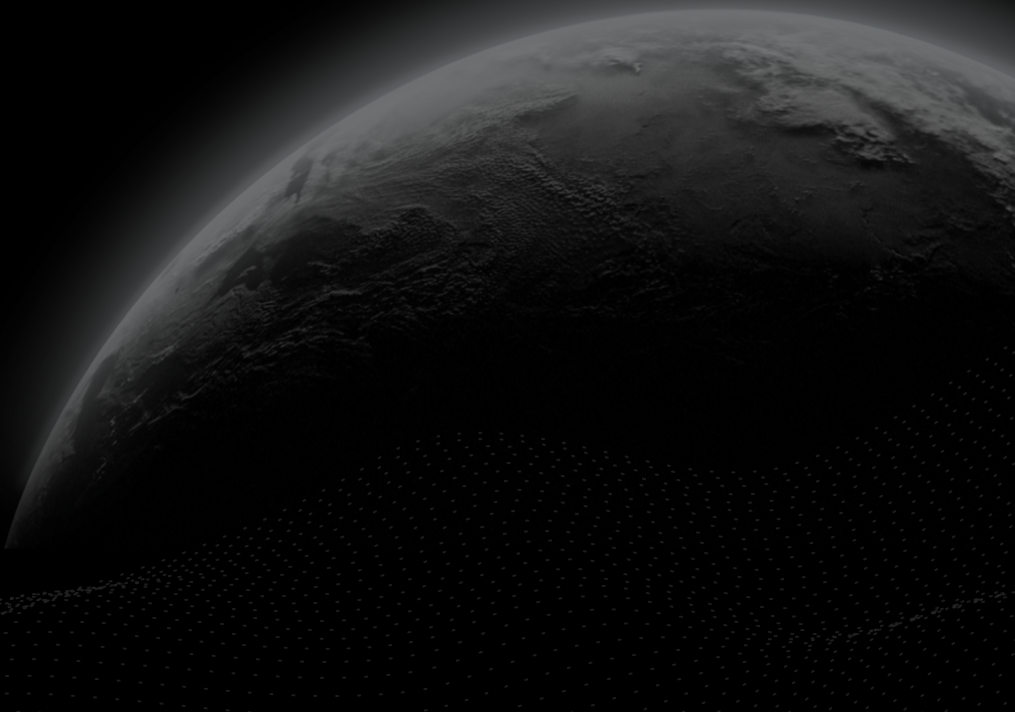




Security Assessment

The Open Network - Formal Verification 1 (Phase 2)

CertiK Verified on Sept 14th, 2022





CertiK Verified on Sept 14th, 2022

The Open Network - Formal Verification 1 (Phase 2)

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES

Other-Contract, Other-Non-Contract

ECOSYSTEM

TON

METHODS

Formal Verification, Manual Review

LANGUAGE

FunC

TIMELINE

Delivered on 09/14/2022

KEY COMPONENTS

N/A

CODEBASE

<https://github.com/newton-blockchain/ton>
[...View All](#)

COMMITTS

<ae5c0720143e231c32c3d2034cfe4e533a16d969>
[...View All](#)

Vulnerability Summary


5

Total Findings

4

Resolved

0

Mitigated

0

Partially Resolved

1

Acknowledged

0

Declined

0

Unresolved

■ 0 Critical

Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.

■ 0 Major

Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.

■ 1 Medium

1 Resolved



Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.

■ 2 Minor

2 Resolved



Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.

■ 2 Informational

1 Resolved, 1 Acknowledged



Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS

THE OPEN NETWORK - FORMAL VERIFICATION 1 (PHASE 2)

I **Summary**

[Executive Summary](#)

[Vulnerability Summary](#)

[Codebase](#)

[Audit Scope](#)

[Approach & Methods](#)

I **Formal verification**

[Introduction](#)

[Trusted computing base](#)

[Description of the contracts and verification scope](#)

[A functional model of the contract code](#)

[Details of the functional modeling](#)

[Re-usability of the functional model](#)

[Parts of the contract covered.](#)

[Accounting invariant of the elector contract](#)

[What could go wrong?](#)

[Wellformedness invariant of the elector contract](#)

[What could go wrong?](#)

[Assumptions of verification for the elector contract](#)

[External guarantees of the elector contract](#)

[Contract will not run out of Grams](#)

[Contract always locks up at least MIN_TOTAL_STAKE Grams.](#)

[Describing the externally observable behavior of the config contract](#)

[Validator-set invariant of the config contract](#)

[Voting invariant of the config contract](#)

[What could go wrong?](#)

[Assumptions of verification for the config contract](#)

[External guarantee of the config contract](#)

[Proof Summary](#)

I **Verification Details**

[Invariant preservations for elector-code.fc](#)

[Verification #1](#)

[Code](#)

<u>Specification</u>
<u>Verification #2</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #3</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #4</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #5</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #6</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #7</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #8</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #9</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #10</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #11</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #12</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #13</u>
<u>Code</u>
<u>Specification</u>
<u>Verification #14</u>
<u>Code</u>

Specification

Verification #15

Code

Specification

Verification #16

Code

Specification

Verification #17

Code

Specification

Verification #18

Code

Specification

Vset invariant preservations for config-code.fc

Verification #19

Code

Specification

Verification #20

Code

Specification

Verification #21

Description

Specification

Voting invariant preservations for config-code.fc

Verification #22

Code

Specification

Verification #23

Code

Specification

Verification #24

Code

Specification

Verification #25

Code

Specification

Verification #26

Code

Specification

Safety theorem for config-code.fc
Verification #27
Code

Specification

I Findings

CKP-01 : Portion of bid above `max_stake` silently discarded.

CKP-02 : No check for existing key in `announce_new_elections`/`conduct_elections`

CKP-03 : Not deducting message cost in `conduct_elections`

CKP-04 : Unused function argument in `announce_new_elections()`

CON-01 : `losses` is not updated if there are no votes in a period

I Appendix

I Disclaimer

CODEBASE | THE OPEN NETWORK - FORMAL VERIFICATION 1 E (PHASE 2)

Repository

<https://github.com/newton-blockchain/ton>






Commit

[ae5c0720143e231c32c3d2034cfe4e533a16d969](#)

AUDIT SCOPE

THE OPEN NETWORK - FORMAL VERIFICATION 1 (PHASE 2)

5 files audited ● 1 file with Acknowledged findings ● 1 file with Resolved findings ● 3 files without findings

ID	File	SHA256 Checksum
● CON	 projects/ton/crypto/smartcont/confi-g-code.fc	444b7d64d05becf137972166f8c19c3cda672e750b634324c8c09dc4d4eca1
● CKP	 projects/ton/crypto/smartcont/lector-code.fc	db7488346b4943809e93dd8fafbcf2296221a3356ae3af4045f95a91d6f68786
● COF	 projects/ton/crypto/smartcont/confi-g-code.fc	fa151891a9acfa54f26e3dd8d84aa4a6c7a3b61149d67b1c7526196f3f52ad10
● COS	 projects/ton/crypto/smartcont/constants.fc	5dca49c65efdb6f552571c1d23eef13a4122d9fa5a4871cd43447a607399c0fd
● ELE	 projects/ton/crypto/smartcont/lector-code.fc	80d0b796890a91890bab14e786b0a5781d905b2a374bf5db05b965071d43add7

APPROACH & METHODS

THE OPEN NETWORK - FORMAL VERIFICATION 1 (PHASE 2)

This report has been prepared for The Open Network to discover issues and vulnerabilities in the source code of the The Open Network - Formal Verification 1 (Phase 2) project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Formal Verification and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

FORMAL VERIFICATION

THE OPEN NETWORK - FORMAL VERIFICATION 1 (PHASE 2)

Introduction

We apply formal verification using the Coq proof assistant to verify basic security properties of the `elector-code.fc` and `config-code.fc` contracts. Specifically we prove that the election functions will not lose track of funds, that the election contract locks up at least `min_total_stakes` grams in each election, and that the config contract correctly counts votes for configuration change proposals.

The rest of the report describes the verification in more detail. The main outputs are

- A functional model of the contract code, suitable for use in a proof assistant.
- Definition of an accounting invariant, and proofs that the functions satisfy it.
- Definition of a minimum stake invariant, and proofs that the functions satisfy it.
- Definition of a voting invariant, and proofs that the functions satisfy it.
- The above proofs are summarized as a detailed set of Verifications, i.e. formally proven claims about each function.
- Finally, a list of Findings, where carrying out the formal proofs revealed a few places where the code should be improved.

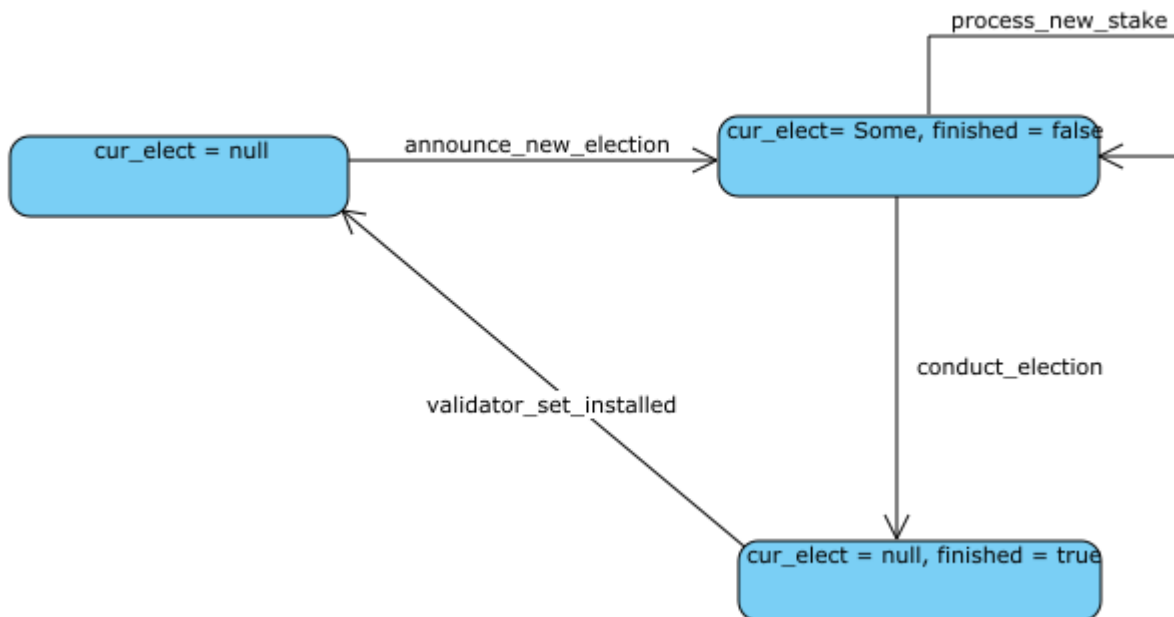
Trusted computing base

Formal verification gives very high confidence in the properties that are proven, but it is always done with a set of background assumptions and trusted specifications. In particular, the machine checked proofs in this development

- Only proves the specific claimed properties.
- Only covers a specific set of contract functions (described below).
- Is done with respect to a model of the code (see section Functional Model below), and relies on manual auditing to check that the model is accurate.
- Is done with respect to a manually written set of assumptions (see section Assumptions of Verification below).

Description of the contracts and verification scope

The `elector-code.fc` implements several functions: staking, election of validators, complaints, and code upgrades. In this verification effort we dealt with the staking/election cycle, while we did not treat complaints or upgrades.



The diagram above shows how the election cycle, which is used to select the validator set for the next period. It is mostly implemented in the `run_ticktock` function in the contract. In the first state, the `cur_elect` field of the contract state is set to `null`. Eventually the function `announce_new_elections` runs and concludes that it is time for a new election period, which updates the `cur_elect` field to indicate that election has begun. It also fetches the relevant data from the next validator set stored as a part of the configuration parameters for the contract, in order to compute the time period over which this election is to stay active, and stores it in the election. Once this is done, it modifies the state of the contract to indicate that an election is ongoing.

While the election is ongoing, users will send new stake requests (handled by the `process_new_stake` function) to register their bids for the current election.

When the current time surpasses the value in the `elect_close` field of the election, the `conduct_elections` function resets the finished flag to true, updates the current validator set to the set stored as the next validator set, and adds the current election to the `past_elects` dictionary under the `elect_at` key obtained from the election.

Once this is done, the `validator_set_installed` function modifies the state so that there is no ongoing election (i.e resets the `cur_elect` field to `null`), and updates the `active_id` stored in the state to the id of the new validator set.

At any time, the functions `process_simple_transfer` and `process_new_stake` are called to allow for a participant to increase their stake in a particular election (this is rejected if there is no ongoing election) and to transfer value to the contract either as a bonus to one of the elections or to the amount stored in the contract respectively. Also, at any time a user can invoke `recover_stake` to retrieve stakes from old elections which have already been unlocked.

Our formal verification covers all the functions mentioned in the above descriptions.

The `config-code.fc` contract is responsible for updating the blockchain configuration register. It provides three main functionalities: it updates the current chain validator set in response to messages from the elector contract, it changes a configuration parameter or contract code in response to a properly signed message from the chain owner (`perform_action`), or it makes such a change in response to a change-proposal which has been voted on by validators (`perform_proposed_action`).

The `perform_action` code path is important for security but it is algorithmically trivial, just an if-statement to check if the signature matches. Therefore we do not consider it in this formal verification effort. Instead we focus on the `perform_proposed_action` path, which is equally security-critical, but involves nontrivial logic spread over several different functions.

Any user can submit a change proposal by calling the contract. The proposal has an expiration time and the user pays a fee proportional to the time until expiration. Then the validators in the current validator set can vote in favor of the proposal. If more than 75% of the validators (counted by weight) vote for it, that is defined as a "win", while if it has not received enough votes at next change of validation set it is defined as a "loss". The chain configuration register includes parameters `min_wins` and `max_losses`, and in order to pass a proposal needs to receive at least `min_wins` wins before it expires. It can also be discarded early if it accumulates more than `max_losses` losses, although this guarantee is less hard (see finding CON-01).

The contract maintains a dictionary containing the status of all current proposals. The main complication that makes the logic nontrivial is that, in order to save on gas costs, the statuses are updated "lazily" in response to votes, rather than at the time when the validator set changes. Therefore, each proposal is tagged with the last validator set considered, and all operations refresh the win/loss counts if they are out of date. In general, the life cycle of a change proposal is as follows:

- A user submits a new proposal: the `register_voting_proposal` function computes the fee, and initializes the proposal with a win-count of zero.
- Validators submit votes, via either external or internal messages, which are dispatched to the `proceed_register_vote` function. This function calls `register_vote` to compute how if the proposal has won this round or passed overall, and deletes expired proposals. If a vote made the proposal pass, `accept_proposal` and `perform_proposed_action` are called to compute and store the updated configuration.
- Meanwhile, the elector contract will periodically send new validator sets, which are checked by `check_validator_set` and remembered as the "next" set.
- When the time is right the `run_ticktock` function will activate the next validator set, defining a new voting period.
- The ticktock function can also call `scan_random_proposal` and `scan_proposal`, which can "garbage-collect" a random outdated proposal without waiting for any further voting on it.

In our proofs we don't analyze the code of `proceed_register_vote`, but just start with the assumption that the proposal entry has been correctly initialized to zero wins. We also don't look into the definition of `perform_proposed_action`, but just note that there is some function which will update the configuration record when called. Otherwise, our formal verification covers all the functions mentioned in the above description.

■ A functional model of the contract code

A prerequisite for any kind of formal verification is to model the code inside the proof assistant. Therefore, one of the main output of the specification process is a formal model of the contract state that is modified as a result of the execution of various methods of the contract. Thus the specifications along with the Coq definitions are a formal state machine representation of the contract.

The higher-level of abstraction of this representation can be seen from the types the functional specifications in Coq. In the FunC source code there are primitive types like `int` and `bool`, but all compound data is typed as just `cell`. For our functional models, we use a strong type system which represents the data as Records with typed fields, and dictionaries with

a type of content. Defining these types was time consuming and important for writing succinct specifications and important invariants that should be preserved by the contract functions.

This process also lends to having useful documentation for the contract, making it easier for the reader to precisely understand how each function operates based on the "higher-level" type of the input data.

However, one caveat of this kind of formal verification is that it is meant to detect issues with the contract logic and its execution only. It would not discover type errors or typos in the FunC code if those errors were inadvertently corrected during the manual translation. This aspect of the contract security is covered as a part of the manual audit of the contract already performed by Certik.

Details of the functional modeling

We mostly work line by line. Exceptions are the pack and unpack operations, which are translated directly into operations on tuples, and loops, which are manually translated into (manifestly terminating) folds. As an example of how we wrote the specification, consider the `unfreeze_all()` function below :

```
_ unfreeze_all(credits, past_elections, elect_id) inline_ref {
  var (fs, f) = past_elections-udict_delete_get?(32, elect_id);
  ifnot (f) {
    ;; no elections with this id
    return (credits, past_elections, 0);
  }
  var (unfreeze_at, stake_held, vset_hash, fdict, tot_stakes, bonuses, complaints) =
  fs.unpack_past_election();
  ;; tot_stakes = fdict.stakes_sum(); ;; TEMP BUGFIX
  var unused_prizes = (bonuses > 0) ?
    credits-unfreeze_with_bonuses(fdict, tot_stakes, bonuses) :
    credits-unfreeze_without_bonuses(fdict, tot_stakes);
  return (credits, past_elections, unused_prizes);
}
```

This FunC function is translated to the following Coq specification :

```

Definition unfreeze_all_spec (credits : Dict Integer) (p : Dict PastElections)
  (elect_id : Integer)(r : RData)
  : option (Dict Integer * Dict PastElections * Integer * RData) :=
match (udict_delete_get_spec p 32%Z elect_id) with
| (Some d', Some v, true) =>
  match (unpack_past_election_spec v r) with
  | Some ( unfreeze_at, stake_held, vset_hash, f_dict, tot_stake, bonuses,
complaints, r') =>
    if (0%Z <? bonuses)%Z
    then
      match (unfreeze_with_bonuses_spec credits f_dict tot_stake bonuses r)
with
      | Some (credits', u_p, r'') => Some (credits', d', u_p , r'')
      | None => None
    end
    else
      match (unfreeze_without_bonuses_spec credits f_dict tot_stake r) with
      | Some (credits', u_p, r'') => Some (credits', d', u_p , r'')
      | None => None
    end
  | None => None
end
| _ => Some (credits, p, 0%Z, r)
end.

```

The key part of the model is a description of all the relevant state stored by the contracts. For each contract, we model the state through a record type in Coq called `RData`. The type is defined so as to capture the information related to the contract in the control registers `c4`, `c5` and `c7` of the TVM. The `c7` control register stores the configuration parameter of the elector contract which are set by the config contract and deleted once the contract terminates. It contains the address of the accounts containing the config code contract and the elector code contract respectively. It also contains various other parameters which govern the elections and the complaints for the elector contract, like the maximum and minimum number of validators allowed in any elected validator set, the price of registering a complaint, the total stake in the contract etc. The data stored in the `c4` register is the main focus of `RData` and forms most of the fields of the record. `c4` represents the root of the persistent storage associated to the account of the elector contract. Most functions in the contract modify this data.

There are two different `RData` types for the two contracts. In the case of the elector contract, the contents of the record represents the current ongoing election, the dictionary of past elections held, the id and hash of the presently elected validator set etc. Finally, `c5` models the list of output actions of the contract and is represented as one field within the `RData` type which is a list of messages. Note that the Coq model of the elector contract does not take into account the gas usage, and hence gas is not modelled as a part of the contract state.

```
Record config_par : Set :=
{
  Csc : option Addr;      (* The configuration smart contract *)
  Esc : option Addr;      (* The elector smart contract *)
  Vst : option vset;      (* Current vset *)
  Vst_next : option vset; (* Next vset *)
  Stk : stake;
  Vg : V_group;
  Cp : option complaint_price;
  Vc : validator_conf;
  Stk_bounds : Integer * Integer * Integer * Integer; (* min_stake, max_stake,
min_total_stake, max_stake_factor *)
  validators_lim : Integer * Integer * Integer (* max_validators, _,
min_validators*)
}.

Record message : Set :=
{
  addr : Addr;
  ans_tag : Integer; (*Int32;*)
  query_id : Integer; (*Int64;*)
  gram : Integer;
  body : message_body;
}.

Record RData : Set :=
{
  c7_tuple : config_par;

  (* The following items specify the c4 "persistent data" control register. *)
  cur_elect : option elections;
  credits : Dict (*key int256*) Integer;
  past_elects : Dict (* key int32*) PastElections;
  grams : Integer;
  active_id : Integer; (*Int32;*)
  active_hash : Integer; (*Int256;*)

  (* The following item specify the c5 "output action" register *)
  outputs : list message;
}.
}
```

Re-usability of the functional model

While in this report we prove that the contract satisfies two particular invariants, the functional model captures all the behavior of the code, so later we or another auditor could use the same model of the code to prove a different invariant. Having such a model enables future verification efforts.

In addition to more invariants, if desired one could also prove that the low-level code of the contract correctly implements the functional models. (In some preliminary experiments we validated this approach by sketching a model of the TVM and the FunC language in Coq, translating the compiler output into a form readable by the proof assistant, and proving the code of one small function.) This is another way that the specifications are reusable for the future; in this scenario, the role of them would be inverted, so that instead of the serving as the code which we prove satisfy a high-level specification, they would be the specification which the low-level TVM code would satisfy.

Since the models are the same for both code proofs and invariant proofs, those proofs could be composed to show that the low-level code satisfies the high-level invariants. This verification approach is thus inherently extensible.

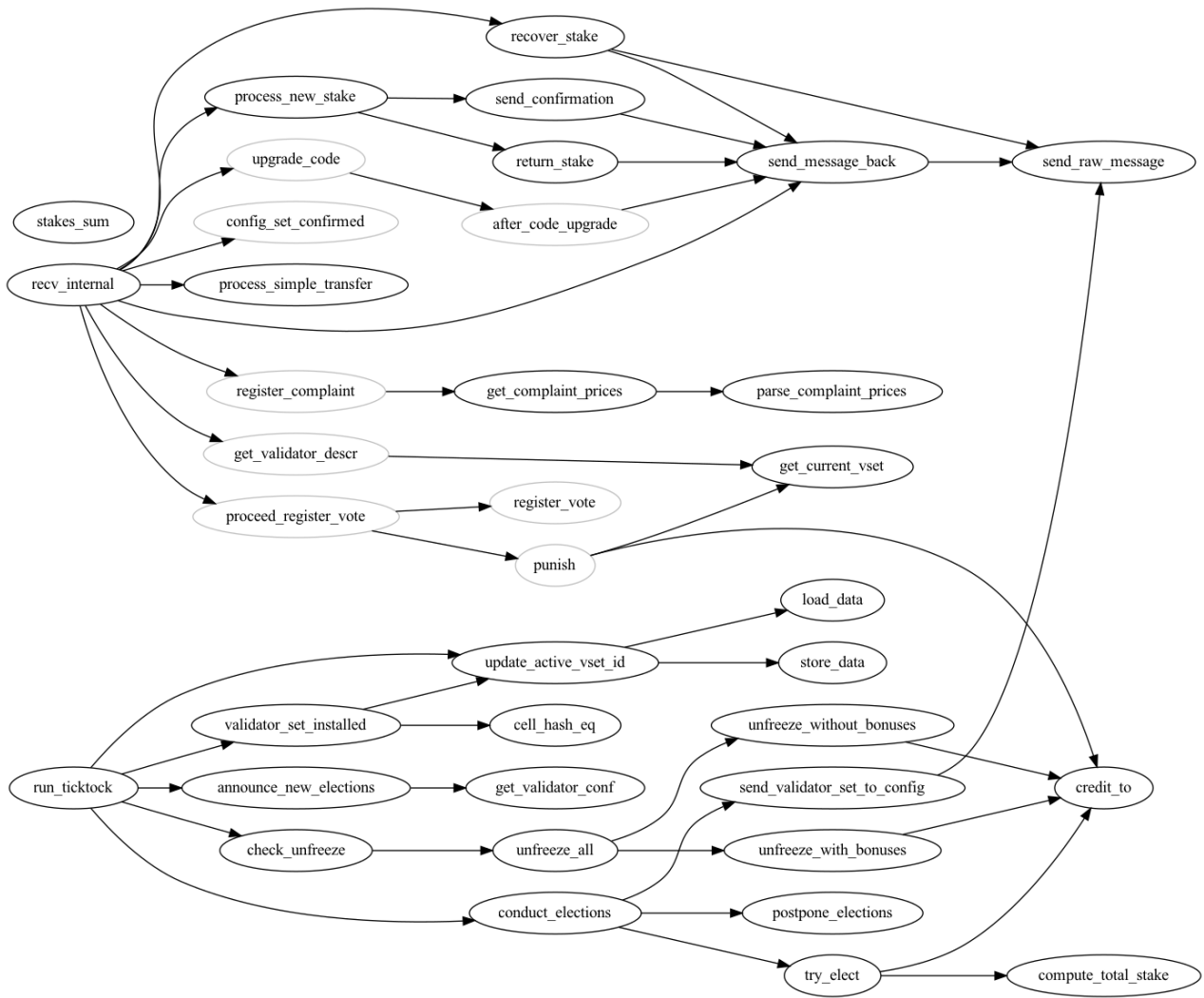
Parts of the contract covered.

The `elector-code.fc` contract contains 58 functions in total. First there are 12 data representation functions (`load_data`, `store_data`, `unpack_elect`, `pack_elect`, `unpack_past_election`, `pack_past_election`, `unpack_complaint_status`, `pack_complaint_status`, `unpack_complaint`, `pack_complaint`, `parse_complaint_prices`, and `get_complaint_prices`). These parse and serialize data stored in TVM cells, and encapsulates the information of about the precise layout of data in storage. However, the rest of the code does not uniformly work on this level of abstraction, there are also many direct uses of the low-level `get_data`, `set_data`, and low-level cell builders and parsers. In our formal development we provide models for the data representation functions. These are quite simple since (as we explained above) the formal specifications already work on a typed description of the contract data and elides the details about the representation format.

Further, there are 12 "getter functions" (`active_election_id`, `participates_in`, `participant_list`, `participant_list_extended`, `compute_returned_stake`, `past_election_ids`, `past_elections`, `past_elections_list`, `complete_unpack_complaint`, `get_past_complaints`, `show_complaint`, `complaint_storage_price`). These do not change contract data and are never used by the contract itself, so we do not model them. (They can be called by light nodes and the TON explorer.)

The interesting part of the contract is the remaining 34 functions, which implement the business logic of the contract. There are two entry points for handling messages (`recv_internal`) and running per-block code (`run_ticktock`), which in turn call other functions. Of these, we have written functional models for the 26 functions which are involved in the staking/election functionality. We omit writing models for 8 functions which are used for the functionality which we do not model (code upgrades, voting for complaints, and acknowledging the message from the config contract).

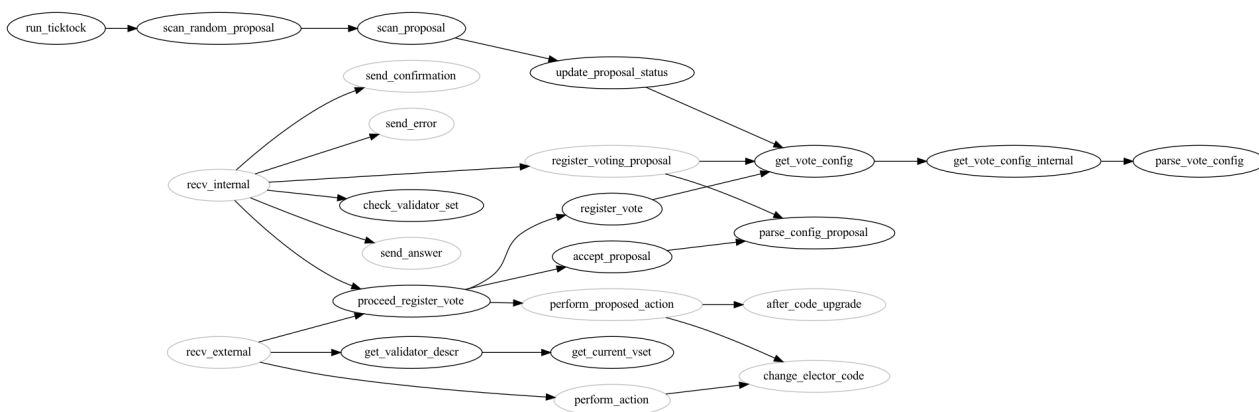
The below figure shows the call graph of the main 34 functions in the contract (to reduce clutter it omits the data representation functions, which are called from most of these functions). The color shows which of them we have provided functional models for (black) and which are not modelled (grey).



As can be seen, we do not provide a model for the complaint functions, which we don't prove correct. We also don't directly model `recv_internal` because we don't model all the functions it depends on—instead we prove that the functions it dispatches into satisfy the invariants.

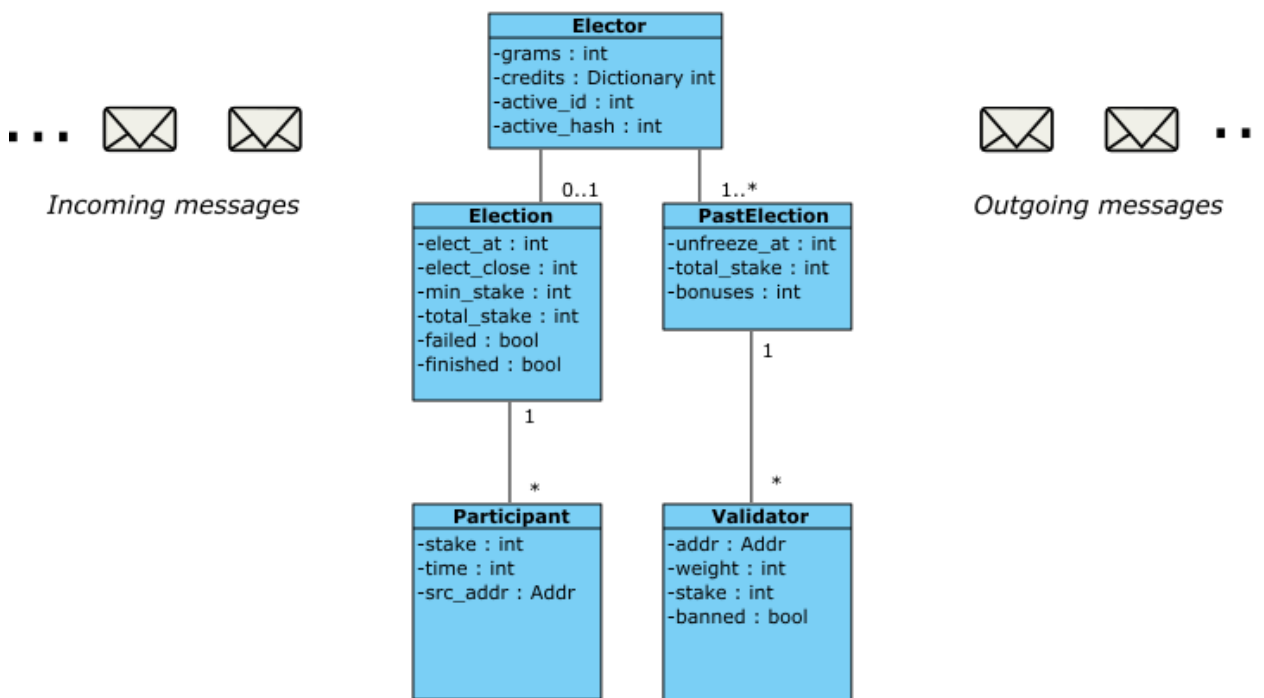
The `config-code.fc` contract contains 35 functions. Of these, 7 are data representation functions (`set_conf_param`, `load_data`, `store_data`, `parse_vote_config`, `unpack_validator_descr`, `begin_pack_proposal_status`, and `unpack_proposal_status`), and 5 are getter functions (`seqno`, `unpack_proposal`, `get_proposal`, `list_proposals`, and `proposal_storage_price`).

The remaining 23 are showed in the figure below. We write models for 13 of them, while leaving 10 unspecified (marked in grey). In particular, in this contract the functions `perform_proposed_action`, `send_confirmation`, and `send_error` are left as a parameters. That is, those three functions do not themselves change the vote counts, so in order prove the voting mechanism the model only needs to say that there is *some* function being called, but can omit the precise code. We do not provide a model for the contract owner mechanism, which we don't prove correct. Similarly to `elector-code`, we don't directly model `recv_internal` / `recv_external`.



Accounting invariant of the elector contract

The overall structure of the elector contract is illustrated in the below figure. The contract receives incoming messages, each of which can carry some *value* of Grams which increments the balance of the contract. In response it will update its internal data structures and possibly add reply messages to the message output queue, from which they will be sent asynchronously. When the output messages are sent, they decrement the contract balance by the value stored in them.



The balance of the contract itself therefore reflects the sum of funds from a number of sources: the stakes for the current validator set, the bids that have been submitted for the upcoming election, and any stakes for previous validator sets that have not yet been withdrawn. The data structures of the contract needs to track exactly how much of this balance belongs to any given participant. This is one of the most crucial things to get correct, because a bug in these calculations could allow someone to withdraw more than they are entitled to, draining the contract of funds. Such a bug would be catastrophic both for the validators (who got their stake stolen) and for the blockchain consensus itself (there would no longer be any stake keeping the validators honest), so it is critical to avoid it.

In order to rule out such bugs, we formally define an *accounting invariant* of the contract, which says that the sum of the values in the contract data structures (the "contract_value") always correspond to the balance of the contract itself (the sum of incoming messages minus sum of outgoing messages), and then prove that each of the contract methods preserve it. The

most interesting part of the invariant definition is that it provides a formal definition of `contract_value`, which explains exactly what value the data structures represent. We show the complete definition below. It is the sum of stakes stored in `cur_elect` (unless the `finished` flag is set), the stakes in `past_elects`, the `credits` and `grams`, and the queue of outgoing messages.

```

Definition sum_with {A:Set} (f : A -> Integer) (d : Dict A) : Integer :=
  sum (List.map (fun kv => f (snd kv)) (dictionary.elements d)).

Definition stake_from_participant (p : Participant) :=
  let 'mkParticipant stake _ _ _ _ := p in stake.

Definition CurElections_value (e : elections) :=
  if (finished e)
  then 0
  else sum_with stake_from_participant (members e).

Definition stake_from_validator (v : Validator) : Integer :=
  let '(mkValidator addr weight stake banned) := v in stake.

Definition PastElections_value (p : PastElections) :=
  bonuses p
  + sum_with stake_from_validator (frozen_dict p).

Definition sum_of_participants (participants : Dict Participant) : Integer :=
  sum (List.map (fun k => stake_from_participant (snd k))
    (dictionary.elements participants)).

Definition sum_of_validators (participants : Dict Validator) : Integer :=
  sum (List.map (fun k => stake_from_validator (snd k))
    (dictionary.elements participants)).

Definition contract_value (r : RData) : Integer :=
match (cur_elect r) with
| Some elect =>
  CurElections_value elect
  + sum_with PastElections_value (past_elects r)
  + sum_with id (credits r)
  + grams r
  + sum_of_value (outputs r)
| None =>
  sum_with PastElections_value (past_elects r)
  + sum_with id (credits r)
  + grams r
  + sum_of_value (outputs r)
end.

```

This definition is short, but note that it could only be written because we have a typed model of the data stored in the contract, the `RData` we described above. In addition to our proofs the the contract functions satisfy it, it also serves as

documentation of the invariants the contract is meant to satisfy, which can help future developers adding new features, or TON users trying to understand it. Previously this was not written down anywhere, and it is not entirely obvious. (In particular, the behavior of the `finished` flag is somewhat subtle: once an election has been marked as finished, the credits in the `cur_elect` structure should no longer be considered as counting towards the contract value because they have also been copied to the `past_elects` structure. This is handled by the `if (finished e)` statement above.)

Once we have this definition, we prove for each function that it never "misplaces" any grams, i.e. when each incoming message is processed the contract value increases by (at least) that amount.

What could go wrong?

The accounting invariant rules out problems where the data structures fail to track the incoming/outgoing value. One illustration is our finding `CKP-01`: if a user submits a bid with too high value (more than the `max_stake` contract parameter), the `try_elect` function will silently forget about the "extra" part instead of adding it to the user `credits`. Similarly `CKP-03` shows a function not tracking the cost of sending a message. These findings came up as we tried to prove the accounting invariant for the corresponding functions.

Wellformedness invariant of the elector contract

The accounting invariant shows that the contract's internal accounting of how many Grams it has is correct. The end users of the contract will also want to know that the number is the intended one. Therefore, we prove a second invariant: that each of the frozen elections in `past_elects` contain at least `min_total_stakes` credits. Together with (informal) inspection of the election logic to show that there is always a frozen election corresponding to the current validator set, that proves that the contract always locks up the right amount of stakes. The property is stated using the `sum_of_validators` definition which we showed above.

```
Record PastElections_wellformed (ps: Dict PastElections) : Prop :=
{
  PastElections_wellformed_bonuses: forall k p,
    get k ps = Some p ->
    (bonuses p) >= 0;
  PastElections_wellformed_total_stake: forall k p,
    get k ps = Some p ->
    sum_of_validators (frozen_dict p) >= MIN_TOTAL_STAKE
}.
```

Again, we prove that each of the functions preserve `PastElections_wellformed (past_elects r)` as an invariant.

What could go wrong?

All functions in our verification set do in fact satisfy the invariant. The most complicated proof is for the `try_elect` function which constructs a new validator set. This is because the function never directly iterates of the final validator set to check how much stake is in it. Instead, this is ensured indirectly: the function gathers all the bids into an intermediate dictionary, then gets them out of the dictionary into an intermediate sorted list (with stake information), then selects a subset of that list that should add up to at least `min_total_stake` (returning failure if this is impossible), and finally builds a validator set from

the subset list. In our proof, we check the relation between the intermediate data structures to make sure that this logic is sound and the function does in fact include enough stakes in the final output.

Assumptions of verification for the elector contract

The theorems we prove use certain *assumptions*, which are stated but not proved in Coq. The proofs will only be valid if the assumptions are satisfied. Some of these are reasonable, while others point to ways the contract code could be improved.

The `try_elec` spec in the very first loop retrieves the stake from the members dictionary and uses the minimum of this stake and `max_stake` to create a new dictionary that is later needed to create a sorted list, with stakes in descending order. The final loop uses this stake from the list, which is the minimum of original stake from members and `max_stake` to create a new validator with some stakes, and send rest of the amount back to the creditors. In case, the stake is greater than the `max_stake`, the value greater than the `max_stake` will not be returned to the creditor and would get lost in the contract. To make our proof work in the presence of this bug, we added an assumption to the proofs while maintaining the same spec. The assumption that states all the stakes in the members dictionary are less than the `max_stake`.

`MIN_TOTAL_STAKE` is unchanged. We want to show that the contract always keeps at least the minimum stake. However, `min_total_stake` is a configuration parameter which could be changed at any time, and if it was increased, then the previously conducted election would become "under water". For the purposes of the proof we assume that there is a particular number `MIN_TOTAL_STAKE`, and all calls the `min_total_stake` config parameter returns that value.

Users do not submit more than `MAX_STAKE`. As we described in Finding CKP-01, the contract can lose grams if users submit too large stake requests. We recommend changing the code, but meanwhile we prove the theorem under the assumption that no such large stake request occurs.

No existing `elect_at` key. As described in Finding CKP-02 we recommend adding another if-statement to the contract to make it easier to prove that the dictionary key is unique, but meanwhile we prove the theorem under an assumption that there will be no key clashes.

Nonnegative `grams` field. The grams field in the contract is only ever incremented, so it can not be negative. This property is provable, but it is sufficiently trivial that we say that it holds by inspection.

These assumptions are stated together as formal definition `rdata_assum`, which is an extra assumption to all our theorems.

```

Definition stake_assum (members : Dict Participant) (max_stake : Integer) : Prop :=
  forall (key : key) (stake time max_factor addr adnl_addr : Integer),
  get key members = Some (mkParticipant stake time max_factor addr adnl_addr) ->
  stake = min stake max_stake.

Record rdata_assum (r : RData) :=
{
  config_assum_stk: Stk_bounds (c7_tuple r)
    = (MIN_STAKE, MAX_STAKE, MIN_TOTAL_STAKE, MAX_STAKE_FACTOR);
  stake_assum_current : forall elct,
    cur_elect r = Some elct -> stake_assum (members elct) MAX_STAKE;
  grams_nonneg : grams r >=0;
  assum_elect_at_key : forall elct,
    cur_elect r = Some elct ->
    finished elct = false ->
    get (elect_at elct) (past_elects r) = None
}.

```

Negligible message costs in `conduct_elections` As we described in is Finding CKP-03, the value of the message sent by this function (1 gram) is not accounted for. We recommend changing the code, but meanwhile we modified the functional model to pretend that this message has zero value.

External guarantees of the elector contract

We can finally put together all the proofs to give statements which are relevant to the end user of the contract.

Contract will not run out of Grams

Specifically, if we consider all "simple transfer", "process stake", and "recover stake" messages sent to the contract, and also all "ticktock" transitions that it can make itself, then the contract state satisfies the following invariant.

```

Definition credits_invariant (st : TON_state) : Prop :=
  let (to_contract, from_contract)
    := List.partition (fun m => Addr_eqb elector_addr (addr (snd m)))
      (ton_messages st)
  in
  sum_of_value (List.map snd to_contract)
  >= contract_value (elector_state st) + sum_of_value (List.map snd from_contract).

```

In other words, the amount of value sent *into* the contract is enough to cover the current contract value minus the value of all the messages sent *out of* the contract. This means that no bug drained the contract balance so much that it could not send the outgoing messages.

Contract always locks up at least `MIN_TOTAL_STAKE` Grams.

We prove that each frozen election contains at least `MIN_TOTAL_STAKE` stakes. By manual inspection we see that there is always at least one frozen election, for the current validator set, which shows that the validators are backed by the appropriate amount of stake.

■ Describing the externally observable behavior of the config contract

Next we describe the invariants of the config contracts. While the contract code itself is smaller, the invariants use more elaborate definitions. Our ultimate goal is to prove a safety property, that a proposal will not be accepted unless it has received enough votes, which in turn requires us to prove that the vote counts in the contract data structures are accurate. The first step is to define a specification of what the counts *should* be, which we can compare against the actual implementation.

We write such a definition in terms of the externally observable behaviour of the contract during the voting process: it receives a sequence of messages and sends confirmation- or error-messages in response. Specifically there are three such events of interests

- The contract processes a vote from a validator
- It processes a new validator set message from an elector, and responds with either a confirmation or an error
- A new block is published and the ticktock function ran.

In the proof assistant we define these message types, and then use the functional model of each contract function to define a `step` function, which t

```

Record message_cprop :=
{
  sig : signature; (*512 bit cell*)
  sign_tag : Integer; (*32 bit Integer*)
  idx : Integer; (* 16 bit Integer*)
  phash : Integer (*256 bit Integer*)
}.

Record message_nextv :=
{
  vst : vset
}.

Inductive message :=
| msg_cprop : forall (s_addr query_id : Integer),
  message_cprop -> message
| msg_nextv : forall (s_addr query_id : Integer)
  (error : bool (* Did the contract reply with an error or not?
  *)),
  message_nextv -> message
| TICK : forall (time : Integer), message.

Record state :Type :=
{
  now : Integer;
  ton_messages : list message;
  config_state : RData
}.

Inductive step : state -> state -> Prop :=
| proceed_register_internal_call_step : forall msgs s_addr query_id m now now' r r',
  proceed_register_internal_call now' s_addr query_id m r = Some r' ->
  now < now' ->
  step { | now := now;
        ton_messages := msgs;
        config_state := r | }
      { | now := now';
        ton_messages := (msg_cprop s_addr query_id m) :: msgs;
        config_state := commit_cfg r' | }
| (* ... Two more cases for validate messages and ticks. ... *)

```

Here the `proceed_register_internal_call` is a short function that models the code in `recv_internal` that handles the voting messages; it is defined in terms of our functional model for the `proceed_register_vote` function. We then define `steps` to mean there is a sequence of zero or more steps between two states. To use programming language terminology, `step` is a small-step description of the contract behaviour, and `steps` is its reflexive-transitive closure. The list of messages `ton_messages` explains exactly what externally observable events happen as the internal data structures of the contract evolves from a state `r` to a state `r'`.

Validator-set invariant of the config contract

Because the voting is done in rounds based on the validator sets. So in order to prove that the voting is done correctly, we first need to establish that it handles the validator set changes in a sensible way.

Specifically, we define a function `calc_epochs` which analyzes a list of events and computes what the validator sets *should* be after processing those messages.

```

Record abs_state := {
  a_cur_vset: vset;
  a_next_vset: option vset;
  a_cur_epoch : list message_cprop;
  a_epochs : list (vset * list message_cprop)
}.

(* Is the time right to install the next vset? *)
Definition next_vset_ready (now : Integer) (vst : vset) :=
  (length_vset vst >=? 40)
  && (utime_since vst <=? now)
  && (tag vst =? 18).

Fixpoint calc_epochs (init_state : abs_state) (mgs : list message) : abs_state :=
  match mgs with
  | nil => init_state
  | msg_cprop s q m :: ms' =>
    let a := calc_epochs init_state ms' in
    { |
      a_cur_vset := a_cur_vset a;
      a_next_vset := a_next_vset a;
      a_cur_epoch := (m :: (a_cur_epoch a));
      a_epochs := a_epochs a |}
  | msg_nextv _ _ true m :: ms' => calc_epochs init_state ms'
  | msg_nextv _ _ false m :: ms' =>
    let a := calc_epochs init_state ms' in
    { |
      a_cur_vset := a_cur_vset a;
      a_next_vset := Some (vst m);
      a_cur_epoch := a_cur_epoch a;
      a_epochs := a_epochs a |}
  | TICK time :: ms' =>
    let a := calc_epochs init_state ms' in
    match a_next_vset a with
    | Some v =>
      if next_vset_ready time v
      then
        { |
          a_cur_vset := v;
          a_next_vset := None;
          a_cur_epoch := nil;
          a_epochs := ((a_cur_vset a), a_cur_epoch a) :: (a_epochs a) |}
      else calc_epochs init_state ms'
    | None => calc_epochs init_state ms'
    end
  end
end.

```

Stated in words, this says that if the contract received an "next validator set" message and did not reply with an error, then at the first block processed after the starting time for that set it should set the current validator set to that message.

The function also partitions the list of vote messages (ordered with the most recent ones at the front) into a list of lists, each one tagged into by the validator set that was in effect as the time. For example, it transforms a list of events

```
(vote e)(TICK)(vote d)(nextv B)(vote c)(vote b)(TICK)(nextv A)(vote a)
```

into a list

```
(B, e) (A, dcb) (INIT_VSET, a).
```

where each sublist of messages is prefixed by the validator set `A`, `B`, ... which was in effect at the time. We call such sublists the "epochs"; they will be useful to define when a proposal should win.

With this definition in place we can define prove the main invariant of the validator set changes:

```
Theorem calc_vset_correct : forall a t t' msgs r r',
  steps {| now := t; ton_messages := nil; config_state := r |}
    {| now := t'; ton_messages := msgs; config_state := r' |} ->
  c7_tuple r = cfg_dic r ->
  cur_vset (c7_tuple r) = Some (a_cur_vset a) ->
  next_vset (c7_tuple r) = a_next_vset a ->
  bounded_abs_state t a ->
  c7_tuple r' = cfg_dic r'
  /\ cur_vset (cfg_dic r') = Some (a_cur_vset (calc_epochs a msgs))
  /\ next_vset (cfg_dic r') = (a_next_vset (calc_epochs a msgs))
  /\ bounded_abs_state t' (calc_epochs a msgs).
```

In words, this says that after processing a set of messages, the validator set stored in the contract `cur_vset (cfg_dic r')` is indeed equal to what the validator set should be according to the specification function `a_cur_vset (calc_epochs a msgs)`. Additionally we prove that the start time of each validator set that occurs in the list of epochs is monotonically increasing and less than the current time `t'`, which proves that no validator set was installed out of order.

Voting invariant of the config contract

We can now define the central part of the config correctness property, which is the specification of the number of times that a proposal *should* win, considering the messages exchanged with the contract.

```

Fixpoint already_voted i ph (ms : list message_cprop) : bool :=
  match ms with
  | nil => false
  | m :: ms' => if (i =? idx m) && (ph =? phash m) then true else already_voted i ph
  ms'
end.

Definition validator_weight i (vs : Dict Validator) : Integer :=
  match dictionary.get i vs with
  | None => 0
  | Some (mkValidator _ _ _ wgt _) => wgt
  end.

Fixpoint proposal_weight (phash0 : Integer) (vs : Dict Validator) (ms : list
message_cprop) : Integer :=
  match ms with
  | nil => 0
  | Build_message_cprop sig _ idx phash :: ms' =>
    let w:= if phash0 =? phash
      then
        if (already_voted idx phash0 ms')
        then 0
        else validator_weight idx vs
      else 0
    in w + proposal_weight phash0 vs ms'
  end.

Definition proposal_win (p : Integer) (vs : vset) (ms : list message_cprop) : bool :=
  proposal_weight p (dict vs) ms >? (total_weight vs) * 3 / 4.

Fixpoint count_wins (p : Integer) (es : list (vset * list message_cprop)) : Integer
:=
  match es with
  | nil => 0
  | (vs, ms) :: es' =>
    if proposal_win p vs ms
    then (count_wins p es') + 1
    else (count_wins p es')
  end.

```

Note that this is defined entirely in terms of the externally observed behaviour of the contract, specifically the list `es : list (vset * list message_cprop)` of messages divided into validator-set epochs. It looks through the history of events to see if a validator has voted before, without any of the imperative state manipulations in the contract.

With this function in place we can define the main invariant of voting process. It says that if the contract data structures start with an initial proposal status `ps`, which then gets updated during the contract execution to a new value `ps'`, the two values are related as follows.

```
Record voting_INV (phash : Integer) (ps ps' : proposal_status) (init_vs :vset)
(init_next_vs : option vset) (vs : vset) (msgs : list message) : Prop := {
  e : list message_cprop;
  es : list (vset * list message_cprop);
  inv_epochs : epochs init_vs init_next_vs msgs = (vs, e) :: es;
  inv_wins : wins ps' = wins ps + count_wins phash ((vs, e)::es);
  inv_weight : effective_weight_remaining vs ps' = (total_weight vs * 3/4 -
proposal_weight phash (dict vs) e);
  inv_hash: has_vote phash e = true -> vset_id ps' = vset_hash vs;
  inv_voters: voters_correct vs e phash (vset_id ps') (voters ps');
  inv_hash_in_list : exists vs0, In vs0 (List.map fst ((vs,e)::es)) /\ vset_id ps'
= vset_hash vs0
}.
```

The point of the relation is the line saying `wins ps' = wins ps + count_wins phash ((vs, e)::es)`; i.e. if the proposal status was initialized to `0`, then after running the contract for a while it will contain the correct value `count_wins phash ((vs, e)::es)`.

In order to be able to prove the `inv_wins` part we also need the other lines of the invariant, which characterize what values the `vset_id`, `voters` and `weight` fields of the proposal status will take. Because the fields are recomputed on demand, it takes some care to explicate exactly what values they will take.

We prove that each of the contract functions preserve this invariant.

What could go wrong?

The actual contract code needs to maintain data structures to track which what validators have already voted and what the total (weighted) vote is, as well as detecting if these are outdated because the validation set has changed, which requires some care. In fact, the `proposal_status` record also includes a `losses` field, so one could also try to define a `count_losses` function and prove that it takes correct values. However, because the proposals are only updated by the voting and the random scan, the value the `losses` field takes is somewhat nondeterministic (finding CON-01).

Assumptions of verification for the config contract

In general we state axioms of some properties for the functions that we didn't fully model, for example saying that the `send_confirmation` function does not modify the contract state (which is evident from inspecting its definition). We also assume that the hash function used to compute validator set hashes is injective (since hash collisions could interfere with the update logic). These assumptions are always satisfied in practice.

For the proof of validator-set invariant we make one more assumption which is less guaranteed, which is that the validator sets are only changed by the elector contract (which should be the case in normal operation). Since the current validator set is part of the configuration, it could also be manually changed by the contract owner or through a change proposal. If the validator set was manually overwritten there would be no checking of the parameters in it, so in particular the result that the start times are monotonically increasing need not hold. We do not think this would break any security properties of the contract, but it could cause the vote counts to no longer match our specification.

External guarantee of the config contract

We can exploit the the invariant to provide a final safety theorem about the `proceed_register_vote` function. It states that when the function is called, either it does not change the configuration at all, or the new configuration was produced by calling `accept_proposal` on a proposal which had sufficiently many votes. This is a precise statement of the guarantees that the config contract offers.

```

Lemma proceed_register_new_config :
  forall msgs m (now now' : Z) r r' r0 vs vs' ph ps
    (e : list (vset * list message_cprop))
    val_weight i s_addr query_id status
    (crtcl min_wins min_t_r max_t_r max_losses min_s_s max_s_s c_p b_p: Z),
  now < now' ->
  proceed_register_vote_spec now (phash m) i val_weight r = Some (status, r') ->
  (cfg_dic r) <> (cfg_dic r') ->
  cur_vset (c7_tuple r) = Some vs ->
  cur_vset (c7_tuple r') = Some vs' ->
  dictionary.get ph (vote_dic r) = Some ps ->
  epochs vs (Some vs') ((msg_cprop s_addr query_id m) :: msgs) = e ->
  get_vote_config_spec crtcl r
    = Some (min_t_r, max_t_r, min_wins, max_losses, min_s_s, max_s_s, c_p, b_p, r0)
->
  count_wins ph e >= min_wins /\
  exists r1 r2 cfg_d p_id p_dict crtcl',
  accept_proposal_spec cfg_d (proposal ps) crtcl' r1
    = Some (cfg_dic r', p_id, p_dict, r2).

```

In words, this states that if the configuration has changed (`(cfg_dic r) <> (cfg_dic r')`), then there was some suitable `ps` which proposed that change, and it has won at least `min_wins` rounds in the pat epochs. The theorem serves as a precise specification; for example we can see that the minimum win number comes from the configuration in effect at the time of the final vote, not the one when the proposal was created.

Proof Summary

The following tables gives an overview of the proof effort. For most of the relevant functions we have written a functional model, but in some cases we can leave it abstract, if the details of the code does not matter. In addition, for many of them we have proven that they preserve the invariants. In cases where there is no proof, this is often because the function is trivial enough to not need it; for the data representation functions (load/store/pack/unpack) one can directly reason about the code by looking at it's definition, so there is no need to prove a lemma. Similarly, many functions in the elector contract do not act on the frozen elections at all, so they do not need a proof about the `MIN_STAKE` invariant.

Function	Model	Accounting Invariant	MIN_STAKE invariant
load_data	- [X]	--	--

Function	Model	Accounting Invariant	MIN_STAKE invariant
store_data	- [x]	--	--
pack_elect	- [x]	--	--
pack_past_election	- [x]	--	--
unpack_elect	-[x]	--	--
unpack_past_election	-[x]	--	--
pack_complaint_status	-[x]	--	--
unpack_complaint_status	-[x]	--	--
pack_complaint	-[x]	--	--
unpack_complaint	-[x]	--	--
parse_complaint_prices	-[x]	--	--
get_complaint_prices	-[x]	--	--
get_current_vset	-[x]	--	--
send_message_back	-[x]	--	--
credits_to	-[x]	-[x]	--
send_confirmation	-[x]	--	--
send_validator_set_to_config	-[x]	--	--
get_validator_conf	-[x]	--	--
return_stake	-[x]	--	--
process_new_stake	-[x]	-[x]	--
process_simple_transfer	-[x]	-[x]	--
unfreeze_without_bonuses	-[x]	-[x]	--
unfreeze_with_bonuses	-[x]	-[x]	--
stakes_sum	-[x]	--	--

Function	Model	Accounting Invariant	MIN_STAKE invariant
unfreeze_all	-[x]	-[x]	--
recover_stake	-[x]	--	--
config_set_confirmed	-[x]	--	--
compute_total_stake	-[x]	--	--
try_elect	-[x]	-[x]	-[x]
announce_new_elections	-[x]	-[x]	-[x]
update_active_vset_id	-[x]	-[x]	-[x]
validator_set_installed	-[x]	-[x]	-[x]
conduct_elections	-[x]	-[x]	-[x]
check_unfreeze	-[x]	-[x]	--
run_ticktock	-[x]	-[x]	--

Function	Model	Vset invariant	Voting invariant
set_conf_param	--	--	--
load_data	-[x]	--	--
store_data	-[x]	--	--
parse_vote_config	-[x]	--	--
get_vote_config_internal	-[x]	--	--
get_vote_config	-[x]	--	--
check_validator_set	-[x]	--	--
send_answer	--	--	--
send_confirmation	--	--	--
send_error	--	--	--
change_elector_code	--	--	--

Function	Model	Vset invariant	Voting invariant
after_code_upgrade	--	--	--
perform_action	--	--	--
get_current_vset	-[x]	--	--
get_validator_descr	-[x]	--	--
unpack_validator_descr	-[x]	--	--
parse_config_proposal	-[x]	--	--
accept_proposal	-[x]	--	--
perform_proposed_action	--	--	--
update_proposal_status	-[x]	--	--
register_vote	-[x]	--	-[x]
proceed_register_vote	-[x]	--	-[x]
scan_proposal	-[x]	--	-[x]
scan_random_proposal	-[x]	--	-[x]
register_voting_proposal	--	--	--
next_validator_step	-[x]	-[x]	--
run_ticktock	-[x]	-[x]	-[x]

VERIFICATION DETAILS

THE OPEN NETWORK - FORMAL VERIFICATION 1 (PHASE 2)

Invariant preservations for elector-code.fc

Verification #1

Code

About the function `credit_to` in the `crypto/smartcont/elector-code.fc`

```
credit_to(credits, addr, amount) inline_ref
```

Specification

```
Lemma credit_to_correct : forall credits k a,  
  exists credits',  
  credit_to_spec credits k a = (Some credits', tt) /\  
  sum_with id credits' = sum_with id credits + a.
```

The code meets the specification.

Verification #2

Code

About the function `(cell, int) unfreeze_without_bonuses` in the ``crypto/smartcont/elector-code.fc`

```
(cell, int) unfreeze_without_bonuses(credits, freeze_dict, tot_stakes)
```

Specification

```
Lemma unfreeze_without_bonuses_correct : forall credits credits' frozen  
  tot_stakes recovered r r',  
  unfreeze_without_bonuses_spec credits frozen tot_stakes r =  
  Some (credits', recovered, r') -> sum_with id credits' + recovered =  
  sum_with id credits + sum_with stake_from_validator frozen.
```

The code meets the specification.

Verification #3

Code

About the function ***unfreeze_with_bonuses*** in the `crypto/smartcont/elector-code.fc`

```
(cell, int) unfreeze_with_bonuses(credits, freeze_dict, tot_stakes, tot_bonuses)
```

Specification

```
Lemma unfreeze_with_bonuses_correct :  
  forall credits credits' frozen tot_stakes bonuses recovered r r',  
    unfreeze_with_bonuses_spec credits frozen tot_stakes bonuses r =  
      Some (credits', recovered, r') ->  
    sum_with id credits' + recovered =  
      sum_with id credits + sum_with stake_from_validator frozen + bonuses.
```

The code meets the specification.

Verification #4

Code

About the function ***process_new_stake*** in the file `crypto/smartcont/elector-code.fc`

```
()process_new_stake(s_addr, msg_value, cs, query_id)
```

Specification

```
Lemma process_new_stake_inv :  
  forall (s_addr : Addr) (msg_val : Integer) (cs : message) (q_id : Integer)  
    (r r' : RData),  
    process_new_stake_spec s_addr msg_val cs q_id r = Some r' ->  
    contract_value r + msg_val >= contract_value r'.
```

The code meets the specification.

Verification #5

Code

About the function ***unfreeze_all*** in the file `crypto/smartcont/elector-code.fc`

```
_ unfreeze_all(credits, past_elections, elect_id) inline_ref
```

Specification

```

Lemma unfreeze_all_inv :
  forall elect (credits credits' : Dict Integer) (p p' : Dict PastElections)
    (grams active_id active_hash : Integer) (elect_id : Integer)
    (unused_prizes : Integer) (r r' r'' r''' : RData),
  RData_wellformed r ->
  load_data_spec r = Some (elect, credits, p, grams, active_id, active_hash, r') -
  >
  unfreeze_all_spec credits p elect_id r' = Some (credits', p', unused_prizes,
  r'') ->
  store_data_spec elect credits' p' (grams+unused_prizes) active_id active_hash
  r'''
  = Some r''' ->
  contract_value r = contract_value r'''.

```

The code meets the specification.

Verification #6

Code

About the function ***process_simple_transfer*** in the file `crypto/smartcont/elector-code.fc`

```
() process_simple_transfer(s_addr, msg_value)
```

Specification

```

Lemma process_simple_transfer_correct : forall s_addr msg_val r r',
  process_simple_transfer_spec s_addr msg_val r = Some r' ->
  contract_value r' = contract_value r + msg_val.

```

The code meets the specification.

Verification #7

Code

About the function ***recover_stake*** in the file `crypto/smartcont/elector-code.fc`

```
() recover_stake(op, s_addr, cs, query_id)
```

Specification

```
Lemma recover_stake_inv :  
  forall op s_addr cs q_id r r',  
    recover_stake_spec op s_addr cs q_id r = Some r' ->  
    contract_value r = contract_value r'.
```

The code meets the specification.

Verification #8

Code

About the function **announce_new_elections** in the file `crypto/smartcont/elector-code.fc`

```
int announce_new_elections(ds, elect, credits)
```

Specification

```
Lemma announce_new_elections_inv :  
  forall (r r' : RData) (now : Integer) (my_address : (Integer * Addr))  
    (e1 : elections) (cr : Dict Integer) (b : bool),  
    cur_elect r = None ->  
    credits r = cr ->  
    announce_new_elections_spec now my_address e1 cr r = Some (b, r') ->  
    contract_value r = contract_value r'.
```

The code meets the specification.

Verification #9

Code

About the function `try_elect` in the file `crypto/smartcont/elector-code.fc`

```
(cell, cell, int, cell, int, int)  
  try_elect(credits, members, min_stake, max_stake, min_total_stake,  
  max_stake_factor)
```

Specification

```

Lemma try_elect_correct :
  forall (credits new_credits : Dict Integer)
    (members : Dict Participant)
    (min_stake : Integer) (max_stake : Integer)
    (min_total_stake : Integer)
    (max_factor_stake tot_weight tot_stake
     numValidators : Integer)
    (new_vset : Dict vinfo)
    (new_frozen : Dict Validator) (r r' : RData)
    (StakeAssum : stake_assum members max_stake),
  try_elect_spec credits members min_stake max_stake min_total_stake
max_factor_stake r =
  Some (new_credits, new_vset ,tot_weight, new_frozen, tot_stake, numValidators,
r') ->
  numValidators <> 0 ->
  sum_with id credits + sum_of_participants members =
  sum_with id new_credits + sum_of_validators new_frozen.

```

The code meets the specification.

Verification #10

Code

About the function **conduct_elections** in the file `crypto/smartcont/elector-code.fc`

```
int conduct_elections(ds, elect, credits)
```

Specification

```

Lemma conduct_elections_correct : forall now r b r',
  rdata_assum r ->
  RData_wellformed r ->
  conduct_elections_spec now r = Some (b, r') ->
  contract_value r' = contract_value r /\ grams r' >= 0.

```

The code meets the specification.

Verification #11

Code

About the function **check_unfreeze** in the file `crypto/smartcont/elector-code.fc`

```
int check_unfreeze()
```

Specification

```
Lemma check_unfreeze_inv : forall (now : Integer) (b : bool) (r r' : RData),
  PastElections_wellformed (past_elects r) ->
  check_unfreeze_spec now r = Some (b, r') ->
  contract_value r = contract_value r'.
```

The code meets the specification.

Verification #12

Code

About the function `***` in the file `crypto/smartcont/elector-code.fc`

```
() run_ticktock(int is_tock)
```

Specification

```
Lemma run_ticktock_inv :
  forall (is_tock now : Integer) (my_address : (Integer * Addr))
    (r r' : RData),
  rdata_assum r ->
  RData_wellformed r ->
  run_ticktock_spec is_tock now my_address r = Some r' ->
  contract_value r = contract_value r'.
```

The code meets the specification.

Verification #13

Code

About the function `update_active_vset_id` in the file `crypto/smartcont/elector-code.fc`

```
int update_active_vset_id()
```

Specification

```
Lemma update_active_vset_id_correct : forall (r r' : RData) (b : bool),
  update_active_vset_id_spec r = Some (b, r') ->
  contract_value r = contract_value r'.
```

The code meets the specification.

Verification #14

Code

About the function ***validator_set_installed*** in the file `crypto/smartcont/elector-code.fc`

```
int validator_set_installed(ds, elect, credits)
```

Specification

```
Lemma validator_set_installed_inv :  
  forall (b : bool) (r r' : RData),  
    validator_set_installed_spec r = Some (b, r') ->  
    grams r >= 0 ->  
    contract_value r = contract_value r' /\ grams r' >= 0.
```

The code meets the specification.

Verification #15

Code

About the function ***check_unfreeze*** in the file `crypto/smartcont/elector-code.fc`

```
int check_unfreeze()
```

Specification

```
Lemma check_unfreeze_inv : forall (now : Integer) (b : bool) (r r' : RData),  
  PastElections_wellformed (past_elects r) ->  
  check_unfreeze_spec now r = Some (b, r') ->  
  contract_value r = contract_value r'.
```

The code meets the specification.

Verification #16

Code

About the function ***run_ticktock*** in the file `crypto/smartcont/elector-code.fc`

```
() run_ticktock(int is_tock)
```

Specification


```

Lemma run_ticktock_inv :
  forall (is_tock now : Integer) (my_address : (Integer * Addr))
    (r r' : RData),
    rdata_assum r ->
    RData_wellformed r ->
    run_ticktock_spec is_tock now my_address r = Some r' ->
    contract_value r = contract_value r'.

```

The code meets the specification.

Verification #17

Code

About the function `try_elect` in the file `crypto/smartcont/elector-code.fc`

```

(cell, cell, int, cell, int, int)
  try_elect(credits, members, min_stake, max_stake, min_total_stake,
max_stake_factor)

```

Specification

```

Lemma try_elect_existence : forall (credits new_credits : Dict Integer)
  (members : Dict Participant)
  (min_stake : Integer) (max_stake : Integer)
  (min_total_stake : Integer)
  (max_factor_stake tot_weight tot_stake
  numValidators : Integer)
  (new_vset : Dict vinfo)
  (new_frozen : Dict Validator) (r r' : RData)
  (StakeAssum : stake_assum members max_stake),
  try_elect_spec credits members min_stake max_stake min_total_stake
max_factor_stake r =
  Some (new_credits, new_vset ,tot_weight, new_frozen, tot_stake, numValidators,
r') ->
  numValidators <> 0 ->
  sum_with stake_from_validator new_frozen >= min_total_stake.

```

The code meets the specification.

Verification #18

Code

About the function `validator_set_installed` in the file `crypto/smartcont/elector-code.fc`

```
int validator_set_installed(ds, elect, credits)
```

Specification

```
Lemma validator_set_installed_past_elects :
  forall (b : bool) (r r' : RData),
    validator_set_installed_spec (*el cr*) r = Some (b, r') ->
    PastElections_wellformed (past_elects r) ->
    grams r >= 0 ->
    PastElections_wellformed (past_elects r').
```

The code meets the specification.

Vset invariant preservations for config-code.fc

Verification #19

Code

About the function `run_ticktock` in the file `crypto/smartcont/config-code.fc`

```
() run_ticktock(int is_tock)
```

Specification

```
Lemma ticktock_step_calc_vset_correct : forall a now now' r r' msgs,
  now < now' ->
  run_ticktock_spec now' r = Some r' ->
  c7_tuple r = cfg_dic r ->
  cur_vset (c7_tuple r) = Some (a_cur_vset (calc_epochs a msgs)) ->
  next_vset (c7_tuple r) = (a_next_vset (calc_epochs a msgs)) ->
  bounded_abs_state now (calc_epochs a msgs) ->
  cur_vset (cfg_dic r') = Some (a_cur_vset (calc_epochs a (TICK now' :: msgs)))
  /\ next_vset (cfg_dic r') = (a_next_vset (calc_epochs a (TICK now' :: msgs)))
  /\ bounded_abs_state now' (calc_epochs a (TICK now' :: msgs)).
```

The code meets the specification.

Verification #20

Code

About the function `recv_internal` in the file `crypto/smartcont/config-code.fc`

```
( ) recv_internal(int msg_value, cell in_msg_cell, slice in_msg)
```

Specification

```
Lemma next_validator_step_calc_vset_correct :
  forall a now now' s_addr query_id m r r' msgs,
    now < now' ->
    next_validator_call now' s_addr query_id m r = Some r' ->
    c7_tuple r = cfg_dic r ->
    ~ is_error r' ->
    cur_vset (c7_tuple r) = Some (a_cur_vset (calc_epochs a msgs)) ->
    next_vset (c7_tuple r) = (a_next_vset (calc_epochs a msgs)) ->
    bounded_abs_state now (calc_epochs a msgs) ->
    cur_vset (cfg_dic r')
      = Some (a_cur_vset (calc_epochs a (msg_nextv s_addr query_id false m ::
msgs)))
    /\ next_vset (cfg_dic r')
      = (a_next_vset (calc_epochs a (msg_nextv s_addr query_id false m ::
msgs)))
    /\ bounded_abs_state now' (calc_epochs a (msg_nextv s_addr query_id false m ::
msgs)).
```

The code meets the specification.

Verification #21

Description

Overall invariant for validator set calculation.

Specification

```
Theorem calc_vset_correct : forall a t t' msgs r r',
  steps { | now := t; ton_messages := nil; config_state := r | }
        { | now := t'; ton_messages := msgs; config_state := r' | } ->
  c7_tuple r = cfg_dic r ->
  cur_vset (c7_tuple r) = Some (a_cur_vset a) ->
  next_vset (c7_tuple r) = a_next_vset a ->
  bounded_abs_state t a ->
  c7_tuple r' = cfg_dic r'
  /\ cur_vset (cfg_dic r') = Some (a_cur_vset (calc_epochs a msgs))
  /\ next_vset (cfg_dic r') = (a_next_vset (calc_epochs a msgs))
  /\ bounded_abs_state t' (calc_epochs a msgs).
```

The code meets the specification.

Voting invariant preservations for config-code.fc

Verification #22

Code

About the function `register_vote` in the file `crypto/smartcont/config-code.fc`

```
(cell, cell, int) register_vote(vote_dict, phash, idx, weight)
```

Specification

```
Lemma register_vote_INV :
  forall now (d d' : Dict proposal_status) init_vs init_next_vs
    vs vs' val_weight r r' accepted s_addr query_id m msgs i ph ps0 ps ps' p,
  register_vote_spec d now ph i val_weight r = Some (d', p, accepted, r') ->
  cur_vset (c7_tuple r) = Some vs ->
  cur_vset (c7_tuple r') = Some vs' ->
  val_weight = validator_weight i (dict vs) ->
  dictionary.get ph d = Some ps ->
  dictionary.get ph d' = Some ps' ->
  idx m = i ->
  phash m = ph ->
  voting_INV ph ps0 ps init_vs init_next_vs vs msgs ->
  voting_INV ph ps0 ps' init_vs init_next_vs vs' ((msg_cprop s_addr query_id m) ::
  msgs).

Lemma register_vote_INV_ineq :
  forall now (d d' : Dict proposal_status) init_vs init_next_vs vs vs' val_weight
    r r' accepted s_addr query_id m msgs i ph ps0 ps ps' p,
  register_vote_spec d now (phash m) i val_weight r = Some (d', p, accepted, r') -
  >
  cur_vset (c7_tuple r) = Some vs ->
  cur_vset (c7_tuple r') = Some vs' ->
  val_weight = validator_weight i (dict vs) ->
  dictionary.get ph d = Some ps ->
  dictionary.get ph d' = Some ps' ->
  idx m = i ->
  phash m <> ph ->
  voting_INV ph ps0 ps init_vs init_next_vs vs msgs ->
  voting_INV ph ps0 ps' init_vs init_next_vs vs' ((msg_cprop s_addr query_id m) ::
  msgs).
```

The code meets the specification.

Verification #23

Code

About the function ***proceed_register_vote*** in the file `crypto/smartcont/config-code.fc`

```
int proceed_register_vote(phash, idx, weight)
```

Specification

```
Lemma proceed_register_vote_INV :
  forall ps0 init_vs init_next_vs msgs s_addr query_id i r r' vs vs' now ph ps ps'
    val_tg val_tg2 cal_addr val_weight stake (banned : bool) status m,
    get i (dict vs) = Some (mkValidator val_tg val_tg2 cal_addr val_weight stake
      banned) ->
    proceed_register_vote_spec now ph i val_weight r = Some (status, r') ->
    cur_vset (c7_tuple r) = Some vs ->
    cur_vset (c7_tuple r') = Some vs' ->
    dictionary.get ph (vote_dic r) = Some ps ->
    dictionary.get ph (vote_dic r') = Some ps' ->
    idx m = i ->
    phash m = ph ->
    voting_INV ph ps0 ps init_vs init_next_vs vs msgs ->
    voting_INV ph ps0 ps' init_vs init_next_vs vs' ((msg_cprop s_addr query_id m) ::
      msgs).
```

```
Lemma proceed_register_vote_INV_ineq :
  forall ps0 init_vs init_next_vs msgs s_addr query_id i r r' vs vs' now ph ps ps'
    val_tg val_tg2 cal_addr val_weight stake (banned : bool) status
    m,
    get i (dict vs) = Some (mkValidator val_tg val_tg2 cal_addr val_weight stake
      banned) ->
    proceed_register_vote_spec now (phash m) i val_weight r = Some (status, r') ->
    cur_vset (c7_tuple r) = Some vs ->
    cur_vset (c7_tuple r') = Some vs' ->
    dictionary.get ph (vote_dic r) = Some ps ->
    dictionary.get ph (vote_dic r') = Some ps' ->
    idx m = i ->
    phash m <> ph ->
    voting_INV ph ps0 ps init_vs init_next_vs vs msgs ->
    voting_INV ph ps0 ps' init_vs init_next_vs vs' ((msg_cprop s_addr query_id m) ::
      msgs).
```

The code meets the specification.

Verification #24

Code

About the function ***scan_proposal*** in the file `crypto/smartcont/config-code.fc`

```
(slice, int) scan_proposal(int phash, slice pstatus)
```

Specification

```
Lemma scan_proposal_INV :  
  forall init_vs init_next_vs msgs now r r' vs ph ps0 ps ps' b,  
    scan_proposal_spec ph ps now r = Some (Some ps', b, r') ->  
    cur_vset (c7_tuple r) = Some vs ->  
    voting_INV ph ps0 ps init_vs init_next_vs vs msgs ->  
    r' = r /\ voting_INV ph ps0 ps' init_vs init_next_vs vs msgs.
```

The code meets the specification.

Verification #25

Code

About the function *scan_random_proposal* in the file `crypto/smartcont/config-code.fc`

```
cell scan_random_proposal(cell vote_dict)
```

Specification

```
Lemma scan_random_proposal_INV :  
  forall init_vs init_next_vs msgs now' d d' r r' vs ph ps0 ps ps',  
    scan_random_proposal_spec d now' r = Some (d', r') ->  
    cur_vset (c7_tuple r) = Some vs ->  
    dictionary.get ph d = Some ps ->  
    dictionary.get ph d' = Some ps' ->  
    voting_INV ph ps0 ps init_vs init_next_vs vs msgs ->  
    r' = r /\ voting_INV ph ps0 ps' init_vs init_next_vs vs (msgs).
```

The code meets the specification.

Verification #26

Code

About the function *run_ticktock* in the file `crypto/smartcont/config-code.fc`

```
() run_ticktock(int is_tock)
```

Specification

```

Lemma ticktock_step_INV_helper :
  forall init_vs init_next_vs msgs now' r r' vs ph ps0 ps ps',
    run_ticktock_spec now' r = Some r' ->
      cur_vset (c7_tuple r) = Some vs ->
        dictionary.get ph (vote_dic r) = Some ps ->
          dictionary.get ph (vote_dic r') = Some ps' ->
            voting_INV ph ps0 ps init_vs init_next_vs vs msgs ->
              voting_INV ph ps0 ps' init_vs init_next_vs vs msgs.

Lemma ticktock_step_INV :
  forall init_vs init_next_vs msgs now now' r r' vs vs' ph ps0 ps ps',
    now < now' ->
      run_ticktock_spec now' r = Some r' ->
        c7_tuple r = cfg_dic r ->
          let a := {| a_cur_vset := init_vs;
                     a_next_vset := init_next_vs;
                     a_cur_epoch := nil;
                     a_epochs := nil |} in
            a_cur_vset (calc_epochs a msgs) = vs ->
              cur_vset (c7_tuple r) = Some vs ->
                cur_vset (cfg_dic r') = Some vs' ->
                  next_vset (c7_tuple r) = a_next_vset (calc_epochs a msgs) ->
                    bounded_abs_state now (calc_epochs a msgs) ->
                      dictionary.get ph (vote_dic r) = Some ps ->
                        dictionary.get ph (vote_dic r') = Some ps' ->
                          voting_INV ph ps0 ps init_vs init_next_vs vs msgs ->
                            voting_INV ph ps0 ps' init_vs init_next_vs vs' (TICK now' :: msgs).

```

The code meets the specification.

■ Safety theorem for config-code.fc

Verification #27

Code

About the function *proceed_register_vote* in the file `crypto/smartcont/config-code.fc`

```
int proceed_register_vote(phash, idx, weight)
```

Specification

```
Lemma proceed_register_new_config :
  forall msgs m (now now' : Z) r r' r0 vs vs' ph ps
    (e : list (vset * list message_cprop))
    val_weight i s_addr query_id status
    (crtcl min_wins min_t_r max_t_r max_losses min_s_s max_s_s c_p b_p: Z),
  now < now' ->
  proceed_register_vote_spec now (phash m) i val_weight r = Some (status, r') ->
  (cfg_dic r) <> (cfg_dic r') ->
  cur_vset (c7_tuple r) = Some vs ->
  cur_vset (c7_tuple r') = Some vs' ->
  dictionary.get ph (vote_dic r) = Some ps ->
  epochs vs (Some vs') ((msg_cprop s_addr query_id m) :: msgs) = e ->
  get_vote_config_spec crtcl r
  = Some (min_t_r, max_t_r, min_wins, max_losses, min_s_s, max_s_s, c_p, b_p, r0)
->
  count_wins ph e >= min_wins /\
  exists r1 r2 cfg_d p_id p_dict crtcl',
  accept_proposal_spec cfg_d (proposal ps) crtcl' r1
  = Some (cfg_dic r', p_id, p_dict, r2).
```


FINDINGS | THE OPEN NETWORK - FORMAL VERIFICATION 1 (PHASE 2)



5

Total Findings

0

Critical

0

Major

1

Medium

2

Minor

2

Informational

This report has been prepared to discover issues and vulnerabilities for The Open Network - Formal Verification 1 (Phase 2) . Through this audit, we have uncovered 5 issues ranging from different severity levels. Utilizing Static Analysis techniques to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
<u>CKP-01</u>	Portion Of Bid Above <code>max_stake</code> Silently Discarded.	Logical Issue	Medium	● Resolved
<u>CKP-02</u>	No Check For Existing Key In <code>announce_new_elections / conduct_elections</code>	Logical Issue	Minor	● Resolved
<u>CKP-03</u>	Not Deducting Message Cost In <code>conduct_elections</code>	Logical Issue	Minor	● Resolved
<u>CKP-04</u>	Unused Function Argument In <code>announce_new_elections()</code>	Inconsistency, Coding Style	Informational	● Resolved
<u>CON-01</u>	<code>losses</code> Is Not Updated If There Are No Votes In A Period	Logical Issue	Informational	● Acknowledged

CKP-01 | FINDING DETAILS

Finding Title

Portion Of Bid Above `max_stake` Silently Discarded.

Category	Severity	Location	Status
Logical Issue	● Medium	projects/ton/crypto/smartcont/elector-code.fc (base): 724	● Resolved

Description

The function `process_new_stake` has a test that the submitted stake is larger than `min_stake`.

```
if (msg_value < min_stake) {  
    ;; stake too small, return it  
    return return_stake(s_addr, query_id, 5);  
}
```

However, nothing prevents a user from submitting a bid larger than `max_stake`.

The function `try_elect` will then restrict the bid to be less than `max_stake`:

```
var (stake, _, pubkey) = (min(key~load_uint(128), max_stake), key~load_uint(32),  
    key.preload_uint(256));
```

If the user submitted more than `max_stake`, then at this point the contract will forget about the excess part, which will become unretrievably locked in the contract balance.

Note that `max_stake` is a configuration parameter which can be changed at any time by the owner of the config contract. This makes the situation potentially more error prone, because such a change could happen between the user's bid submission and the next election. So if the limit was lowered, the user could submit a bid which was valid at the time of submission but triggered this bug at the time of election.

Recommendation

Add an extra test to rule out this situation. If we can assume that the `max_stake` parameter will not change between the calls to `process_new_stake` and `conduct_election`, then adding another if-statement to `process_new_stake` would be sufficient. Otherwise, the `try_elect` function should add the excess Grams to the user's `credits`.

CKP-02 | FINDING DETAILS

Finding Title

No Check For Existing Key In `announce_new_elections` / `conduct_elections`

Category	Severity	Location	Status
Logical Issue	● Minor	projects/ton/crypto/smartcont/elector-code.fc (base): 1010	● Resolved

Description

The `conduct_elections` function computes the next validator set and then stores it in the `past_elect` dictionary. The key it is stored under is the `elect_at` field, the time at which the election takes place. If there somehow was already a past validator set stored under the same key, the past set would be overwritten and all the stakes in it lost.

The `elect_at` field is chosen in `announce_new_elections` to be a time in the future (`now() + elect_begin_before`), so intuitively it will be distinct from other election times in the past. However, there is some extra logic that can move it to a slightly earlier time ("pretend that the elections started at `t0`"). Exactly which time `t0` would be depends on configuration parameters which are set and changed outside the elector contract, which makes it hard to formally prove that there can never be a clash. The likelihood of this going wrong may seem remote, but another check would make the contract more self-contained.

Recommendation

We recommend adding an extra check in `announce_new_elections()` that the chosen `elect_at` time is not already in the `past_elects` dictionary, and return an error if it was. (In that case, the `announce_new_elections()` will be re-run later.)

CKP-03 | FINDING DETAILS

Finding Title

Not Deducting Message Cost In `conduct_elections`

Category	Severity	Location	Status
Logical Issue	● Minor	projects/ton/crypto/smartcont/elector-code.fc (base): 180, 855	● Resolved

Description

There are two cases in the elector contract when it sends a message and includes approximately 1 Gram of value in order to pay for the message processing costs. These are in `process_new_stake` (the call to `send_confirmation`) and in `conduct_elections` (the call to `send_validator_set_to_config`). In `process_new_stake` this cost is deducted from the posted stake, so that the accounting is correct:

```
msg_value -= 10000000000;    ;; deduct GR$1 for sending confirmation
```

However, there is no logic in `conduct_elections` which explains where the value of the message should be deducted from, which breaks the invariant that data structures reflect the current balance of the contract.

Recommendation

Clarify the logic of this function. Decrement the message value from e.g. the `grams` field of the contract.

Alleviation

[TON]: The comment clarifying the `send_validator_set_to_config()` fee source was added:

```
1204    ;; it is expected that initial balance of elector is sufficient to cover
sending of message to config
1205    ;; since elector and config are special contracts they do not pay fees and
thus the whole sum
1206    ;; will be returned back
```

CKP-04 | FINDING DETAILS

Finding Title

Unused Function Argument In `announce_new_elections()`

Category	Severity	Location	Status
Inconsistency, Coding Style	● Informational	projects/ton/crypto/smartcont/elector-code.fc (base): 978~1016	● Resolved

Description

The function `announce_new_elections` takes an argument `elect` despite declaring a local variable with the same name which is used in the function body. The argument is redundant.

Recommendation

Modify the function signature to remove the unused argument.

CON-01 | FINDING DETAILS

Finding Title

`losses` Is Not Updated If There Are No Votes In A Period

Category	Severity	Location	Status
Logical Issue	● Informational	projects/ton/crypto/smartcont/config-code.fc (base): 237	● Acknowledged

Description

The voting configuration includes the parameters `min_wins` and `max_losses`, which are used to initialize new proposals. As we prove in this report, a proposal needs to win at least `min_wins` times in order to pass. If it loses more than `max_losses` times, it may be discarded.

However, the latter is not a hard guarantee. The check for losses happens either when somebody votes for the proposal, or if the proposal is selected by the `scan_random_proposal()` function. If nobody voted on it for a long period of time, where the validator set changed multiple times and the proposal did not happen to be scanned, then the fact that it did not win in that period would not get recorded, so the loss count would be smaller and it could still pass.

Recommendation

Clarify the intended logic of the function. If the `max_losses` is essential, the `update_proposal_status()` function would need some state to keep track of how many validator set changes have happened, not just whether the validator set has changed at all.

Alleviation

[CertiK]: The `losses` is only updated if `scan_random_proposal()` triggers or `register_vote()` for this proposal happens.

APPENDIX X | THE OPEN NETWORK - FORMAL VERIFICATION 1 (PHASE 2)

Finding Categories

Categories	Description
Logical Issue	Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.
Coding Style	Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.
Inconsistency	Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY

KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.

