	The source code of the contracts were verified
`/src`	The verification source code

[10] OCaml - generic purpose functional programming language on which Coq proof assistant is based.