Towards Verifying Ethereum Smart Contracts at Intermediate Language Level

Ximeng Li[1,3(✉)], Zhiping Shi[1(✉)], Qianying Zhang[2], Guohui Wang[2], Yong Guan[3,4], and Ning Han[1]

[1] Beijing Key Laboratory of Electronic System Reliability and Prognostics, Capital Normal University, Beijing, China
{lixm,shizp,2181002006}@cnu.edu.cn

[2] Beijing Engineering Research Center of High Reliable Embedded System, Capital Normal University, Beijing, China
{qyzhang,ghwang}@cnu.edu.cn

[3] Beijing Advanced Innovation Center for Imaging Theory and Technology, Capital Normal University, Beijing, China
guanyong@cnu.edu.cn

[4] International Science and Technology Cooperation Base of Electronic System Reliability and Mathematical Interdisciplinary, Capital Normal University, Beijing, China

Abstract. Smart contracts have exhibited great potential in a spectrum of applications, ranging from digital currency to online gaming. Yet smart contracts are known to be prone to errors and vulnerable to attacks. The validation of smart contracts before their deployment is an indispensable step for their correctness and security, and the highest level of guarantee can be provided using formal verification. The level of difficulty, reliability, etc., of the formal verification of a smart contract is deeply affected by the programming language in which the contract is implemented. In this paper, we discuss the benefits of verifying smart contracts at the level of intermediate languages, in comparison with machine-level languages and user-level languages. We augment the existing formalization of Yul – the intermediate language of Ethereum, realize an ERC20 token contract in this language, and verify the guarantees of all the functions provided by this contract. All this development has been performed in the proof assistant Isabelle/HOL. It demonstrates the feasibility and some of the most important advantages of mechanized verification for smart contracts at the intermediate-language level, such as a balance between the intuitiveness of the verification target and the ability to validate lower-level mechanisms like the function dispatcher.

1 Introduction

The blockchain technology [29] has raised a significant amount of attention both from the technological community specifically and from the society at large. A blockchain is a digital ledger consisting of blocks of records, which are linked together through hash values. Copies of the same ledger are maintained at a great

Y. Ait-Ameur and S. Qin (Eds.): ICFEM 2019, LNCS 11852, pp. 121–137, 2019.
https://doi.org/10.1007/978-3-030-32409-4_8

number of network nodes. The ledger is append-only, with a consensus mechanism guaranteeing a unified view of newly appended blocks. This design enables distributed consensus over data, while providing guarantees such as tamper-resistance, denial-resistance, and backward-traceability.

The blockchain hosts not only plain data but also executable programs. The programs executed over the blockchain are often called smart contracts (as was conceptualized in [25]). Typically, they prescribe the actions performed (e.g., money transfer between accounts) under a number of predefined conditions. Owing to the guarantees provided by the underlying blockchain, distributed consensus is obtained over the outcome of the execution of smart contracts. Smart contracts have found application in numerous areas, such as financing, supply-chain management, smart manufacturing, health information management, etc.

While adding much to the versatility and power of the blockchain, smart contracts can be prone to errors, and vulnerable to security attacks – just like ordinary computer programs. Since they often deal with monetary concerns, the misbehaviors of smart contracts could directly cause harm to the economic rights of the participants. The fact that smart contracts are often written in an unconventional language (e.g., Solidity), and run on unconventional infrastructure, invites further possibilities of attack. One of the most notorious attacks on smart contracts is the DAO attack, which caused ∼$60M to be lost (e.g., [11]) by the legitimate participants of the DAO contract [6].

To minimize the chances of errors and attacks, smart contracts must be thoroughly validated before being deployed. Formal verification provides the highest level of correctness and security guarantees in the validation of IT systems, smart contracts included. When formally verifying a smart contract, the abstraction level of the contract is a critical factor to be considered. This abstraction level is determined by the language in which the contract is to be realized. For Ethereum smart contracts, verification has been attempted both for high-level languages such as Solidity (e.g., [30]), and low-level languages such as EVM (Ethereum Virtual Machine) bytecode (e.g., [19]). In general, the use of a high-level language adds to the intuitiveness and manageability of the verification, while the use of a low-level language minimizes the trust base of the verification. Neither approach tends to enjoy the most important benefits of both.

In this paper, we explore the middle ground – the verification of smart contracts in an intermediate language (IL). This helps strike a balance between the intuitiveness of the verification, and the ability to reduce the needed trust base, in ensuring the safety and security of smart contracts. Based on formal semantics, we conduct a substantial case study for IL-level verification of smart contracts. The verification is performed in a proof assistant (Isabelle/HOL), adding to the confidence level on the results obtained. Our main technical contributions are:

- revised formalization of the Yul language (the IL of Ethereum), including the formalization of function lookup due to observed mismatch between the specification of Yul in English and its existing formalization (Sect. 3),
- realization of an ERC20 token contract [2] in Isabelle/HOL, in the formalized Yul language (Sect. 4), and

```
contract Token {
  mapping(address=>uint256) public balances;
  ...
  function balanceOf(address _owner) public view returns(uint256) {
      return balances[_owner];
  }
  ...
}
```

Fig. 1. The token contract with Balance-retrieval Functionality in solidity

– mechanized proofs of the guarantees provided by each function in the contract – in the form of pre/post-conditions for the body of each function, and for the external call invoking each function (Sect. 5).

Our development totals ∼10k lines of code in Isabelle/HOL, of which ∼500 lines correspond to the realization of the token contract, ∼4k lines correspond to the specification and proof for the function definitions in this contract, and the rest correspond to the specification and proof for the calls to these functions.

2 Verifying Smart Contracts at the IL Level

In this section, we discuss the comparative benefits of formally verifying smart contracts at the intermediate-language level. We use Solidity [4], EVM bytecode [28], and Yul [8] as representative examples for smart contract languages at the high level, the low level, and the intermediate level, respectively.

Verifying Contracts in Solidity. Solidity is the official programming language of Ethereum. It offers contracts, balances, transfers, etc., as programming abstractions. A contract allowing for the retrieval of the balances of all the participants in some token could be implemented as in Fig. 1. In this figure, the contract is represented by the *contract* construct of Solidity, the balances are maintained in a *mapping* (from the address of each owner of the token to the current balance of the owner), and the operation retrieving the balance of a specific owner is implemented as a *function*.

As a structured, user-level language for smart contracts, Solidity allows for intuitive representation of the business logic of each contract. This facilitates the development of a specification in a formal verification (e.g., preconditions, post-conditions, loop invariants, etc.). On the other hand, as a high-level language, the features of Solidity are relatively complicated (with static and dynamic arrays, mappings, inheritable contracts, access modifiers, imports, etc.). Furthermore, the language is partly in its maturing process, and, hence, the evolvement of its features is relatively fast. These two facts pose great challenges to the development and stabilization of a formal semantics for Solidity, and the implementation of a verification system on top of the semantics.

```
function balanceOf(owner) -> bal {
  bal := sload(accountToStorageOffset(owner))
}
```

Fig. 2. The Balance-retrieving function of token contract in Yul

Verifying Contracts in EVM Bytecode. EVM bytecode is the language of the execution engine of Ethereum – the Ethereum Virtual Machine (EVM). Implementing the contract of Fig. 1 in EVM bytecode requires the implementation of e.g., a function dispatcher that directs each call to the contract to a specific function using the `JUMPI` instruction, the computation of the storage location of a specific owner's balance using arithmetic and stack-operating instructions, the retrieval of the balance of the specified owner using the `SLOAD` instruction, etc.

As a machine-level language, EVM bytecode does not permit a verification engineer to clearly see the business logic of the smart contract to be verified. This could lead to difficulties in developing the specification for the verification, and in coming up with the necessary auxiliary information to guide the verification. On the other hand, EVM bytecode is much less involved and more stable in terms of language features, than a user-level language such as Solidity. This facilitates the development and stabilization of a formal semantics. Furthermore, verifying the bytecode excludes the possibilities for errors introduced by the compiler, adding to the level of confidence on the verification result.

Verifying Contracts in Yul. Yul is the intermediate language of Ethereum, it enables structured programming with constructs for contracts, functions, conditional branches, and loops. At the same time, it supports the direct programming of low-level mechanisms such as the dispatcher of calls to specific contract functions, and the direct obtainment of the return data from calls.

An implementation of the balance-retrieval function in Fig. 1 in the Yul language is shown in Fig. 2. The computation of the storage address for the owner's balance is performed using the auxiliary function *accountToStorageOffset*, the implementation of which is elided from the figure.

The aforementioned characteristics of Yul indicate that it would not be difficult to comprehend the business logic of a smart contract while making a formal specification for the code of the contract (as is the case for a high-level language such as Solidity). At the same time, Yul supports functionalities that are occasionally necessary in the implementation of smart contracts, but are not directly offered by a high-level language (e.g., retrieval of resulting data of contract calls). Furthermore, the function dispatcher and other low-level mechanisms explicitly contained in a contract implemented in Yul can be directly examined in a formal verification, excluding the chances for the introduction of errors into these mechanisms by a compiler. Finally, the feature set of Yul is succinct and stable in comparison to that of Solidity, which reduces the difficulty level of formalization.

3 The Formalization of Yul

The formalization of the Yul language in Isabelle/HOL serves as the (only) basis of our verification of the ERC20 token contract. A preliminary formalization of Yul (previously called Julia) has been performed by Hirai [3] in the Lem tool [22]. From Lem, we generate definitions of the syntax and (big-step) semantics of Yul in Isabelle/HOL, and we revise the formalization for use as a basis of our work. In this section, we first briefly introduce the basics of Isabelle/HOL, and then describe the formalization of Yul by Hirai and our revision of it.

3.1 The Basics of Isabelle/HOL

Isabelle/HOL is an environment that provides the ability to reason formally in Higher Order Logic inside the Isabelle framework [27]. System verification using Isabelle/HOL reduces the verification problem to the construction of a formal proof. The modeling of the system is often performed by functional programming, and the proofs are often constructed by applying predefined tactics, or using the declarative-style language Isar.

For simple definitions, the keyword *definition* is used. In case the definition involves pattern matching or recursion, the keyword *function* or *fun* is needed. In lemmas and theorems, all the hypotheses can be listed between a pair of semantic brackets ⟦...⟧ and separated with semicolons.

The notation [] represents an empty list, and $e\#l$ represents the list that results from prepending the element e to the list l. The term *Map.empty* represents an empty map (the map that takes each key to *None*), $mp[k \mapsto v]$

```
datatype expression =
    FunctionCall id0 "expression list"
  | Identifier id0
  | Literal "literal_kind" "type_name"
```

Fig. 3. The existing formalization of Yul expressions

```
datatype statement =
    Block "statement list"
  | FunctionDefinition id0 "(id0 × type_name) list"
      "(id0 × type_name) list" statement
  | VariableDeclaration "(id0 × type_name) list" expression
  | Assignment "id0 list" expression
  | If expression statement
  | ForLoop expression statement statement
  | Expression expression
  ... ...
```

Fig. 4. The existing formalization of Yul statements

represents the map that results from updating the map mp by mapping k to $Some(v)$, and $mp_1 +\!\!+ mp_2$ represents the map that results from updating the map mp_1 according to the map mp_2, i.e., for each key k, if mp_2 takes k to $Some(v)$, then $mp_1 +\!\!+ mp_2$ takes k to $Some(v)$; otherwise $mp_1 +\!\!+ mp_2$ takes k to $(mp_1\ k)$. For a record rcd with field fd, $(fd\ rcd)$ represents the value of fd in rcd, and $rcd(\!|fd := v|\!)$ represents the record rcd with the field fd updated to the value v.

3.2 The Original, and Revised, Formalization of Yul

The two main syntactical categories of Yul are expressions and statements. Their formalizations are shown in Figs. 3 and 4, respectively. In both figures, *id0* is the type for the identifiers of variables and functions. There are two types of function calls – the call to a function in the current contract (internal calls), and the call to a different contract (external calls). Both are supported by the type *FunctionCall id0* "*expression list*" in Fig. 3: if a function defined in the current scope is associated with the function identifier, then an internal call is performed, while if the builtin function *Call* is associated with the function identifier, an external call is performed. Since a function may have a list of return values (in addition to a list of parameters), the type constructor *FunctionDefinition* in Fig. 4 has two lists as arguments. Although Fig. 4 is non-exhaustive in the statements of Yul, the full definition of *statement* is not much more involved than what is shown. It can be seen that the language has a succinct syntax.

```
fun func_map :: "statement ⇒ ((id0, value0) Map.map)" where
    "func_map (Block []) = Map.empty"
  | "func_map (Block (stmt # stmts)) =
       (func_map stmt ++ func_map (Block stmts))"
  | "func_map (FunctionDefinition f params rets stmt) =
       (Map.empty)(f ↦ FunctionV f params rets stmt)"
  | "func_map _ = (Map.empty)"
```

Fig. 5. The definition of *func_map*

A *global state* g of a contract contains the address of the currently executing contract *address* g, the currently executing contract *current* g, the memory of the execution engine *memory* g, the active number of bytes in the memory *memory_size* g, the value transfered with the call invoking the execution of the current contract *tmoney* g, the input data of this call *calldata* g, the current log content *logs* g, the function from account addresses (modeled by integers) to the corresponding accounts *accounts* g, and other components relevant to the execution of contracts. A *local state* l is a map from identifiers (of type *id0*) to values of type *value0*. For each account at address *addr* (a 160-bit address), i.e., $acc = accounts\ g\ addr$, *storage acc* represents the storage of the account,

balance acc represents the balance of the account, and *code acc* represents the code of the account. A contract is an account with non-empty code.

The existing formalization of Yul also contains the big-step semantics for expressions and statements, defined using two *evaluation functions*. The function *eval_expression* takes a global state, a local state, an expression, and a natural number as arguments, and returns the final result of evaluating the expression. Here, the natural number is a counter introduced only to facilitate a termination proof for the well-definedness of *eval_expression* in Isabelle/HOL. The function *eval_statement* takes a global state, a local state, a statement and a natural number (serving also as a counter for proving termination), and returns the result of executing the statement. The two functions are mutually recursive since a statement may have in it a function call (an expression), and an expression may be the invocation of a function whose body is a statement.

In the original formalization [3], the functions that can be internally called in the current scope are maintained by associating each such function to its identifier in the local state, after processing the function definition. However, this only allows for calling functions whose definitions are syntactically located before the calls. Nonetheless, as mandated in the informal specification of Yul [8]

> "Functions can be referenced already before their declaration (if they are visible)."

To rectify this mismatch between the official documentation of Yul, and its existing formalization, we define the function *func_map* to build a map *fctx* for all the functions defined in a statement (see Fig. 5). We augment the parameter list of the functions *eval_statement* and *eval_expression* to contain this map, thereby recording which functions are defined in the current scope, both before and after the point where a function is called. With this revision, the terms

$$\begin{array}{l} \mathit{eval\_expression}\ \ g\ l\ \mathit{fctx}\ \mathit{expr}\ n \\ \mathit{eval\_statement}\ \ g\ l\ \mathit{fctx}\ \mathit{stmt}\ n \end{array}$$

represent the evaluation of expressions and execution of statements, respectively, with knowledge of the available functions in the current scope. We inductively prove that the result of evaluation a statement or an expression does not depend on the value of the counter n, as long as n is sufficiently large for the evaluation function to be fully unrolled.

Our revision of the formalization of Yul also contains the addition of a number of definitions for the evaluation of builtin functions, such as subtraction, multiplication, division, the function retrieving the value transfered with the current call, the function returning the address of the caller account, etc. Most of these additions to the original formalization are used in our realization of the token contract in the formal Yul language.

4 Realizing the Token Contract in Yul

A token contract keeps track of the total supply of a token, its current distribution among its owners, and its flow between its owners. The ERC20 standard

for token contracts mandates a number of interfaces to be provided, such as querying the total supply of the token and the current balances of the owners, and transferring a specified amount of tokens to a specified user [2].

We realize a version of ERC20 token contract in the formalized Yul language in Isabelle/HOL. However, in the presentation of this section, we refrain from using the Isabelle syntax due to its verbosity.

4.1 The Storage Layout of the Contract

The storage of an Ethereum smart contract is arranged in slots that are addressed by 256-bit integers. We model the storage layout of the token contract as follows, where *keccak* is the keccak-256 hash function, and $uint256(n)$ is the bit string of length 256 for the unsigned integer n.

- The owner of the contract is stored at slot 0.
- The total supply of the token is stored at slot 1.
- The balance of the account at address *addr* is stored at slot

$$keccak(uint256(addr).uint256(2))$$

- The allowance for token transfer from the account at address $addr_1$ by the account at address $addr_2$ is stored at slot

$$keccak(uint256(addr_2).keccak(uint256(addr_1).uint256(3)))$$

In the above, the use of the keccak function to obtain the storage locations of the balances and allowance mimics how the storage is allocated by a compiler of the Solidity language. It utilizes the fact that the population of data in the storage space is sparse, and properties of a secure hash function such as collision avoidance, to avoid the mapping of different data to the same storage slot.

Table 1. The functions provided by the token contract to its users

total_supply_func	Query the total supply of the token
balance_of_func	Query the balance of a specific owner of the token
allowance_func	Query the amount of tokens an owner allows a spender to spend
transfer_func	Transfer a specified amount of tokens to a specified user
transfer_from_func	Transfer a specified amount of tokens from a specified user to a specified second user
approve_func	Approve transfer of a specified amount of tokens by a spender

4.2 The Code Layout of the Contract

The code layout of the token contract is shown in Fig. 6. The code is organized as a *Block* (cf. Fig. 4) consisting of the functions in the user interface, the utility functions that support the implementation of the contract, and the dispatcher statement that directs each contract call to the specific function invoked. There are altogether 19 functions. The functions in the user interface and their description are given in Table 1. Below, we selectively elaborate on the dispatcher statement and the interface function *transfer_func*.

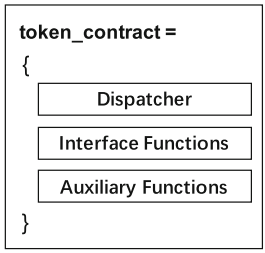

Fig. 6. The code layout of the token contract

```
if gt(callvalue(), 0) { revert(0, 0) }
switch selector_func()
case 0x10991a86 /* "balance_of_func(address)" */ {
  return_uint_func(balance_of_func(decode_as_address_func(0)))
}
...
case 0xb513186f /* "transfer_func(address,uint256)" */ {
  transfer_func(decode_as_address_func(0), decode_as_uint_func(1))
}
default { revert(0, 0) }
```

Fig. 7. The dispatcher statement

The Dispatcher. A call to the token contract essentially triggers the execution of the dispatcher statement. The code of the dispatcher statement is given in Fig. 7. It is first checked that no money is transfered to the contract using the condition that the value of the call should not be greater than zero. The value of the call as an unsigned integer is retrieved using the builtin function *callvalue*. Then, the function to which a call should be directed is obtained using the function *selector_func* and the switch statement. The function *selector_func* (also included in the implementation) computes the first 4 bytes of the input data to the call – these 4 bytes represent the keccak-256 hash of the signature of the function to be invoked. The subsequent chunks of the input data (of 32 bytes each) contain the arguments to be passed to the specific function invoked. The i-th argument is retrieved using *decode_as_uint_func(i)* or *decode_as_address_func(i)*. In addition to decoding an argument from the input data (or call data), the latter also checks that the decoded argument is in the form of an account address (of 160 bits). The function *return_uint_func* signals the exit of the currently executing contract, with the result placed at bit 0 in the memory of the execution engine. In case the caller attempts to send ether to the contract, or the invoked function is not found, the state is reverted using the builtin function *revert*.

```
function transfer_func(to, amount) {
  deduct_from_balance_func(caller(), amount)
  add_to_balance_func(to, amount)
  log(1, caller(), to, amount)
}
```

Fig. 8. The function `transfer_func`

The Function transfer_func. The code of the function *transfer_func* (the functionality of which is informally explained in Table 1) is given in Fig. 8. In the function body, the function *deduct_from_balance_func* is first invoked to deduce the specified amount of tokens from the caller account. The function *add_to_balance_func* is then invoked to add the same amount of tokens to the destination account of the transfer. Finally, the transfer event is logged with topic 1, together with the caller of *transfer_func*, the destination of the transfer, and the amount of transfered tokens as parameters.

In Fig. 8, *log* is a builtin function [8]. On the other hand, *deduct_from_balance_func* and *add_to_balance_func* are part of the contract implementation. The latter function makes use of a function for safe addition (*safe_add_func*) to avoid overflow when increasing the balance of the destination account.

The Readability of Yul Code. It is demonstrated by Figs. 7 and 8 that smart contract code in Yul has a greater level of readability than low-level instructions. This benefits the intuitiveness level of formal verification.

5 Verification of the Token Contract

We prove the guarantees of calling *each function of the token contract* in the ERC20 interface (c.f., Table 1) in Isabelle/HOL. To this end, we first establish the guarantees of all the utility functions that are used to implement the interface functions. Below, we *selectively* present our results.

5.1 The Guarantees of the Utility Functions

Below, we present the theoretical result about the guarantees of the utility function for safe addition. This function is used by the function *add_to_balance_func* that increases the balance of a specified account by a specified amount (cf. Fig. 8).

```
lemma safe_add_body_correct:                                          1
 "⟦ n > 4;
    ∀fid. builtin_ctx fid ≠ None
          ⟶ (context0 g ++ fctx) fid = builtin_ctx fid;               4
    l a_id = Some (IntV a); l b_id = Some (IntV b);
    is_uint256 a; is_uint256 b
   ⟧ ⟹                                                                7
    (a + b < two256 ∧
     eval_statement g l fctx (body_of safe_add_func) n
      = Normal (g, l(r_id:=Some (IntV (a+b))), RegularMode)           10
     ∨
     a + b ≥ two256 ∧
     eval_statement g l fctx (body_of safe_add_func) n               13
      = Exit (RevertExit g 0 0)
    )"
```

In the above, the identifiers *a_id* and *b_id* are the parameters of the function *safe_add_func*. The lemma *safe_add_body_correct* asserts that if *a_id* and *b_id* have values a and b, respectively, that are 256-bit unsigned integers, then evaluating the body of *safe_add_func* yields $a+b$ (that is stored in the return variable *r_id*) if $a+b$ does not exceed $2^{256}-1$, and an exception reverting the state otherwise. The condition $n > 4$ is imposed only because when fully evaluating the body of *safe_add_func* in the semantics, the counter n decreases 5 times. The evaluation would result in an error for any $n \leq 4$. The condition at lines 3 and 4, on the other hand, requires that each identifier of a builtin function should indeed be mapped to the right builtin function by *context*0 *g* ++ *fctx*. Here, *builtin_ctx* is a pre-defined mapping from each identifier of a builtin function to the builtin function, and *context*0 *g* is the map for all the globally available identifiers.

The proof of the lemma *safe_add_body_correct* is by case analysis on the truth of $a+b < two256$, and by simplification using the semantics of Yul. We omit the discussion of the statement/proof of the lemmas for the other utility functions.

Remark 1. The guarantees for the functions of the token contract (e.g., *safe_add_func*) correspond to the notion of *total correctness* [10] – it is stated that under specific conditions the execution terminates, resulting in global and local states that satisfy specific conditions.

5.2 The Guarantees of Calls to the Token Contract

We first introduce a series of definitions that are used to formulate the theoretical results about the calls to the contract. The term "*keccak_base_key base key*" is defined to give the keccak-256 hash value of the list of 64 bytes where the first 32 bytes are those of the value *key* and the next 32 bytes are those of the value *base*. The term "*memory_values m addr sz*" is defined to give the list of bytes (each as an integer) in the memory m starting at the address *addr* and ending at the address $addr + sz - 1$. The term "*sel_val cd val*" is defined to say that the signature hash of the function to which the current call is dispatched is *val*. The term "*uint_arg_idx cd idx val*" is defined to say that the *idx*-th

argument value in the input data *cd* of the call is the unsigned integer *val*. The term "*addr_arg_idx cd idx val*" is defined to require that in addition to *uint_arg_idx cd idx val*, the *idx*-th argument has the form of an account address. For the account *acc*, storage offsets o_1 and o_2, balances b_1 and b_2, and the amount a of tokens, "*upd_bal acc* o_1 o_2 b_1 b_2 a" is written for $acc(\!|storage := (storage\ acc)(o_1 := IntV(b_1 - a), o_2 := IntV(b_2 + a))|\!)$.

```
n > k; length args = 7; length gs = 8; length ls = 8;
argvs = [IntV gas, IntV addr, IntV val,                                  2
         IntV offt_in, IntV sz_in, IntV offt_out, IntV sz_out];
∀i. i ≥ 0 ∧ i < 7 ⟶
    eval_expression (gs!i) (ls!i) fctx (args!i) n                        5
      = Normal (gs!(i+1), ls!(i+1), (argvs!i));
g' = gs!7; l' = ls!7;
(context0 g' ++ fctx) b_call_id = Some (GBuiltinV Call);                 8
∀fid. context0 g' fid = builtin_ctx fid
```

Fig. 9. The list *assms* of assumptions

A number of conditions are shared as assumptions by multiple theoretical results about calls to contracts. We write

$$assms\ args\ argvs\ gas\ addr\ val\ offt_{\text{in}}\ sz_{\text{in}}\ offt_{\text{out}}\ sz_{\text{out}}\ gs\ ls\ g'\ l'\ fctx\ n\ k$$

for the list of assumptions shown in Fig. 9. Here, *args* is a list of 7 argument expressions for a contract call, *argvs* is a list of 7 argument values for the same call, *addr* is an account address, *val* is an amount of money, $offt_{\text{in}}$ and $offt_{\text{out}}$ are two memory offsets, sz_{in} and sz_{out} are two counts of memory bytes, *gs* is a list of global states, *ls* is a list of local states, and n and k are two natural numbers. The condition spanning lines 2 and 3 says that the list *argvs* is obtained by wrapping the series of integer values provided using the type constructor *IntV*. The condition spanning lines 4–6 says that the evaluation of the i-th argument expression yields the i-th argument value, turning the global and local states to the next ones in the respective lists *gs* and *ls*. The condition at line 8 says that after evaluating all the arguments (thereby reaching the global state g'), the identifier for the builtin function *Call* is still properly mapped to *Call* according to g' and the local function context $fctx'$. The condition at line 9 says that the global state g' properly maintains the mapping for the builtin functions.

Below, we present the theorem about the guarantees of each call to the token contract that invokes the function *transfer_func* (cf. Table 1), when the source account (the caller) has a sufficient amount of tokens to transfer, and the transfer does not lead to an overflow of the balance at the destination.

```
theorem normal_call_transfer:
 "⟦ assms args argvs gas addr 0 offt_in sz_in offt_out sz_out gs ls g' l' fctx n 17;
    current g' = accounts g' (address g');                                    3
    balance ((accounts g') (address g')) ≥ 0;
    code ((accounts g') addr) = Some token_contract;
    cd_0 = memory_values (memory g') offt_in (nat |sz_in|);                   6
    valid_mem (list_to_map cd_0) 4 64;
    sel_val cd_0 0xb513186f; addr_arg_idx cd_0 0 to_0; uint_arg_idx cd_0 1 a_0;
    o_1 = keccak_base_key 2 (address g'); o_2 = keccak_base_key 2 to_0;       9
    storage (accounts g' addr) o_1 = IntV b_1;
    ((storage (accounts g' addr)) (o_1 := IntV (b_1−a_0))) o_2 = IntV b_2;
    is_uint256 a_0; is_uint256 b_2; b_1 ≥ a_0; b_2 + a_0 < two256             12
 ⟧ ⟹
 eval_expression (gs!0) (ls!0) fctx (FunctionCall b_call_id args) (n_0 + 1)
 = Normal (                                                                   15
    g'(|memory_size := max (max (memory_size g') (offt_in + sz_in)) (offt_out + sz_out),
        current := if address g' = addr then upd_bal(current g', o_1, o_2, b_1, b_2, a_0)
                                        else current g',                      18
        accounts := (accounts g')(addr := upd_bal(accounts g' addr, o_1, o_2, b_1, b_2, a_0))
        logs := ListV (memory_values
                          (mem_upd_4 (memory g') 1 (address g') to_0 a_0)     21
                           0 128) # logs g' |),
    context0 g' ++ fctx_erc20, TrueV)"
```

In the theorem statement, the condition at line 3 says that *address* g' is indeed the address of the currently executing account in g'. The condition at line 5 requires that the code being called is that of the token contract (c.f. Fig. 6). The condition at line 6 says that the input data to the call (as obtained from the global state g' reached after the evaluation of the arguments) is cd_0. The condition at line 7 says that the input data to the call contains valid data after four initial bytes, for 64 bytes in a row – the argument values are contained in these bytes. The conditions at line 8 say that the signature hash for the function to be executed is the one for *transfer_func*, and the 0-th and 1-th arguments in the input data of the call are to_0 (the address of the destination account of the transfer) and a_0 (the amount of tokens to be transfered), respectively. The conditions at line 9 say that the storage offsets for the balances of the source and destination accounts of the transfer are o_1 and o_2, respectively. The conditions at line 10 and line 11 say that these two balances are b_1 and b_2, respectively. The latter condition is stated with consideration of the fact that if the destination account is the same as the source account, then the balance of the destination account decreases when the tokens have been sent but not received. The updated global state described in lines 16–22 reflects the change in the account balances due to the transfer, and the recording of the transfer in the log.

The proof of theorem *normal_call_transfer* is conducted using lemmas that connect the result of calling the token contract to the result of evaluating the function *transfer_func*. These latter lemmas are in turn based on lemmas about the guarantees of the utility functions (e.g., for safe addition, as shown in Sect. 5.1). Transformations are performed such that the resulting global state is described directly wrt. g' that is reached after evaluating the arguments for the call. Hence, for side-effectless argument expressions, it is also directly in terms of

the initial global state $gs\,!\,0$. The case where the source account does not have a sufficient amount of tokens to be transfered, or the transfer leads to an overflow of the balance at the destination, is covered by a separate theorem.

As a corollary, we have formally shown that a token transfer preserves the total amount of tokens, provided that there is no collision of the keccak-256 hash values of the addresses for all the accounts that own the token.

Remark 2. As is demonstrated in the theorem *normal_call_transfer*, the guarantees for the calls to the contract functions are formulated to precisely reflect all changes in the global and local states. This provides a solid basis for establishing further safety and security properties in a broad range (e.g., [15]).

Finally, if the caller of the contract attempts to send money to the contract, then the call is terminated with the effects on the states reverted.

theorem *call_with_money:*
"⟦$val_0 > 0$;
assms args argvs gas addr val_0 $offt_{\text{in}}$ sz_{in} $offt_{\text{out}}$ sz_{out} *gs ls g' l' fctx n* 7
⟧ ⟹
eval_expression (gs!0*) (ls*!0*) fctx (FunctionCall b_call_id args) (*$n_0 + 1$*)*
= Normal (g'⦇*memory_size := max (max (memory_size g')(*$offt_{\text{in}}$ *+* sz_{in}*))*
$(offt_{\text{out}} + sz_{\text{out}})$ ⦈,
l', FalseV)"

Note that the potential increase in the number of active memory bytes is not canceled, which is consistent with the semantics described in [17,28].

In the verification of the token contract in Isabelle/HOL, the contract code in Yul has been sufficiently comprehensible for it to be used as the reference for specifying the initial pre/post-conditions. These pre/post-conditions are further revised in the proving process – the formal proof helps make all the assumptions and effects associated with an invocation of the token contract explicit. Furthermore, since the dispatching logic of calls to specific functions is an integral part of the token contract at the IL-level, the dispatcher is naturally covered by the verification. This provides added confidence that the dispatcher does not contain errors that could have otherwise been introduced by a compiler.

6 Related Work

Verification of Smart Contracts by Theorem Proving. The strongly negative impact of errors and flaws of smart contracts motivated their verification by theorem proving. In [19], the EVM is formalized in Lem [22], and a few safety properties of simple contracts are proven in Isabelle/HOL based on formal definitions generated in this proof assistant. In [9], a program logic is defined to syntactically reason about properties of EVM bytecode. This development is based on the formalization of [19]. In [18], a semantics of EVM bytecode is defined in the K-framework, which provides the basis for program analysis and

theorem proving [23] for Ethereum smart contracts. In [17], a small-step semantics of EVM bytecode is defined (with partial mechanization in the F* language), and a few security properties are defined on the basis of this semantics for the verification of Ethereum smart contracts. In [5], a library of formal proofs is developed for Ethereum smart contracts in the Coq proof assistant, based on a demand-driven formalization of a Solidity-like language. In [30], a type system and a big-step semantics are defined (in Coq) for Lolisa – a Solidity-like programming language developed by the authors. In [14], an approach to verifying Hyperledger Fabric chaincode (in Java) in the KeY prover is proposed. The main idea is to extend KeY to handle the major API methods that are provided by the Hyperledger blockchain and used for writing the chaincode.

The developments mentioned above formalize smart contracts and prove their properties at either the user-language level or the machine-language level. In [24], an intermediate language, Scilla, is defined in Coq, for the analysis and verification of smart contracts. Unlike our development that leverages the existing intermediate language in the ecosystem of Ethereum, Scilla is a new language for which the translation from high-level languages like Solidity, and into low-level languages like EVM bytecode is yet to be defined.

Validation of Smart Contracts in General. Numerous developments have been carried out to validate smart contracts by non-theorem-proving means. For space reasons, the following discussion is non-exhaustive on these developments.

In [13], the role of refinement in verifying and preserving the correctness of smart contract designs (e.g., in the Event-B formalism) is discussed. In [20], the problem of verifying smart contracts is addressed by generating and solving horn clauses. In [16], a static analysis is proposed for Ethereum smart contracts, and the analysis comes with a soundness proof. In [12], the SPIN tool is leveraged to model check smart contracts. In [26], the target properties of a smart contract is expressed as patterns, and the verification/falsification of properties is performed by finding the corresponding patterns. In [21], a method of finding bugs in smart contracts via symbolic execution is proposed. In addition, hybrid approaches to the verification of smart contracts are proposed and used in the VaaS framework [7] and the CertiK project [1].

7 Conclusion

Formal verification can be applied to provide the highest level of correctness and security guarantee for smart contracts. The language used to realize the smart contract affects multiple aspects of the verification. Specifically, the use of an intermediate language (IL) ensures a relatively low level of complexity in formalizing the language itself (owing to the succinctness of the language features), a relatively high level of intuitiveness of the verification (owing to the existence of structured programming constructs), and a relatively high level of confidence on the verification result (owing to the partially reduced trust base).

To demonstrate some of these benefits, we present a concrete formal verification of an Ethereum smart contract at the IL-level, in a proof assistant. The

smart contract is an ERC20 token contract, which we realize in the Yul language, the formalization of which we revise to rectify its observed deviation from its informal specification. We prove the guarantees of calls to all the interface functions of the token contract in Isabelle/HOL. The development totals ∼10k lines of code (excl. code generated from Lem). In the verification, we take advantage of the good level of comprehensibility of Yul to devise the initial pre/post-conditions for the contract functions. These pre/post-conditions are then revised in the proving process, such that all the assumptions and effects for the contract functions are precisely identified. The complexity of the formal proof is partially reduced by the simplicity of Yul and its formal semantics relative to a high-level language. The overall approach applies easily to other Ethereum contracts.

Potential directions for future work include support for easier smart contract proofs for Yul via proof automation and program logics, as well as refinement verification of Yul contracts to preserve guarantees down to the lowest level.

Acknowledgments. This work was supported by the National Key R&D Plan (2017YFB1301100), National Natural Science Foundation of China (61876111, 61572331, 61602325), Capacity Building for Sci-Tech Innovation – Fundamental Scientific Research Funds (025185305000), and the Youth Innovative Research Team of Capital Normal University. We thank the anonymous reviewers for their valuable comments that helped with the improvement of this paper.

References

1. CertiK. https://certik.org/
2. ERC20 standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard
3. Eth-isabelle. https://github.com/pirapira/eth-isabelle
4. Solidity (v0.5.8). https://solidity.readthedocs.io/en/v0.5.8/
5. Token libraries with proofs. https://github.com/sec-bit/tokenlibs-with-proofs
6. Understanding the DAO attack. http://www.coindesk.com/understanding-dao-hack-journalists/
7. VaaS. https://sso.beosin.com/#/?vaas
8. Yul. https://solidity.readthedocs.io/en/v0.5.8/yul.html
9. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), pp. 66–77 (2018)
10. Apt, K.R.: Ten years of Hoare's logic: a survey - part 1. ACM Trans. Program. Lang. Syst. **3**(4), 431–483 (1981)
11. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: 6th International Conference on Principles of Security and Trust (POST), pp. 164–186 (2017)
12. Bai, X., Cheng, Z., Duan, Z., Hu, K.: Formal modeling and verification of smart contracts. In: 7th International Conference on Software and Computer Applications (ICSCA), pp. 322–326 (2018)
13. Banach, R.: Verification-led smart contracts. In: Proceedings of 3rd Workshop on Trusted Smart Contracts (2019)

14. Beckert, B., Herda, M., Kirsten, M., Schiffl, J.: Formal specification and verification of Hyperledger Fabric chaincode. In: Third Symposium on Distributed Ledger Technology (SDLT) (2018)
15. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
16. Grishchenko, I., Maffei, M., Schneidewind, C.: Foundations and tools for the static analysis of Ethereum smart contracts. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 51–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_4
17. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
18. Hildenbrandt, E., et al.: KEVM: a complete formal semantics of the Ethereum virtual machine. In: 31st IEEE Computer Security Foundations Symposium (CSF), pp. 204–217 (2018)
19. Hirai, Y.: Defining the Ethereum virtual machine for interactive theorem provers. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
20. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: 25th Network and Distr. System Security Symposium (NDSS) (2018)
21. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 254–269 (2016)
22. Owens, S., Böhm, P., Nardelli, F. Z., Sewell, P.: Lem: a lightweight tool for heavyweight semantics. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 363–369. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22863-6_27
23. Park, D., Zhang, Y., Saxena, M., Daian, P., Rosu, G.: A formal verification tool for Ethereum VM bytecode. In: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT (FSE), pp. 912–915 (2018)
24. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. CoRR, abs/1801.00687 (2018)
25. Szabo, N.: Smart contracts (1994). http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html
26. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: practical security analysis of smart contracts. In: ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 67–82 (2018)
27. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_7
28. Wood, G.: Ethereum: a secure decentralised generlised transaction ledger. https://gavwood.com/paper.pdf
29. Yaga, D., Mell, P., Roby, N., Scarfone, K.: Blockchain technology overview. Technical report, NISTIR 8202 (2018)
30. Yang, Z., Lei, H.: Lolisa: formal syntax and semantics for a subset of the solidity programming language. CoRR, abs/1803.09885 (2018)