Celestial: A Smart Contracts Verification Framework

Samvid Dharanikota*§ Suvam Mukherjee*§ Chandrika Bhardwaj† Aseem Rastogi* Akash Lal*

Microsoft Research India
*{t-sadha,t-sumukh,aseemr,akashl}@microsoft.com
†chandrika.bhardwaj@gmail.com

***Abstract*—We present CELESTIAL, a framework for formally verifying smart contracts written in the Solidity language for the Ethereum blockchain. CELESTIAL allows programmers to write expressive functional specifications for their contracts. It translates the contracts and the specifications to F⋆ to formally verify, against an F⋆ model of the blockchain semantics, that the contracts meet their specifications. Once the verification succeeds, CELESTIAL performs an erasure of the specifications to generate Solidity code for execution on the Ethereum blockchain. We use CELESTIAL to verify several real-world smart contracts from different application domains such as tokens, digital wallets, and governance. Our experience shows that CELESTIAL is a valuable tool for writing high-assurance smart contracts.**

***Index Terms*—blockchain, smart contracts, verification**

I. INTRODUCTION

Smart contracts are programs that enforce agreements between parties transacting over a blockchain. Till date, more than a million smart contracts have been deployed on the Ethereum blockchain with applications such as digital wallets, tokens, auctions, and games. As of September 2020, Ethereum smart contracts hold digital assets worth over $40 billion [31], making them as safety-critical as aviation or medical devices.

The most popular language for smart contract development is Solidity [32]. A contract in Solidity is akin to a class definition in an object-oriented language, with fields and methods. Solidity contracts are compiled to Ethereum Virtual Machine (EVM) bytecode for execution on the blockchain.

Unfortunately, this existing ecosystem is not suitable for programming correct and secure smart contracts. Solidity has obscure operational semantics understood only partially by most programmers. This often leaves vulnerabilities in the smart contracts. Repeated high-profile attacks (e.g. `TheDAO` [28] and `ParityWallet` [29] attacks) orchestrated around these vulnerabilities have resulted in financial losses running into millions of dollars. Worse, smart contracts are "burned" into the blockchain on deployment, which does not allow subsequent patches to fix the vulnerabilities. As a result, it is necessary to ensure at the time of deployment that the smart contracts are bug-free and they correctly implement their intended functionality.

On the other hand, smart contracts are relatively small pieces of code with simple data-structures [40]. All these qualities combined—their critical nature, immutability after deployment, and small size—make smart contracts a good fit for formal verification. Formally proving that the contracts satisfy properties of interest would provide high-assurance before they are deployed. However, the challenge is to lower the formal verification entry barrier for smart contracts developers.

Towards that goal, we present CELESTIAL, a framework for developing formally verified smart contracts. CELESTIAL allows programmers to annotate their Solidity contracts with Hoare-style specifications [43] capturing functional correctness properties. The contracts and the specifications are translated to F⋆ [59], which in an automated manner, proves that the contracts meet their specifications. Once F⋆ returns a verified verdict, CELESTIAL erases the specifications from the input contracts, and emits verified Solidity code that can be deployed and executed on the Ethereum blockchain as usual. By using Solidity as the source language, and providing fully-automated verification, CELESTIAL ensures a low entry barrier for smart contracts programmers to enjoying the strong guarantees of formal verification.

F⋆ is a proof assistant and program verifier with a fully dependent type system. We find it suitable for smart contract verification for several reasons. First, it provides SMT-based automation which, as we show empirically, suffices for fully-automated verification of real-world smart contracts. Second, F⋆ supports user-defined effects, allowing us to work in a custom state and exception effect [33] modeling the blockchain semantics. Finally, F⋆ supports expressive higher-order specifications, though we use its first-order subset with quantifiers and arithmetic (adding our own libraries for arrays and maps).

We evaluate CELESTIAL by verifying several real-world Solidity smart contracts, holding millions of dollars of financial assets. The contracts span different application domains including ERC20 tokens, multi-signature wallets, and a governance protocol for a large cloud service provider. We studied the contracts (and in some cases discussed with the developers) to design their functional correctness specifications and formally verified that the contracts meet those specifications. In the process, we uncovered bugs in some cases (e.g. missing overflow checks), manifesting as F⋆ verification failure. Once we fixed those bugs (e.g. by adding runtime checks), F⋆ was able to successfully verify the contracts in all the cases. We also measure the overhead introduced due to such additional instrumentation by comparing the gas consumption of the CELESTIAL and the original Solidity versions of the contract. Across all of our experiments, we observed a maximum 20% overhead.

§Equal contribution

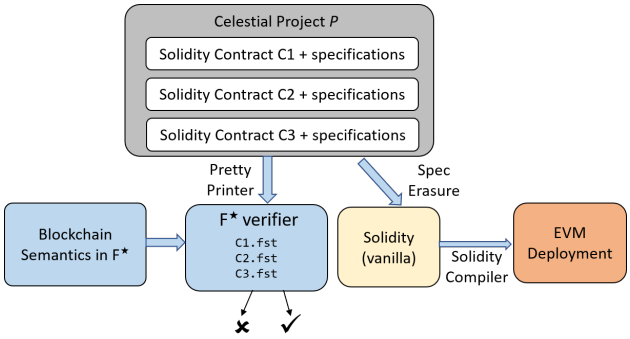

Fig. 1: Architecture of the CELESTIAL framework. The trusted components are filled with blue.

Summarizing our main contributions:

1) We present CELESTIAL, a framework for developing formally verified smart contracts for the Ethereum blockchain. The framework allows programmers to annotate their Solidity contracts with functional correctness specifications, that are verified, in an automated manner, using F*.
2) We evaluate CELESTIAL by verifying functional correctness properties for several real-world, high-valued smart contracts from different application domains.

Our experience with CELESTIAL leads us to believe that we can make automated formal verification accessible to the smart contract developers. Given their critical nature and immutability post-deployment, we certainly hope that formal verification becomes a pre-requisite for developing and deploying high-assurance smart contracts.

Related Work: There has been substantial research focused on detecting defects in smart contracts. Existing solutions include publishing surveys of known vulnerabilities [34] and suggesting programming best practices [11], [21]. A range of static analysis tools (such as `Slither` [37], `Securify` [60], `Zeus` [45], `Oyente` [50] and `Mythril` [15]) are geared towards identifying smart contract vulnerabilities in an automated fashion. However, these analyzers typically target known low-level properties, such as reentrancy, overflow/underflow, and gas exceptions. This results in two issues: the programmer is left on her own to defend against unknown attacks, and these tools do not provide functional correctness guarantees. These issues have spurred an industry of startups focused on formal verification of smart contracts, such as `Certora` [4] and `VerX` [24], [56]. We discuss these, and additional related work, in detail in Section VI.

The rest of this paper is organized as follows. Section II provides an overview of the CELESTIAL framework. Section III describes the formal verification backend. In Section IV, we discuss implementation details of the CELESTIAL framework, and we discuss experimental results in Section V. We conclude in Section VII.

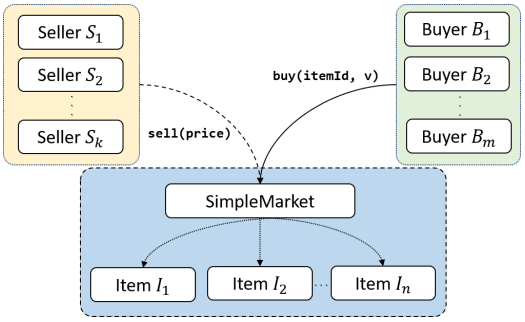

Fig. 2: A simple blockchain based e-commerce application.

II. OVERVIEW

The high-level architecture of the CELESTIAL framework is outlined in Figure 1. A CELESTIAL project contains a set of contracts (e.g. C1, C2, and C3 in the figure) written in Solidity. Additionally, these contracts may be annotated with functional specifications encoding properties of interest. CELESTIAL provides two kinds of translations for these contracts. The first one translates the contracts and their specifications to F* [59], a dependently-typed functional programming language designed for program verification. F*, using a model of the blockchain semantics (Section III), verifies that the contracts meet their specifications. Once F* returns a "verified" verdict, a second translation erases all the specifications to emit vanilla Solidity contracts for execution on the Ethereum blockchain. In this section we use a simple application (Section II-A) to describe the specification language of CELESTIAL (Section II-B). We discuss the verification scope and limitations of the framework later in Section II-C.

A. SIMPLEMART

Consider a simple blockchain-based e-commerce application SIMPLEMART from Figure 2. The application contains a `SimpleMarket` contract (Listing 1) which interacts with one or more buyers and sellers that may either be smart contracts themselves or externally-owned accounts. A seller registers an item for sale by invoking the `sell` method of `SimpleMarket`, passing the price of the item as an argument. In response, `SimpleMarket` creates an instance of the `Item` contract, which holds various metadata about the new item available for sale. It also emits an event (`eNewItem`) informing the seller about the idenity (in this case, the address) of the new item. A buyer may purchase an item by invoking the `buy` method of `SimpleMarket`, passing the item address as an argument, along with the ether amount matching the item price. If the item has not been sold already, `SimpleMarket` records the sale in its state, which involves adding the ether towards the total sales proceeds for the respective seller and marking the item as being sold. The seller may then withdraw the ether from `SimpleMarket` by invoking its `withdraw` method.

For the properties of interest, consider the `buy` method of the `SimpleMarket` contract. For the functional correctness of

```
contract SimpleMarket {
  mapping(address => uint) sellerCredits;
  mapping(address => Item) itemsToSell;
  uint totalCredits;
  event eNewItem (address, address);
  event eItemSold (address, address);

  function sell (uint price) public
    returns (address itemId) {
    Item item= new Item(address(this), sender, price);
    itemId = address(item);
    itemsToSell[address(item)] = item;
    emit eNewItem(sender, itemId);
  }

  function buy (address itemId)
    public payable {
    Item item = itemsToSell[itemId];
    if (item == null) { revert ("No such item"); }
    if (value != item.getPrice())
    { revert ("Incorrect price"); }
    address seller = item.getSeller();
    totalCredits = safe_add (totalCredits, value);
    sellerCredits[seller] =
      safe_add (sellerCredits[seller], value);
    delete (itemsToSell[itemId]);
    emit eItemSold(buyer, itemId);
  }

  function withdraw (uint amount) public {
    if (sellerCredits[sender] >= amount) {
      sender.transfer(amount);
      sellerCredits[sender] -= amount;
    } else {  revert ("Insufficient balance"); }
  }
}
```

Listing 1: The `SimpleMarket` Solidity contract

```
contract Item {
  address seller; uint price; address market;
  function getSeller () returns (address)
    modifies []
    post (ret == seller)
  { return seller; }
  // other methods
}
contract SimpleMarketplace {
  // contract fields
  ...
  invariant balanceAndSellerCredits {
    balance >= totalCredits &&
    totalCredits == sum_mapping (sellerCredits)
  }
  function buy (address itemId) public
    modifies [sellerCredits, totalCredits, itemsToSell,
        log]
    tx_reverts !(itemId in itemsToSell)
              || value != itemsToSell[itemId].price
              || value + totalCredits > uint_max
    post (!(itemId in itemsToSell)
         && sellerCredits[seller] == old(sellerCredits)[
           seller => sellerCredits[seller] + value]
         && log == (eItemSold, sender, itemId)::old(log))
  { // implementation of the buy function }
}
```

Listing 2: `Item` and `SimpleMarket` CELESTIAL contracts

this method, we would like to formally verify that if a buyer initiates buy with a valid item and price, then the item is sold and the seller sales proceeds are credited, leaving all other sellers' proceeds unchanged. In addition, we would also like to verify that the call does not result in arithmetic overflow of the seller's proceeds, since this can result in honest sellers losing their credits.

```
contract A {
  uint x, y; // fields, as usual

  invariant { φ1 } //contract-level invariant

  function foo () public
    modifies [x] //fields that are modified
    tx_reverts φ2 //revert condition (under-specified)
    pre φ3 //precondition
    post φ4 //postcondition
  { s } //implementation
}
```

Listing 3: A representative CELESTIAL contract

B. Specification Language

A CELESTIAL contract is Solidity code annotated with logical specifications. Listing 2 shows excerpts of the CELESTIAL versions of `Item` and `SimpleMarket` contracts.

The general form of a CELESTIAL contract is shown in Listing 3. These annotations are Hoare-style specifications, similar to languages like Dafny [48]. The specifications are written over the contract fields, function arguments, as well as implicit variables such as `balance` (the contract balance), `value` (ether value in a payable method), and `log` (the transaction event log, modeled as a list of events). Our specifications cover the full power of first-order reasoning with quantifiers, along with theories for arithmetic (both modular and non-modular), arrays and maps. We provide programmers the ability to write pure functions that can be invoked only from specifications, not Solidity methods, to enable code reuse. We now explain the individual elements of the specifications.

a) Contract invariant: Contract invariant is a predicate on the state of the contract (i.e. its field values) that is expected to be valid at the boundaries of its public methods. When verifying a contract, the invariant is added to the pre- and postconditions of every public method, meaning that every public method assumes the invariant when it begins execution and (through formal verification) ensures the invariant upon exit. All the contract fields in a CELESTIAL contract are necessarily private (see Section II-C). Thus, any code that is not in the contract itself cannot change the contract state without calling its methods, all of which ensure the invariant before returning. Additionally, CELESTIAL ensures that all its contracts are *external callback free* (Section IV-C) to disallow re-entrancy based attacks from external contracts. Hence, it is safe to assume the invariant at the beginning of public methods. Constructors are special; they only guarantee invariant in their postcondition but don't assume it as a precondition. For example, the invariant on line 4 in Listing 2 specifies that the contract's `balance` equals or exceeds the total proceeds from sales which has not been already claimed by the respective sellers (`sum_mapping` is a library function for summing values in an int-valued map). Note that the verification will fail if we make this invariant an equality, since we cannot guarantee that the contract is deployed to an address which does not already have some balance.

b) Field updates: The `modifies` clause allows programmers to precisely specify contract fields that a method

may update, conversely the contract fields that the method must not change. The `getSeller` method in `Item` has an empty `modifies` clause (line 4 in Listing 2), which specifies that the function may read the state of the contract, but cannot make any updates. In Listing 2, the `modifies` clause in line 17 specifies that the method `buy` updates both the `sellerCredits` and `itemsToSell` maps, the `totalCredits` field, and the event `log`.

c) Pre- and postconditions: Method preconditions, specified using the `pre` keyword, are properties that are assumed to hold at the beginning of a method execution. Public methods must have a trivial precondition `true`; indeed it is impossible to enforce anything stronger on method arguments because they are supplied by the untrusted external world. The contract `invariant` can, however, be safely assumed as a precondition as mentioned before. Postconditions (specified via `post`) are properties that must hold when the method terminates successfully, that is, without reverting the transaction. The postconditions may refer to field values at the beginning of the method using the `old` keyword. For example, the condition in line 22 in Listing 2 specifies that the final `sellerCredits` is the original `sellerCredits` map with only the `seller` key updated.

d) Revert conditions: `tx_reverts` under-specifies the conditions under which a method reverts, i.e. if `tx_reverts` holds at the beginning of a method, the method will definitely revert. For example, the `buy` function definitely reverts if the buyer invokes it with an item which is not available for sale, or the buyer provides ether which does not match the item price, or the `totalCredits` overflows. This is captured in the spec in line 18. Not specifying `tx_reverts` is equivalent to `tx_reverts(false)`.

e) Conditions for safe arithmetic: In Solidity, arithmetic operations may silently over- or underflow, whereas division by 0 results in reverts. CELESTIAL, when translating to F⋆, adds assertions before every arithmetic operation which check for no over- and underflows, and division by 0. When these assertions fail, the programmer has to add specs or runtime checks that allow the verifier to prove the safety of the arithmetic operations. They may use the CELESTIAL library for safe arithmetic that has built-in runtime checks, for example, the `safe_add` operation in line 23 of Listing 1. This library is similar to OpenZeppelin SafeMath [17] and provides both the implementation and associated specifications of safe arithmetic.

Summarizing `buyPost` from Listing 2, we can see that using the CELESTIAL specifications, we have expressed the properties of interest. The revert condition specifies that the method reverts if the item is not present or the ether sent by the buyer does not match the item price. The method also reverts when `totalCredits` overflows. Since an invariant of the contract is that `totalCredits` is the sum of pending credits of all the sellers, when `totalCredits` does not overflow, individual seller credits also don't overflow. Finally, line 22 specifies that only the item seller's credits are incremented by price of the item, while credits for all other sellers remain same.

C. Verification Scope and Limitations

a) Threat model: All contracts and user accounts that are not part of a CELESTIAL project P are treated as the *external world* in the context of P contracts. The external world is free to initiate arbitrary transactions by calling public methods of the P contracts. The methods may be invoked in any order, and with arbitrary arguments. The external world, however, cannot access the private fields and methods of P contracts; indeed the underlying platform EVM ensures this. Through formal verification, CELESTIAL guarantees that the specifications for P contracts hold at runtime, even when interacting with such an unconstrained external world.

b) Trusted Computing Base: The TCB of CELESTIAL includes the CELESTIAL compiler that mainly consists of the two syntax translations, the F⋆ model of the blockchain (Section III), the F⋆ toolchain itself, and the Solidity compiler (these components are colored blue in Figure 1). As mentioned in the threat model, we do not make any assumptions about the external world. We leave it as future work to minimize trust on our F⋆ blockchain semantics by extracting an executable semantics from it (using the usual F⋆ extraction [12]) and validating it against the Solidity testsuite.

c) Solidity Language Restrictions: CELESTIAL does not support the `delegatecall` Solidity instruction. This instruction is used to call functions from other contracts in a way that the callee may directly change the state of the calling address, thereby breaking the function call abstraction. Since this is an inherently insecure feature (for example, the `ParityWallet` [29] attack exploited it), the secure development recommendations suggest against its use [5]. CELESTIAL also does not support embedding EVM assembly since our F⋆ model does not cover assembly and its interop with Solidity. We could allow the programmers to write trusted specifications for assembly blocks and treat them opaquely; we leave such an extension for future work. To check the prevalence of these features in real-world contracts, we performed an empirical study; we present the findings in Appendix A. In summary, we found that not more than one-third of highly used and highly valued contracts use these features, and even then in controlled manner where their usage is restricted to a small set of libraries.

Recall that the `SimpleMarket` contract from Listing 1 interacts with the sellers and buyers by emitting events (lines 13 and 27), instead of using the Solidity `call` construct to invoke their functions. This is indeed the suggested programming practice when interacting with the external world. For example, the `bid` function in the `SimpleAuction` contract in Solidity documentation [19] recommends the same style. Unsafe uses of `call` may result in unexpected reentrancy, which has also resulted in high-profile attacks (see, for example, the `TheDAO` attack [34]). When it is unavoidable, CELESTIAL allows the use of `call`, but requires the programmers to additionally prove that doing so will not result in arbitrary reentrancy; we detail this in Section IV. CELESTIAL also

restricts the transfer of ether to occur only via the Solidity `transfer` construct (line 32 in Listing 1), which ensures that the called contract has insufficient gas to make a reentrant function call.

d) Modeling Limitations: Our F* semantics, presented next, does not model gas consumption. As a result, CELESTIAL contracts may revert due to out-of-gas exceptions. The model also does not cover low-level failures such callstack depth overflow. However, these failures can only cause the transaction to revert and therefore do not compromise the verification guarantees. Since we do not model all runtime exceptions (out-of-gas, stack depth overflow, etc.), this is one of the reasons that the `tx_reverts` condition for a function is an under-specification for when the function may revert. We also do not precisely model the block-level parameters such as the timestamp. Finally, we follow prior work and assume that `balance` never overflows [53].

III. VERIFYING CELESTIAL CONTRACTS IN F*

CELESTIAL compiles the contracts and their specifications to F*, which are then verified against a trusted F* library modeling the blockchain semantics. Once F* verification succeeds, the translated contracts are guaranteed to be safe (as per their specifications). And since we trust the F* blockchain semantics to be sound with respect to the actual blockchain semantics, the safety guarantees carry over to the corresponding Solidity contracts.

Our blockchain semantics in F* consists of the definition of the blockchain state datatype and a custom F* *effect* that encapsulates this state behind the abstraction of an effect layer. The contracts call the stateful API exported by the library and specify precise changes to the blockchain state in their pre- and postconditions, that are verified by F*. In this section, we first explain the model, and then describe the translation of the CELESTIAL contracts to F*.

A. Blockchain state

We model the blockchain state as consisting of 3 main elements: (a) state of all the contracts (i.e. values of the contract fields), (b) contract balances, and (c) an event log. Since all CELESTIAL contract fields are private, a contract can only directly read or write its own fields, while interacting with the other contracts through method calls. The event log models the per-transaction event log of the Ethereum blockchain; contracts can use the Solidity emit API to output events to this log.

a) Contracts state: We model the state of all the contracts in the blockchain as a heterogeneous map from addresses to records, where the record corresponding to a contract instance contains the values of all its fields. For the `Item` contract from Listing 2, the record type would be:

```
type item_t = { market : address; seller : address; price : uint }
```

Below is the API provided by the contract map (# parameters are implicit parameters inferred by F* at the call sites):

```
type address = uint (* 256 bit unsigned integers *)
val contract (a:Type) : Type (* a is the record of contract fields *)
val cmap : Type (* the heterogeneous contracts map *)

val addr_of (#a:Type) (c:contract a) : address
val live (#a:Type) (c:contract a) (m:cmap) : prop
val sel (#a:Type) (c:contract a) (m:cmap{live c m}) : a
val create (#a:Type) (m:cmap) (x:a) : contract a & cmap
val upd (#a:Type) (c:contract a) (m:cmap{live c m}) (x:a) : cmap
```

The API defines the type address as 256 bit unsigned integers. The contract type is parametric over the record type a that contains all the contract fields (e.g. item_t). Type cmap is the heterogeneous contracts map type.

The sel function returns the a-typed record value mapped to a contract instance in the map. The API requires that the contract be live in the map (type m:cmap{live c m} is a refinement type that requires that the m argument at the call sites satisfies live c m). The liveness requirement basically says that the contract must be present in the contracts map, preventing sel to be called with arbitrary addresses. The create function returns the freshly created contract and the new cmap that includes a mapping for the new contract, internally assigning a fresh address to the new contract. We elide the implementation of the API for space reasons; all of our development is submitted as supplementary materials.

b) Contracts balance: We model the contracts balance using a map from addresses to uint (the type of 256-bits unsigned integers). An alternative would have been to add balance as another one of the contract fields (thus maintaining them as part of the contracts map), but a separate map allows us to specify the balances for external accounts, that do not have an entry in the contracts map.

c) Event log: The event log is a list of events (a:Type & a is a dependent tuple of a Type and a value of that type):

```
type event = { to : address; ev_typ : string; payload : (a:Type & a) }
type log = list event
```

With these components, the blockchain state is the following record type:

```
type bstate = { cmap : cmap; balances : Map.t address uint; log : log }
```

B. Libraries for arrays and maps

We have implemented libraries for Solidity arrays and maps in F*. Our current implementation only supports dynamically-sized arrays for now, support for compile time fixed-sized arrays is future work. The libraries export operations that match the corresponding Solidity API, and several lemmas that enable the contracts to reason about their properties. For example, following is a snippet of our array library:

```
val array (a:Type) : Type
val push (#a:Type) (s:array a{length s < uint_max}) (x:a) : array a
val push_length (#a:Type) (s:array a{length s < uint_max}) (x:a)
  : Lemma (requires ⊤) (ensures (length (push s x) == length s + 1))
```

C. An F* effect for contracts

Having set up the model for the blockchain state, we now add a layer on top so that contracts may manipulate the state and precisely specify the modifications in pre- and

postconditions. We leverage the type-and-effect system of F* for this purpose.

F* distinguishes value types such as uint from *computation types*. Computation types specify the effect of a computation, its result type, and optionally some specifications (e.g. pre- and postconditions) for the computation. For example, Tot uint classifies pure, terminating computations that return an uint value. Similarly uint → Tot uint is the type of pure, terminating functions that take a uint argument and return a uint result. uint → uint is a shorthand for uint → Tot uint; all the blockchain state functions above have an implicit Tot effect.

Following Ahman et al. [33], a state and exception effect for computations that operate on mutable state and may throw exceptions is as follows (st is the type of mutable state):

```
type result (a:Type) =
  | Success : x:a → result a
  | Error : e:string → result a

effect STEXN st a (pre:st → prop) (post:st → result a → st →
prop) = ...
```

The semantics of the computations in the STEXN effect may be understood as follows: a computation e of type STEXN a pre post when run in an initial state s_0 satisfying pre s_0, terminates either by throwing an exception (modeled as returning an Error-valued result) or by returning a value of type a (modeled as returning Success-valued result). In either case, the final state s_1 is such that post s_0 r s_1 holds, where r is the return value of the computation. F* also supports divergent effects, in which case the computations are also allowed to diverge.

a) Customizing STEXN for contracts: Contract computations naturally fall into the state and exception effect; they read from and write to the mutable blockchain state, and they may throw an exception by calling revert.

However, the revert operation in Ethereum is slightly different from exceptions in, say, OCaml in that it also reverts the underlying state to what it was at the beginning of the transaction, while in OCaml, the state changes are retained. To accommodate this, we instantiate the state st in STEXN above with

```
type st = { tx_begin : bstate; current : bstate }
```

where the field tx_begin records the state at the beginning of a transaction. Contracts modify the current state, unless they revert, in which case the current state is reset to tx_begin.

```
(* state + exception with st as the state *)
effect ETH (a:Type) (pre:st → prop) (post:st → result a → prop) =
  STEXN a st pre post
```

Using ETH effect, the APIs for begin_transaction, revert, and commit_transaction are as follows:

```
let begin_transaction () : ETH unit (λ _ → ⊤) (λ s0 r s1 →
  is_success r ∧ s0 == s1) = () (* no op *)

let revert () : ETH unit (λ _ → ⊤) (λ s0 r s1 →
  is_error r ∧ s1 == {s0 with current = s0.tx_begin}) = ...

let commit_transaction () : ETH unit (λ _ → ⊤) (λ s0 r s1 →
  is_success r ∧ s1 == {s0 with tx_begin = s0.current}) = ...
```

```
let buy self sender value now i_addr : Eth unit
  (λ s → live self s ∧ invariant self s) (* precondition *)
  (λ s → (* revert condition *)
    let items, scredits = s.[self].items, s.[self].seller_credits in
    ¬items `contains` i_addr ∨ value ≠items[i_addr].price ∨
    sum_mapping scredits + value > uint_max)
  (λ s0 _ s1 → (* postcondition *)
    live self s1 ∧ invariant self s1 ∧
    let item = s0.[self].items[i_addr] in
    s1.[self].credits = add_credits s0.[self].credits item.seller value ∧
    balance self s1 == balance self s0 + value ∧
    ¬s1.[self].items `contains` i_addr)
= let items = get_items self in
  let item = Map.sel items item in
  if item = null then revert ``Item does not exist'';
  let price = Item.get_price item sender now in
  if price ≠value then revert ``Price does not match'';
  record_sale item sender
```

Fig. 3: F* translation of the buy function

The function begin_transaction is a no-op, its precondition is trivial (⊤), while its postcondition states that it does not revert (is_success r) and it leaves the state unchanged (s0 == s1). revert, on the other hand, returns an error value, and its output state s1 is same as its input state s0 with current component replaced with s0.tx_begin, i.e. the state at the beginning of the transaction. commit_transaction does the opposite, it replaces the tx_begin component with s0.current to commit the current state.

The function to get the current state for a contract is as follows, note that the contract is selected from the current component of the state:

```
let get_contract (#a:Type) (c:contract a) : ETH a
  (λ s → live c s.current.cmap)
  (λ s0 x s1 → x == Success (sel c s.current.cmap) ∧ s0 == s1) = ...
```

Similarly, the library provides functions send to transfer balance to a contract and emit to emit an event to the event log. We leave their details to the supplementary materials.

To make our specifications easier to read and write, we define the following effect abbreviation:

```
effect Eth (a:Type) (pre:bstate → prop) (revert:bstate → prop)
  (post:bstate → a → bstate → prop)
  = ETH a (λ s → pre s.current)
      (λ s0 r s1 →
        (revert s0.current ⟹ Error? r) ∧
        (Success? r ⟹ post s0.current (Success?.x r) s1.current))
```

The pre- and postconditions in the Eth effect are written over the current blockchain state (bstate), as opposed to over the st record. Further, the postcondition is a predicate on a value of type a–it only specifies what happens when the contract function terminates successfully. The revert predicate is a predicate on the input state, which if valid means that the function reverts. We find this abbreviation well-suited for our examples, providing the full-flexibility of the ETH effect to the programmers is of course possible.

D. Compilation to F*

CELESTIAL translates the contracts and specifications into F* computations that have Eth effect. Figure 3 shows a

simplified version of the compiled buy function from the SIMPLEMART contract in F⋆. We explain important elements of the translation below.

a) One F⋆ module per contract: Each CELESTIAL contract translates to an F⋆ module. Methods are translated to effectful Eth functions, and specification functions are translated to PURE F⋆ functions. Every module generated for a CELESTIAL contract defines a record type containing all the fields of the contract. The translation also generates (private) getters and setters for each of the fields. For example, following is the generated get_price function for the `Item` contract:

```
type item_t = { market : address; seller : address; price : uint }
type item_contract = contract item_t
let get_price (self:item_contract) = let t = get_contract self in t.price
```

where the contract type is the type defined by the contracts map library as in Section III-A and get_contract is the library function from Section III-C.

b) Translation of types: Translation for Solidity types into F⋆ is straightforward. Base types boolean, unsigned integers, strings translate to their counterparts in F⋆. Arrays and maps translate to corresponding libraries in F⋆ that we have developed to match the Solidity semantics. A contract type A in Solidity translates to A_contract in F⋆.

c) Function arguments: F⋆ translation of each public function has explicit parameters for self and sender. The value parameter is added to payable functions in order to capture the ether amount sent as part of the function call. Our verification only models transaction-level properties and not block-level properties. As a result, block parameters, such as the (underspecified) now timestamp are also explicitly added to the functions as arguments. After these, the function-specific arguments follow.

d) Preconditions: The translation of every public function gets to *only* assume the liveness of self and the contract invariant in the initial blockchain state. Indeed, since these functions can be called by arbitrary, non-verified code, we cannot expect the callers to ensure more sophisticated preconditions on the arguments. If the function requires some other preconditions, they have to be checked at runtime.

e) Postconditions: Similarly, the translation of every public function ensures the liveness of self and the contract invariant in its postcondition. This also justifies the invariant assumption in the precondition. Since all the contract fields in CELESTIAL contracts are private, any code that is not in the contract itself, cannot directly change the contract state without calling some of its functions, all of which ensure the invariant before returning. In addition, the contract may ensure other postconditions, e.g. using the add_credit function, buy ensures that the credits of the seller are incremented by value while the credits of all the other sellers remain same.

f) Revert conditions: Revert conditions are translated as is, e.g. buy reverts when either (a) i_addr is not in its items map, or (b) the passed ether does not match the item price, or (c) adding item price to the seller credits overflows.

g) Function definitions: The translation of a function body uses the per-field getters and setters systematically. Reading a field f translates to get_f and similarly writing fields translates to set_f. Calls to public functions of other contracts are translated to calls to corresponding functions in other F⋆ modules (contracts), e.g. Item.get_price in Figure 3. Library calls to arrays, maps, etc. translate to corresponding libraries calls in F⋆, that we have designed to match the Solidity semantics.

IV. IMPLEMENTING CELESTIAL

A. The CELESTIAL framework

We use ANTLR [1], [54] to parse CELESTIAL contracts and generate abstract syntax trees (ASTs). Each of the syntax translators highlighted in Figure 1 are Python scripts that traverse these ASTs and produce the target code (either F⋆ or Solidity). The translators to F⋆, for specifications as well as implementation, are combined 2300 lines of Python code. The spec-erasing translator to Solidity is even simpler, being just about 750 lines of Python code. The blockchain model is around 1200 lines of F⋆ code. To aid developer experience, we have written a plugin for Visual Studio Code [25] that supports full syntax highlighting for CELESTIAL, along with scripts which automate F⋆ and Solidity code generations and verification.

B. Implementation Limitations

We focused our implementation efforts on Solidity constructs used in our case studies. Fundamentally we disallow `delegatecall`, embedded assembly and uncontrolled reentrancy. Our implementation additionally leaves out syntactic features such as inheritance, abstract contracts and tuple types. These mostly just provide syntactic sugar that should be easy to support in future versions of CELESTIAL. Our implementation currently also does not support passing arrays and structs as arguments to functions.

While our implementation allows loops in contract functions, we currently do not support writing loop invariants. We also only provide weak specifications for block level constructs (such as `timestamp`, `number` and `gaslimit`), transaction level constructs (such as `origin` and `gasprice`), and functions for obtaining hashes (such as `keccak256` and `sha256`).

C. Contract Local Reasoning

Calling external contracts can lead to *reentrant* behavior where the external contract calls back into the caller. It is often non-intuitive to reason about reentrant behaviors. CELESTIAL disallows such behaviors by checking for *external callback freedom* (ECF) [39], [56]. This property states that every contract execution that contains a reentrant callback is *equivalent* to some behavior with no reentrancy. When this property holds, it is sufficient to reason about non-reentrant behaviors only: any specification over those set of behaviors will hold for all behaviors as well. Thus, ECF allows for contract-local reasoning: a programmer can use CELESTIAL to write and prove specifications of their contracts without assuming anything about the behavior of external contracts. ECF holds for most real-world contracts [56]. Formally, with

```
1 contract A {
2     bool lock;
3
4     function foo () public
5         tx_reverts lock
6     {  if(lock) { revert; } ...  }
7
8     function bar (address x) {
9        lock = true;
10       // external call
11       x.call(...);
12       lock = false;
13       // read/write of blockchain state
14       ...
15    }
16 }
```

Listing 4: Ensuring External Callback Freedom

ECF, we can soundly model a `call` instruction as returning immediately with no side-effects (no callbacks).

CELESTIAL has two ways of checking for ECF; one of these must hold for each external call. The first check is syntactic and borrowed from the VERX tool [56]. An external call is deemed ECF compliant if it is guaranteed to only be called at the end of a transaction. In other words, for any public method that may transitively invoke an external call, it must ensure that it does not read or write to the blockchain state after the call.

The first check is lightweight. It only requires inspecting the control flow of contract without any symbolic reasoning. External calls that do not fall in this category must satisfy CELESTIAL's second check that asserts that any callbacks made by an external call are guaranteed to revert. We explain this check using the CELESTIAL contract shown in Listing 4. There is an external call in method `bar` on line 11. To prevent reentrancy, the programmer uses a contract field called `lock` and follows the protocol that the `lock` will be assigned true when making an external call. Furthermore, each public method of the contract (such as `foo`) will revert if `lock` is set to *true*. It is easy to see that if the external contracts tries to call back a method of `A`, the transaction will abort.

CELESTIAL supports this reasoning as follows. The translation to F* adds a sequence of assertions preceding each external call (that does not satisfy CELESTIAL's first check). For each public method of the contract, it takes the `tx_reverts` condition on the method, say ϕ, and inserts `assert` ϕ before the external call. This will ensure that a call back to a public method is guaranteed to revert.

We make one remark about this check. A function body can be decorated with multiple `tx_reverts` specifications. For the ECF check, we pick a disjunction of all argument-free specifications, i.e., ones only over the fields of the contract. This ensures that the ECF assertion is well formed. Note that in the absence of any such `tx_reverts` specification on some public method, the ECF assertion will become `assert false`, effectively disallowing the external call unless the user adds more specifications.

V. EVALUATION

We evaluate the development experience with CELESTIAL by writing verified versions of 8 Solidity smart contracts, which include microbenchmarks as well as *real-world contracts*.[1] The selection criteria for our benchmark suite were to include a wide range of common application scenarios (such as crypto-currency tokens, wallets, marketplace, auctions and governance), and to include "high-valued" contracts (which currently hold millions of dollars of financial assets or have processed millions of transactions). Thus, ensuring functional correctness of these smart contracts is important.

For each contract, we added detailed functional specifications. We then used F* to verify the contracts for functional correctness. If the verification failed, we minimally modified the code in order to discharge the verification conditions. For contracts which required such modifications, we additionally compared the performance, with respect to gas consumption, against the original Solidity version.

We performed our experiments on a machine with an Intel Core i7-7600U dual-core CPU, with 16GB RAM, and running Windows 10. We measured gas consumption by writing JavaScript tests in the Truffle framework [22], which uses an in-memory blockchain to simulate transactions. We used the 0.6.8 version of the Solidity compiler for generating EVM bytecode for all the contracts. For Solidity contracts written against earlier versions of the compiler, we performed minor rewrites to make them compatible with v0.6.8. Table I summarizes the various case studies that we performed. The rest of this section describes each application in greater detail.

A. AssetTransfer

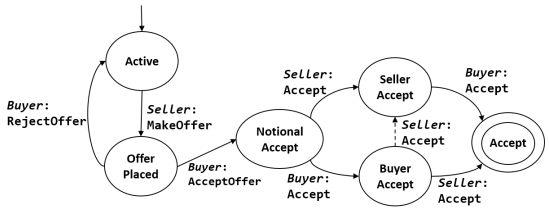

Fig. 4: The AssetTransfer state machine. The dashed arrow indicates a buggy state transition.

Application: AssetTransfer [18] is a microbenchmark that provides a smart contract based solution for transferring assets between a buyer and a seller. The contract encodes the asset transfer protocol as a finite state machine (FSM), as shown in Figure 4, with the different states denoting the varying stages of approval for the transfer. The use of FSMs has been recommended in various smart contract development scenarios [20], [51], for reasons of safety. The contract has notions of *roles*, such as Buyer and Seller, and state transitions are guarded by appropriate roles (for example, the contract can transition from `Active` to `OfferPlaced` only if the Seller invokes the `MakeOffer` function).

[1] Available as part of supplementary material included with this submission.

				CELESTIAL		
Benchmark	**Type**	**#C**	**#Sol**	**#Spec**	**#Impl**	**V-Time (sec)**
AssetTransfer*	Marketplace	1	130	70	187	4.26
OpenZeppelin ERC20	Token	4	171	97	200	8.82
BinanceCoin*	Token	2	133	25	136	29.98
WrappedEther*	Token	1	62	62	114	20.00
EtherDelta*	Wallet	1	281	57	351	63.97
Consensys MultiSig*	MultiSig Wallet	2	378	163	289	77.80
SimpleAuction*	Auction	1	66	61	101	22.45
Governance Contract	Consortium Management	1	417	121	149	86.86

TABLE I: CELESTIAL case studies. We report the number of contracts in the application (#C), LOC of the original Solidity implementation (#Sol), LOC of the CELESTIAL version, divided between specification (#Spec) and implementation (#Impl), and finally the F* verification time (averaged over 3 runs). Benchmarks marked with * had the additional CELESTIAL safe arithmetic library deployed, which is added towards #Impl

Specifications. Figure 4 forms the specification for this contract, that is, we check that each of the contract methods respect the transitions mentioned in the FSM diagram. For example, the following is the spec for `MakeOffer`:

```
function MakeOffer (uint _price)
  modifies [sellingPrice, state, log]
  tx_revert (old(state) != Active && sender != Seller)
  post (state == OfferPlaced && sellingPrice == price)
{ // implementation }
```

The specification ensures that the method makes the correct state transition (`Active` → `OfferPlaced`), and this transition can only be caused by the Seller. Interestingly, the verification of this specification failed, which led us to discover two bugs in the implementation of the state transition logic in the contract. These bugs could potentially leave the whole transfer in a frozen state. For instance, one of the bugs led to the erroneous state transition shown in Figure 4. It caused the contract to mistakenly transition to the `SellerAccept` state, even after both the Seller and Buyer had accepted the transfer, which makes the final state (`Accept`) to become unreachable, unless the Buyer accepts yet again. Fixing these bugs allowed verification to go through. Previous work [62] has noted similar bugs in a different version of the contract. The original contract also had overflow/underflow vulnerabilities, which we eliminated using runtime checks.

Performance. To measure the performance implications due to our code modifications, we ran both the CELESTIAL and Solidity versions of the contract through a typical workflow of deployment, followed by multiple transactions which represent a complete asset transfer. On an average, the CELESTIAL version consumed $1.12\times$ more gas compared to the original. We account for both the contract as well as any associated library, for instance for safe arithmetic, when measuring the deployment cost.

B. ERC20 Tokens

Application. In Ethereum, developers can create their own cryptocurrencies, called *tokens*. ERC20 is a *standard* [6], comprising informal specifications for a core set of token functionalities. Tokens implementing ERC20 provide a familiar interface to users, while allowing different such tokens to seamlessly interact with each other. Till date, over 250K ERC20 tokens have been deployed on Ethereum, handling financial assets worth *millions of dollars*. We formally verified the OpenZeppelin ERC20 contract [16], which is a popular reference implementation of some of the key ERC20 functions, such as transferring tokens from one account to another and approving third parties to spend tokens on a user's behalf. We also verified the ERC20-based BinanceCoin (BNB) [3] token, which has a current "fully diluted" market capitalization of over $480 million.

Specifications. We based some of our specifications on earlier efforts to formally verify the OpenZeppelin ERC20 token [13], [62]. Listing 5 shows an excerpt from the CELESTIAL ERC20 contract. The implementation maintains the balance (number of issued tokens) for each contract address using a `_balances` map. CELESTIAL allows us to easily express the important invariant (line 5) that the sum over the balances for each user equals the total number of tokens issued.

```
1 contract ERC20 {
2   mapping (address => uint) _balances;
3   uint _totalSupply;  // total issued tokens
4
5   invariant _balanceAndSellerCredits {
6     _totalSupply = sum_mapping(_balances)
7 }}
```

Listing 5: Excerpt from the CELESTIAL ERC20 contract

The remaining specifications capture the business logic of key ERC20 functions. As an example, consider the post-condition for the `_transfer` function in Listing 6, which is used for atomically debiting a source account, and crediting the amount in a destination account. The postcondition ensures that the correct debit and credit operations occur in the source and destination accounts, and all other accounts remain unchanged. The ERC20 token makes copious use of arithmetic operations. OpenZeppelin designed a SafeMath Library [17] to perform runtime checks for overflows and underflows, which the original ERC20 token leverages to ensure runtime safety for arithmetic operations. In contrast, we used the CELESTIAL safe arithmetic operations in public functions, and eliminated runtime checks altogether in private functions by means of appropriate preconditions, such as the one defined in line 4 of Listing 6.

```
1 function _transfer (address from, address to, uint amt)
2   private tx_reverts ..., modifies [...]
3   pre balances[from] >= amt &&
4       balances[to] + amt <= uint_max
5   post ite(from == to, balances == old(balances,
6          balances == old(balances)[
7            from => old(balances)[from] - amt,
8            to => old(balances)[to] + amt]))
9   { // implementation }
```

Listing 6: Specifications for token transfer

In addition to the core ERC20 functions, the BNB token allows a user to freeze or unfreeze their own funds. The token maintains an additional mapping from `address` to `uint`, called `freezeOf`, which tracks the amount of funds frozen for each user. As a result, the invariant we established for ERC20 (Listing 5) changes slightly to the following (note that the `balanceOf` field is equivalent to the `_balances` field of ERC20):

```
1 contract BNB {
2   mapping (address => uint) balanceOf;
3   mapping (address => uint) freezeOf;
4   uint totalSupply;
5   invariant _totalSupply {
6     totalSupply == sum_mapping(balanceOf) + sum_mapping(
          freezeOf)
7   }
8 }
```

Listing 7: Excerpt from the CELESTIAL BNB contract

C. WrappedEther

Application. Ether was the original currency for the Ethereum blockchain, and tokens based on the ERC20 tokens came later. As a result, ether and ERC20 tokens are incompatible between each other. WrapperEther (WETH) [27] solves this problem by providing functions that "wrap" native ether into an ERC20 compatible token. Users can trade ether for an equivalent number of WETH tokens, which can then be used to interact with other ERC20 compatible tokens. At any point, users can trade-in their WETH to get back ether. At the time of writing, the WETH contract that we consider [26] holds ether worth over $550 million.

Specifications. We wrote precise specifications for all the functions in WETH, including operations involving wrapping ether, transferring wrapped tokens between accounts, and unwrapping WETH to return an equivalent amount of ether. As an example, here is the specification for the `deposit` function, used to wrap ether:

```
1 function deposit () public credit modifies [balanceOf]
2   tx_reverts old(balanceOf)[sender] + value > uint_max
3   post (balanceOf == old(balanceOf)[
4     sender => old(balanceOf)[sender] + value])
5 {
6   // other code ...
7   balanceOf[sender] = safe_add(balanceOf[sender],value);
8   // other code ...
9 }
```

Listing 8: Specifications for wrapping ether

The `balanceOf` field is used to record the amount of WETH tokens associated with each address. The postcondition ensures that the correct amount of tokens is deposited to the corresponding account, and all other accounts are unchanged. Note that our use of `safe_add` in line 7 ensures that the function reverts in case of an overflow, which is a condition we capture in the `tx_reverts`. Thus, the postcondition in line 4 does not need to reason about overflows.

Performance. On an average, the CELESTIAL version consumed 1.13× more gas compared to the original Solidity version. The overhead can primarily be attributed to the added runtime checks on arithmetic operations (`safe_add`, for example).

D. EtherDelta

Application. Since there are a wide variety of tokens available, developers have designed several *wallet* contracts that perform token management on behalf of the users. Wallets typically allow users to deposit different *types* of tokens, allow third-parties to transact on their behalf, and even exchange one type of token for another. The wallet we investigated in particular was EtherDelta [7] which, at the time of this writing, holds financial assets worth over $4.5 million.

Specifications. Listing 9 shows an excerpt from the CELESTIAL version of EtherDelta. The map `tokens` tracks, for each token type, the amount of tokens held by each account, with `tokens[null]` tracking the amount of ether held by each account.

```
1 contract EtherDelta {
2   mapping (address => mapping (address => uint)) tokens;
3   bool _lock_;
4   invariant eth_balances {
5     balance >= sum_mapping(tokens[null])
6   }
7   function withdrawToken (address token, uint amt)
8     post (new(tokens) == tokens[token => tokens[token][
          sender => tokens[token][sender] - amt]])
9     tx_reverts token == null || tokens[token][sender] <
          amt || _lock_
10    modifies[tokens, totalBalance, balance, log, _lock_]
11  {
12    // other code ...
13    token.call("transfer(address,uint)", sender, amt);
14    // other code ...
15  }
16 }
```

Listing 9: Excerpt from EtherDelta CELESTIAL version

The invariant establishes that the amount of ether held by the wallet exceeds the total ether held by the users. This is crucial, otherwise users may lose ether after depositing it in the wallet. The `withdrawToken` function allows users to withdraw any token type other than ether. The postcondition establishes that the only effect of this function is to correctly update the `sender` entry in the `tokens` map. Note that the function uses the `call` function to invoke an external function, which is why we followed the protocol outlined in Section IV-C to prove the absence of reentrancy using the boolean variable `_lock_`.

E. Consensys MultiSig

Application. The Consensys MultiSig wallet [9] allows users to perform ether transactions (such as deposits, withdrawals, and transfers), but adds an additional layer of security by enforcing that each transaction be confirmed by a set of approvers. The version of the contract we investigate, called `MultiSigWalletWithDailyLimit` [10], presently holds ether worth over $19 million and additionally

allows users to withdraw a limited amount of ether without the approvals.

Specifications. The key specifications ensure that each transaction gets signed by the requisite number of approvers before it is executed. For example, consider the postcondtion for `confirmTransaction`, which is invoked by an approver to add his/her signature to a transaction with identifier `_txId`.

```
1 // bump up number of signs for _txId
2 signCounts == old(signCounts)[_txId => old(signCounts)[
      _txId] + 1] &&
3 // record the sign from _sender
4 signs == old(signs)[_txId => old(signs)[_txId][_sender =>
      true]] &&
5 ite(
6   // is number of signs for _txId sufficient?
7   signCounts[_txId] == _required,
8   // execute the transaction
9   log == (old(tx)[_txId].dest, eTransfer, old(tx[_txId].
        val)::old(log) &&
10  tx == old(tx)[_txId => Tx(old(tx)[_txId].dest, old(tx)[
        _txId].val, true)],
11  tx == old(tx) ) // do nothing
```

Listing 10: Post-condition of `confirmTransaction`

The postcondition specifies that, upon confirmation from an approver, the number of signatures (`signCounts`) for `_txId` should be incremented by 1 (line 2), and the signature from the approver be recorded in the `signs` map (line 4), with all other entries remaining unchanged. The contract stores a mapping `tx` of transaction objects indexed by the identifiers. If the transaction receives the necessary number of signatures (line 7), then it must be executed by sending ether to the destination of the transaction object (line 9). In CELESTIAL, sending of ether results in an `eTransfer` event getting added to the log, and the user can then reason with it (as we do here) to ensure that the appropriate amount of ether is sent. It must also record in `tx` that `_txId` has been completed successfully (line 10). If `_txId` does not receive the required number of signatures, then the `tx` data structure must remain unchanged (line 11). CELESTIAL was able to verify this, and all the other specifications, in around 4 minutes.

Performance. Due to limitations in our current CELESTIAL implementation, the Solidity and CELESTIAL implementations differ in important ways. Since we do not currently support arrays in arguments, deploying the contract with n approvers requires n transactions in CELESTIAL, while this can be achieved in a single transaction in the original version. Thus, the cost of deploying the CELESTIAL version with 3 owners is $1.2\times$ higher, compared to the original version. Also, since we currently do not support writing loop invariants, we used additional data structures to support the same functionality. This resulted in operations such as adding approvers, replacing them or changing the number of signatures to consume less gas for CELESTIAL. However, the use of safe arithmetic operations during the approvals results in the CELESTIAL contract requiring $1.4\times$ more gas, compared to the original version. The purpose of this experiment was to demonstrate that the current CELESTIAL implementation is expressive enough to capture the semantics of MultiSig, and these gas numbers only serve to highlight the performance implications of the current CELESTIAL restrictions.

F. SimpleAuction

Application. SimpleAuction [19] is a microbenchmark in the Solidity documentation, and represents a hypothetical blockchain-based auction. During deployment, the owner must provide the duration for which the auction remains open for bidding. While the auction is running, users can submit open bids by transferring ether equivalent to their bid valuations. When a subsequent bid exceeds the current highest bid h from a user s, the contract allocates h ether to the account of s using a `pastBids` map. At any point of time, s can withdraw the funds allocated to it. When the auction ends, the owner of the contract can retrieve ether equivalent to the highest bid, and subsequent bids are rejected.

Specifications. An important invariant we establish for the contract is that while the auction is active, the balance of the contract never falls below sum of the highest bid and the funds allocated to all previously highest bidders:

```
active => (balance >= sum_mapping(pastBids) + maxBid)
```

We were also able to write and prove the postcondition that when the auction ends, the owner of the auction is able to retrieve ether equivalent to the highest bid:

```
!active && (log == (owner, eTransfer, maxBid)::old(log))
```

G. Governance Contract

Application. We also investigated a contract from a large cloud service provider that manages a consortium of mutually-trusted members interacting on a *private* Ethereum blockchain. The contract comprises a set of rules governing operations such as inviting fresh members to join the consortium and adding or removing existing members. The contract is complex, since it maintains many correlated data structures, loops and access control policies, with each logical operation involving intricate changes to multiple data structures. Due to the proprietary nature of the contract, we abstain from showing code or specifications for it explicitly. We did not include several functions in the original contract, whose operations were orthogonal to the governance logic.

Specifications. We briefly describe some of the important properties that we proved.

1) Among members in the consortium, some are designated as being "administrators". An important invariant is that the number of administrators cannot be zero. A violation of this invariant will result in the consortium transitioning to a frozen state, with no further processing of transactions.
2) In the contract, logical units of information are maintained in aggregate by several data structures. For example, the contract maintains an array of existing members. However, members can either be referenced by a string identifier, or an address. Thus, the contract maintains a couple of additional mappings that maintain, respectively, associations between string identifiers and addresses, to the correct indices in the array. We specify several invariants to ensure that these data structures are always consistent. For example, we specify that there are no duplicates in the

array, no two string identifiers map to the same array index, and the value of each string identifier must not exceed the length of the array of members.

3) We precisely captured the postconditions for operations such as member additions, where we ensure that the operation only updates the necessary keys and indices, while leaving the remaining entries untouched.

We note that some of these properties are similar to those proved by Lahiri et al [47] for a variation of an open-source governance contract [23]. In that work, the authors also specify and prove that the number of administrators never becomes 0. The open-source contract also maintains the following map and array:

```
address[] addressList;
mapping (address => bool) inSet;
```

The array `addressList` stores a list of addresses who are administrators, while the `inSet` mapping is used to check whether an address is an administration in $O(1)$ time. The array is maintained explicitly since Solidity disallows iterating over the keys of a mapping. In this setting, Lahiri et al specified that no duplicates should exist in `addressList`, and that `inSet`$[a]$ is true iff a exists in some index of `addressList`. Note that these specifications are similar in spirit to the invariants we specified in point 2 above. These indicate that the properties we proved in CELESTIAL may be of interest for governance contracts in general.

VI. RELATED WORK

The literature on ensuring correct smart contracts can be classified into the following broad categories.

Surveys and Best Practices. There is a wealth of available material that highlights known vulnerabilities during smart contract development [34], [36], [61], with some recent work even surveying the extent to which these vulnerabilities have been exploited [55]. These efforts have resulted in literature suggesting best coding practices for smart contract development in Solidity [11], [21]. Our design of CELESTIAL is inspired by these practices, for instance, ruling out low-level instructions as well as uncontrolled reentrancy, however, they are not just for avoiding programming pitfalls; we leverage these restrictions to aid semantic verification.

Testing. Frameworks like Truffle [22] allow users to write unit and integration tests for smart contracts in JavaScript. The transactions are typically executed in an in-memory mock of the EVM, such as Ganache [14]. In addition to testing functional behaviors and finding bugs, such tests reveal useful diagnostic information such as gas consumption.

Contract Analysis. A large number of tools have been developed that statically analyze smart contracts (Solidity source code or EVM bytecode) to reveal various vulnerabilities. Examples include MadMax [38] (targeting vulnerabilities due to gas exceptions) and Slither [37] (for identifying security vulnerabilities). Oyente [50] leverages symbolic execution to rule out several classes of vulnerabilities. ContractFuzzer [44] offers a fuzzing based solution for identifying security bugs, where interesting inputs are generated by analyzing the contract's interfaces and using test oracles.

Solythesis [49] is a source-to-source Solidity compiler that instruments the Solidity code with runtime checks to enforce invariants, but specifications particular to each function can't be specified in this framework and it has a significantly high gas overhead because of the runtime checks. VeriSmart [58] offers a highly precise verifier for ensuring arithmetic safety of Ethereum smart contracts, which discovers transaction invariants, but is unable to capture quantified transaction invariants. Tools like teEther [46] leverage symbolic execution to find vulnerable executions and automatically generate exploits.

Each of these tools target a known set of vulnerabilities and offer specialized solutions for them. In contrast, CELESTIAL verifies custom specifications of contracts, relying on verification to rule out all vulnerabilities against that specification.

Formal Verification. VeriSol [47], [62] checks conformance between a state-machine-based workflow and the smart contract implementation, for contracts of Azure Blockchain Workbench [2]. VeriSol does not check for reentrancy; it simply assumes its absence, as opposed to CELESTIAL that enforces it as part of the contract verification. Further, VeriSol does not model arithmetic over/underflow, or check for unsafe type casts, which were an important aspect of our case studies.

VerX [24], [56] is another recent tool looking at formal verification of Solidity smart contracts. VerX uses a syntactic check to ensure ECF (which we use in CELESTIAL as well), however it cannot verify that the program in Listing 4 satisfies ECF. VerX aims for automation of verification by inferring predicates in an abstraction-refinement loop. Such techniques tend to be limited in their ability to reason with quantifiers; VerX uses special built-in predicates like `sum` for quantified reasoning over maps. CELESTIAL, on the other hand, allows for the full power of first-order reasoning with quantifiers. VerX implements its own custom symbolic execution, whereas CELESTIAL uses a simple syntax translation to F* and delegates all analysis to the mature F* verifier. Unfortunately, the VerX tool is not openly available for further comparisons.

Hirai [42] provided an encoding of the EVM semantics in Lem [52], allowing its use with several interactive theorem provers to establish safety properties for smart contracts. Hildenbrandt et al [41] provided a formalization of the EVM semantics in the $\mathbb{K}$ framework [57], which allowed them to derive deductive verifiers for auditing the safety of smart contracts. Working at the EVM level is more precise and removes the Solidity compiler from the TCB, however, it is also more time consuming and hard to scale to the larger, complex contracts that we have evaluated in Section V. Bhargavan et al. [35] provide an approach to translate a subset of Solidity to F* for verification, as well as a method to decompile EVM bytecode to F* to check low-level properties such as establishing worst-case gas bounds for a transaction. Their work is presented as a proof-of-concept only, with limited evaluation and restricted to a small subset of the language.

VII. Conclusion

We have presented CELESTIAL, a framework for developing formally verified smart contracts. CELESTIAL provides fully automated verification, using F*, of Solidity contracts annotated with functional correctness specifications. With the help of several real-world case studies, we conclude that formal verification can be made accessible to smart contract developers for programming high-assurance contracts. Our next steps include enriching our F* model of blockchain with more features and validating it using the Solidity testsuite as well as exploring proofs of cross-transaction properties.

References

[1] Antlr. https://www.antlr.org/.

[2] Azure blockchain workbench. https://azure.microsoft.com/en-us/solutions/blockchain/.

[3] Binance coin. https://www.binance.com/en.

[4] Certora. https://www.certora.com/.

[5] Consensys secure development recommendations. https://consensys.github.io/smart-contract-best-practices/recommendations/.

[6] Eip 20: Erc-20 token standard. https://eips.ethereum.org/EIPS/eip-20.

[7] Etherdelta. https://etherdelta.com/.

[8] Ethereum in bigquery: a public dataset for smart contract analytics. https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics.

[9] Ethereum multisignature wallet. https://github.com/Gnosis/MultiSigWallet.

[10] Ethereum multisignature wallet on etherscan. https://etherscan.io/address/0x851b7F3Ab81bd8dF354F0D7640EFcD7288553419.

[11] Ethereum smart contract security best practices. https://consensys.github.io/smart-contract-best-practices/.

[12] Executing F* code. https://github.com/FStarLang/FStar/wiki/Executing-F*-code.

[13] Formal verification of erc20 implementations with verisol. https://forum.openzeppelin.com/t/formal-verification-of-erc20-implementations-with-verisol/1824.

[14] Ganache. https://github.com/trufflesuite/ganache.

[15] Mythril. https://github.com/ConsenSys/mythril.

[16] Openzeppelin erc20. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol.

[17] Openzeppelin safemath. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol.

[18] Remix ethereum ide. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/asset-transfer.

[19] Simple open auction, solidity documentation. https://solidity.readthedocs.io/en/v0.6.8/solidity-by-example.html#simple-open-auction.

[20] Solidity docs: State machines. https://solidity.readthedocs.io/en/v0.6.8/common-patterns.html#state-machine.

[21] Solidity security considerations. https://solidity.readthedocs.io/en/v0.6.8/security-considerations.html.

[22] Truffle suite. https://www.trufflesuite.com/.

[23] Validator set contracts. https://github.com/Azure-Samples/blockchain/tree/master/ledger/template/ethereum-on-azure/permissioning-contracts/validation-set.

[24] Verx. https://verx.ch/.

[25] Visual studio code. https://code.visualstudio.com/.

[26] Weth9 on etherscan. https://etherscan.io/address/0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2.

[27] Wrapped ether. https://weth.io/.

[28] Understanding the dao attack. https://www.coindesk.com/understanding-dao-hack-journalists, 2016.

[29] The parity wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/, 2017.

[30] Etherscan. https://etherscan.io/, 2020.

[31] Etherscan: Contract accounts. https://etherscan.io/accounts/c, 2020.

[32] Solidity v0.7.2. https://solidity.readthedocs.io/en/v0.7.2/, 2020.

[33] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 515–529. ACM, 2017.

[34] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

[35] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. Formal verification of smart contracts: Short paper. In Toby C. Murray and Deian Stefan, editors, *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, pages 91–96. ACM, 2016.

[36] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks and defenses. *CoRR*, abs/1908.04507, 2019.

[37] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 8–15. IEEE / ACM, 2019.

[38] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):116:1–116:27, 2018.

[39] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[40] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 531–548. ACM, 2019.

[41] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 204–217. IEEE Computer Society, 2018.

[42] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, volume 10323 of *Lecture Notes in Computer Science*, pages 520–535. Springer, 2017.

[43] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[44] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 259–269. ACM, 2018.

[45] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[46] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1317–1333. USENIX Association, 2018.

[47] Shuvendu K. Lahiri, Shuo Chen, Yuepeng Wang, and Isil Dillig. Formal specification and verification of smart contracts for azure blockchain. *CoRR*, abs/1812.08829, 2018.

[48] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[49] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 438–453. ACM, 2020.

[50] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016.

[51] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer Science*, pages 523–540. Springer, 2018.

[52] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 175–188. ACM, 2014.

[53] Daejun Park, Yi Zhang, and Grigore Rosu. End-to-end formal verification of ethereum 2.0 deposit smart contract. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2020.

[54] Terence John Parr and Russell W. Quong. ANTLR: A predicated- *LL(k)* parser generator. *Softw. Pract. Exp.*, 25(7):789–810, 1995.

[55] Daniel Pérez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? *CoRR*, abs/1902.06710, 2019.

[56] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*, pages 18–20, 2020.

[57] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010.

[58] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VERISMART: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1678–1694. IEEE, 2020.

[59] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

[60] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 67–82. ACM, 2018.

[61] Antonio Lopez Vivar, Alberto Turégano Castedo, Ana Lucila Sandoval Orozco, and Luis Javier García-Villalba. An analysis of smart contracts security threats alongside existing solutions. *Entropy*, 22(2):203, 2020.

[62] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, volume 12031 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2019.

	balance (1306 contracts)	**txn volume** (1767 contracts)	**txn count** (8218 contracts)
`delegatecall()`	4.4%	2.8%	22.6%
`gasleft()` / `msg.gas`	2.0%	3.2%	0.9%
`assembly`	34.8%	25.1%	17.3%

TABLE II: Percentage of top-ranking contracts that use a particular Solidity construct.

APPENDIX A
MINING SOLIDITY CONTRACTS

We performed an empirical study on the number of contracts using different constructs/features. We queried a publicly available up-to-date dataset [8] to rank Ethereum smart contracts by the amount of Ether held (`balance`), by the number of transactions processed (`txn count`) and by the total value sent to the contract in transactions, i.e., transaction volume[2] (`txn vol`).

Figure 5 shows that, at the time of writing, the top $10,000$ contract addresses ranked by balance account for 99.84% of the total Ether held by all Ethereum smart contracts. Similarly, top $10,000$ contracts ranked by the number of transactions account for 91.42% of the total transactions (Figure 6), and the top $10,000$ contract addresses ranked by the transactions volume cover 96.30% of the total volume of the transactions sent to the contract addresses (Figure 7).

On trying to retrieve the contract source codes from Ether-Scan [30], we noticed that the source code for all the $10,000$ top-ranking contract addresses were not available, and hence considered the top $16,000$ contract addresses as representative addresses for each of these categories. We were able to retrieve the sources for 9211, 10570 and 8247 addresses out of the top $16,000$ addresses in the balance, transaction-count and transaction-volume categories respectively.

Further, in each of these categories, we noticed a high degree of repetition in the source codes. We pruned out duplicate contracts that had *exactly* the same source code. On filtering out these duplicates, we were left with 1306, 8218 and 1767 contracts in the balance, transaction-count and transaction-volume categories respectively. We note that further pruning of duplicates is possible because some contracts only differ in identifier names, harcoded constants, etc., but we did not perform this level of pruning.

We report the frequencies for some Solidity language constructs that we do not model (Section II-C), disallow or are an implementation limitation (Section IV) in Table II.

While we disallow `delegatecall` and do not model `gasleft()` (formerly `msg.gas`), only a very small number of contracts use these constructs and therefore do not hurt the expressivity of CELESTIAL. The number of contracts that use inline assembly are relatively higher. Further inspection revealed that this is due to repeated use of popular Solidity libraries such as `ECVerify`, `Oraclize`, `Buffer` and `String`

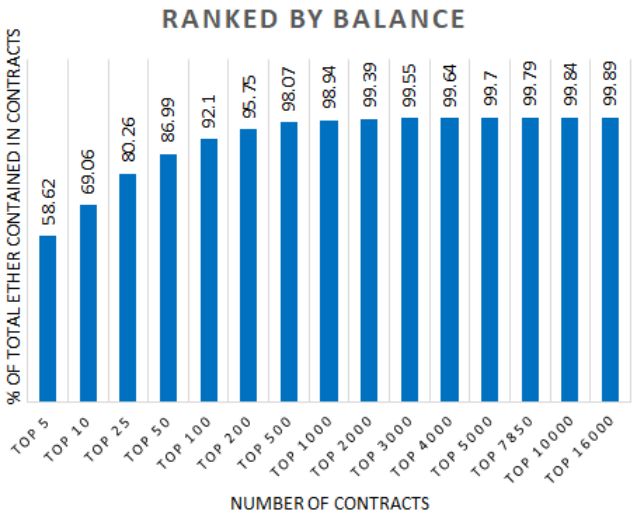

Fig. 5: Contract accounts ranked by balance.

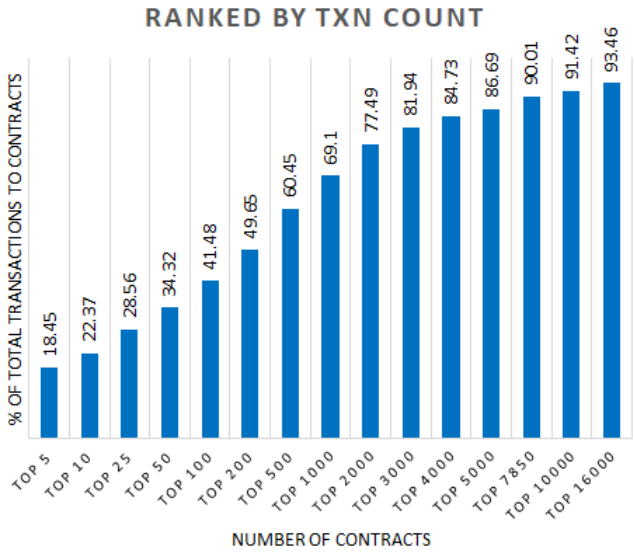

Fig. 6: Contract accounts ranked by number of transactions sent to them.

[2]By transaction volume of an address, we mean the total value sent to an address in all transactions to it.

that use inline assembly, as opposed to programmers inlining assembly themselves. A possible future work can be to model the specifications of these libraries in F* and allow them to be invoked directly from CELESTIAL code. We can rely on open-source auditing instead of formal verification to verify those specifications, given that most cases can be covered with just a few libraries. Another alternative would be to use EVM-based verification tools [57] to discharge these specifications. Programmers at large can still only concentrate on Solidity-level verification with CELESTIAL.

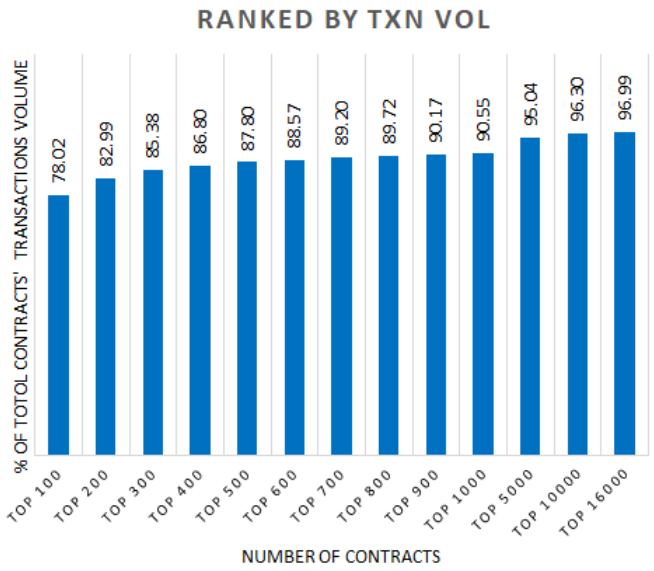

Fig. 7: Contract accounts ranked by volume of transactions sent to them.