MichelsonLiSA: A Static Analyzer for Tezos

Luca Olivieri
University of Verona
Corvallis S.r.l.
Verona/Padua, Italy
luca.olivieri@univr.it

Thomas Jensen
INRIA
Rennes, France
thomas.jensen@inria.fr

Luca Negrini
Ca' Foscari University of Venice
Corvallis S.r.l.
Venice/Padua, Italy
luca.negrini@unive.it

Fausto Spoto
University of Verona
Verona, Italy
fausto.spoto@univr.it

***Abstract*—Smart contracts are immutable code deployed in a blockchain, whose execution modifies its global state. Code immutability leads to immutable bugs. To prevent such bugs, static program analysis infers information about the behavior of the code, statically, before code execution and deployment. This paper introduces MichelsonLiSA, a static analyzer based on abstract interpretation for the verification of smart contracts written in the Michelson low-level language of the Tezos blockchain. It applies MichelsonLiSA to the identification of security issues arising from cross-contract invocations.**

***Index Terms*—Program analysis, smart contracts, injection, blockchain, Michelson language, Tezos.**

This is a pre-print version of *L. Olivieri, T. Jensen, L. Negrini and F. Spoto, "MichelsonLiSA: A Static Analyzer for Tezos," 2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops), Atlanta, GA, USA, 2023, pp. 80-85, doi: 10.1109/PerComWorkshops56833.2023.10150247.*

I. Introduction

Blockchains are trustless distributed systems, whose peers do not trust each other. Data and code in blockchain are immutable and accessible to everybody, at any time, for transparency and to achieve trustlessness. However, malicious peers can fully and directly access data and code. This code, known as smart contracts, are executable programs that modify the world state of the blockchain. They are historically applied, for instance, to the management of financial assets.

Immutable code means immutable bugs, exploitable for security attacks. Static program analysis helps here, by inferring the run-time behavior of the smart contracts, statically, before code execution and deployment, thus preventing the exploitation of the bugs.

This paper introduces MichelsonLiSA, a static analyzer based on abstract interpretation [1] that verifies smart contracts written in the Michelson low-level language of the Tezos blockchain. It builds on LiSA [2], [3] (Library for Static Analysis), a framework that can be adapted to different programming languages, mainly variable-based.

The contributions of this paper are (1) the definition of a translation from the stack-based form of Michelson to a variable-based static single assignment (SSA) language, better suited for static analysis in LiSA; (2) a new static analysis for identifying security issues related to cross-contract invocations (a form of code injection), that can be performed over a preliminary taint analysis. The resulting analyzer is available at https://github.com/lisa-analyzer/michelson-lisa.

```
parameter (pair int int) ; # Parameter declaration: two
     integers
storage int ; # Storage declaration
code { # Code declaration

  CAR ; # Push the input parameter to the stack and discard
       the current storage value
  UNPAIR ; # Pop the input pair from the stack, split it
       into two integers and push them on the stack instead
  ADD ; # Pop the two integers and push their sum instead

  NIL operation ; # Push an empty list of operations
       required to end the contract

  PAIR ; # Build the final stack: a pair consisting of a
       list of operations and the value to keep in storage (
       in this case, the result of addition)
}
```

Fig. 1: A Michelson smart contract to add two input integers and store their sum in blockchain.

Paper structure: Sec. II introduces the Michelson language. Sec. III describes MichelsonLiSA and its design choices, such as the the SSA form of the code. Sec. IV defines the cross-contract invocation analysis implemented on MichelsonLiSA. Sec. V highlights the limits of the analysis. Sec. VI reports related work and concludes.

II. The Michelson Language

The Tezos blockchain provides a toolbelt (IDEs, command-line interface, transaction explorer) allowing programmers to implement smart contracts in high-level program languages such as Python, OCaml or JavaScript. All such languages are compiled into a unique native low-level language, called Michelson, that is then deployed in blockchain. It is a domain-specific language, statically typed and stack-based. It has no fields nor global variables. Its instruction set is low-level but Turing-complete. Michelson smart contracts are specified by three components: (1) a parameter declaration (explicitly typed input); (2) a storage declaration (explicitly typed blockchain store locations); and (3) a code declaration (a sequence of bytecode instructions). Technically, the input is a single value that specifies the required inputs for executing the code. However, the use of aggregate types (e.g., `pair` and `or`) enables the specification of several inputs, as Fig. 1 shows.

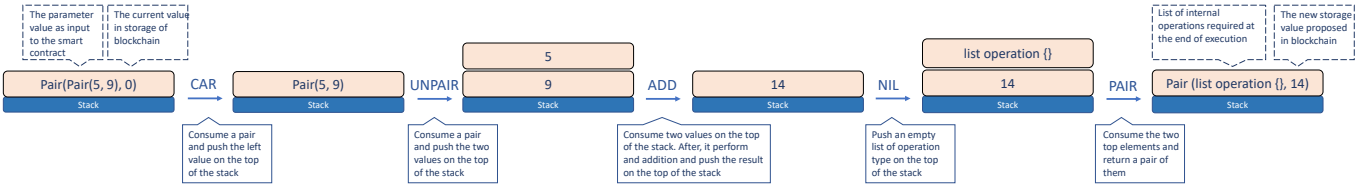

Fig. 2: An example of execution of the smart contract in Fig. 1.

Michelson's instructions pop and/or push stack elements. A smart contract execution request (*invocation*) specifies the address of the smart contract in blockchain and its input. The execution starts from a stack whose only element is a pair of the input and of the current value of the storage of the contract. Fig. 2 shows an example of execution of the contract in Fig. 1, with input `Pair(5,9)`, assuming that the current value of the storage of the contract is `0`: the initial stack contains the singleton value `Pair(Pair(5,9),0)`. Note that the user provides the input, while the blockchain protocol retrieves the storage value from the blockchain state. The first instruction, `CAR`, splits the pair and projects it on its first component `Pair(5,9)` (the input), that is then pushes on the stack: the current storage value is discarded. The subsequent `UNPAIR` instruction decomposes `Pair(5,9)` into its two components `5` and `9`, that are pushed on the stack. The `ADD` instruction computes their sum `14`, that becomes the stack top. The `NIL` instruction pushes an empty list of operations to perform at the end of the execution and the final `PAIR` instruction boxes the list and the result into a pair: the result of the execution. The blockchain protocol will store the second component (`14`) in the storage of the contract, for future use.

Michelson has around 100 instructions[1]: stack manipulations (`PUSH`, `DROP`, `SWAP`), high-level data structures creators and modifiers (`MAP`, `UPDATE`, `SIZE`), arithmetic operations (`SUM`, `SUB`, `AND`), control flow modifiers (`IF`, `LOOP`), and blockchain-specific operations (see Tab. I). This paper focuses on the most interesting ones.

TABLE I: A subset of the Michelson blockchain instructions.

Instruction	***Description***
`ADDRESS`	pop a contract value from the stack and push the address of that contract to the stack.
`AMOUNT`	push the amount of the current transaction to the stack.
`BALANCE`	push the current amount held by the executing contract to the stack.
`CHAIN_ID`	push the chain identifier to the stack (an identifier for a chain, used to distinguish the test and the main chains).
`CONTRACT`	replace the top of the stack after cast to a contract type.
`CREATE_CONTRACT`	push a contract creation operation to the stack. It allows a contract to create another contract.
`IMPLICIT_ACCOUNT`	push the address of a new implicit account[2] to the stack.
`LEVEL`	push the current block level to the stack.
`NOW`	push the current block timestamp to the stack.
`SELF`	push the current contract as element of type `contract` to the stack.
`SELF_ADDRESS`	push the address of the current contract to the stack.
`SENDER`	push the contract that started the current internal transaction to the stack.
`SET_DELEGATE`	push a delegation operation to the stack. An account can delegate their rights to participate in consensus and in governance to another account.
`SOURCE`	push the contract that initiated the current transaction to the stack.
`TOTAL_VOTING_POWER`	push the total voting power value of all contracts to the stack. It is based on the staking balances of the contracts involved in a given voting period[3].
`TRANSFER_TOKENS`	push a transaction operation to the stack.
`VOTING_POWER`	push the voting power value of a contract to the stack. It is based on the staking balance of the contract in a given voting period.

III. An Overview of MichelsonLiSA

MichelsonLiSA is a static analyzer for Tezos smart contracts written in Michelson, based on abstract interpretation. It relies on LiSA [2], [3], that implements standard components for abstract interpretation, such as a *control-flow graphs* (CFG), a framework with several built-in static analyses (such as type analysis and information flow analysis), and fixpoint algorithms. MichelsonLiSA implements additional components managing the translation from Michelson to the intermediate representation (IR) of LiSA. Next section presents these components and the challenges faced to support Michelson.

A. Parsing and CFG Construction

The first step to verify and analyze a smart contract is to parse its syntax. Michelson has an official grammar, but that misses some syntactic sugar (such as annotations, use of brackets, smart contract structure, and macros) widely used in real-world contracts. Hence, we enriched the grammar and implemented it with ANTLR [4], a popular parser generator that builds a lexer and a parser from the grammar. The lexer reads the source and produces a sequence of meaningful strings (*lexemes*); the parser takes them as input and builds an abstract syntax tree (*AST*) that reflects the grammar, or otherwise rejects the program with a syntax error. From the AST, it is possible to build an analyzer able to perform syntactic checks, but the AST has limited expressiveness. However, the AST is a good starting point for building a CFG, from where the missing expressiveness can be recovered. The CFG building phase starts after the parsing of Michelson source code into syntax trees. The CFG builder translates the code into an IR based on SSA form [5], [6] and builds the LiSA CFGs.

B. Intermediate Representation in SSA Form

Michelson is a low-level, stack-based language. According to [7], the use of a stack makes it difficult to apply standard static analysis techniques. Therefore, an IR is necessary to provide an efficient model for the analysis in terms of transformation time and produced code. LiSA is designed to handle

[1] https://tezos.gitlab.io/active/michelson.html#core-instructions
[2] http://tezos.gitlab.io/active/glossary.html#implicit-account
[3] http://tezos.gitlab.io/active/glossary.html#voting-period

```
0: PUSH int 23;
1: PUSH int 13;
2: SUB;
3: DROP;
4: PUSH int 23;
5: PUSH int 13;
6: PAIR;
7: UNPAIR;
8:
```

(a) Michelson code

```
0: []
1: [v1]
2: [v1,v2]
3: [v3]
4: []
5: [v4]
6: [v4,v5]
7: [v6]
8: [v7, v8]
```

(b) Symbolic stack

```
0: v1 = PUSH(int, 23);
1: v2 = PUSH(int, 13);
2: v3 = SUB(v1,v2);
3: DROP(v3);
4: v4 = PUSH(int, 23);
5: v5 = PUSH(int, 13);
6: v6 = PAIR(v4,v5);
7: v7 = get_left(v6);
   v8 = get_right(v6);
8:
```

(c) SSA form

Fig. 3: Example of transformation to SSA form.

```
0: IF
1: { # True branch
2:   PUSH int -1;
3: }
4: { # False branch
5:   PUSH int 7;
6: }
7:
```

(a) Michelson code

```
0: stack  [v0]
1: stack  []
2: stack1 []
3: stack1 [v1]
4: stack1 [v1]
5: stack2 [v2]
6: stack1 [v1],
   stack2 [v2]
7: stack  [v3]
```

(b) Symbolic stack

```
0: IF(v0)
1: { # True branch
2:   v1 = PUSH(int,-1);
3: }
4: { # False branch
5:   v2 = PUSH(int, 7);
6: } v3 = phi(v1, v2) #
       Junction point
7:
```

(c) SSA form

Fig. 4: Example of transformation of a conditional into SSA form, with a junction point. The ϕ-function is written as `phi`.

a generic program language but is currently variable-oriented. For this reason, we translate the stack-based representation into a variable-based IR, by using the SSA form. The translation maps each Michelson instruction into a list of MichelsonLiSA instructions, by using new fresh variables. It tracks, abstractly, the values on the stack through a symbolic stack of such variables. Stack elements are thus identified through symbolic names instead of their exact values. Instructions that push values on the stack are translated into variable assignments, with fresh variables standing for stack elements, each assigned exactly once. Instructions that pop from the stack are translated into MichelsonLiSA instructions taking those variables as parameters. Some instructions can both pop and push stack elements. Fig. 3 shows an example of translation to SSA for some common instructions. `PUSH <type> <data>` pushes a constant of the declared type: it is translated with a fresh new variable that gets assigned a constant of a declared type. `SUB` consumes its two operands from the stack and pushes their difference instead: it is translated as a function that receives the operands as arguments and yields their difference. `DROP` pops and discards the top of the stack: it is translated with a function with no return value and in this case the stack top is discarded from the symbolic stack. `PAIR` consumes the two topmost stack elements, packs them into a pair that pushes on the stack instead: it is translated as a function with two arguments that yields the pair. `UNPAIR` pops a pair, splits it and pushes its two components instead: it is translated with two functions, that select the two components and store them into fresh new variables.

Michelson includes instructions for conditionals, such as `IF`, and for iteration, such as `LOOP`, both leading to branches and junction points. For junctions, SSA reconciles distinct values of the same variable, arising along different paths, through ϕ-functions [6], as shown in Fig. 4. The idea is to translate instructions separately along each path using disjoint

```
parameter (pair int int);
storage int;
code {
  CAR;
  DUP;
  UNPAIR;
  COMPARE;
  GT;
  IF
  {  # True branch
    UNPAIR;
    ADD;
  }
  { # False branch
    UNPAIR;
    SUB;
  }

  NIL operation;
  PAIR;
}
```

(a) Michelson code

```
v0 = parameter_storage();
v1 = CAR(v0);
v2 = DUP(v1);
v3 = get_left(v2);
v4 = get_right(v2);
v5 = COMPARE(v3, v4);
v6 = GT(v5);
IF(v6)
{ # True branch
  v7 = get_left(v1);
  v8 = get_right(v1);
  v9 = ADD(v7, v8);
}
{ # False branch
  v10 = get_left(v1);
  v11 = get_right(v1);
  v12 = SUB(v10, v11);
}
v13 = phi(v9, v12);

v14 = NIL(operation);
v15 = PAIR(v14, v13);
```

(b) SSA form

Fig. 5: Michelson code and its SSA form. Given a parameter, the contract performs an addition if the first component of the input pair is larger than the second one; otherwise, it performs a subtraction. The result is a pair consisting of an empty list of operations and of the new value for the storage data.

sets of variables, and then merging them at the junction point into unique fresh variables, each standing for the same stack element along distinct paths.

Fig. 5 reports the translation of a realistic Michelson contract into SSA and the corresponding symbolic stack. Michelson smart contracts interact with the context of Tezos where they execute. In fact, at the beginning of their execution, the stack holds a pair of the input value and of the current storage value. This must be made explicit in the SSA translation, as in Fig. 5 with `v0 = parameter_storage()`. Instrumentation is needed for data structures as well. Namely,

Michelson supports high-level data structures (sets, lists, maps, optionals) and has specific instructions to operate on them, such as `ITER`, `LOOP_LEFT` and `IF_CONST`. These typically push additional elements on the stack. For instance, `ITER` consumes a collection from the stack and applies a set of instructions to each of its elements. These gets simulated, in SSA, by using assignments to additional variables.

C. Analysis

MichelsonLiSA analyzes the SSA code once it is put inside a control-flow graph (CFG), that expresses the control structure of the code. LiSA uses a general design for CFGs, so that they apply to different programming languages. Namely, a LiSA CFG has nodes standing for syntactic statements and edges that represent flow of control among them. Each statement is then rewritten into *symbolic expressions*, that is, expressions in the internal language of LiSA used to implement the semantics of the statements in a language-independent way. Such symbolic expressions are low-level atomic instructions that can be composed to model the semantics of many source languages. The concrete behaviors of a program (the *concrete semantics*) expressed through symbolic expressions are then approximated (i.e., abstracted) into an abstract version of them (the *abstract semantics*) [1]. The different forms of abstraction over the semantics are called *abstract domains*. The abstract semantics of a CFG is defined as a fixpoint, that will be reached in a finite number of iterations if the *abstract domain* has finite height, or through the use of widening operators [1]. The abstract states computed during that fixpoint computation are a sound over-approximation of the concrete semantics of the program, that a checker can use to issue warnings.

IV. Cross-Contract Invocation Analysis

This section shows an example of the analysis that we have implemented in MichelsonLiSA. It spots potential situations where an arbitrary code injection is possible, leading to the execution of arbitrary code in blockchain. Namely, the methods of a smart contract C in blockchain can be executed directly, with a call originated from outside the blockchain, or indirectly, as an internal *cross-contract* call from inside another contract. This latter case is used for instance to query the state of C or to execute one of its external methods. A typical example is the execution of a token transfer from a contract A to a contract C. This requires an internal *cross-contract* call which, in the case of token transfer, in Michelson can be performed by using the `TRANSFER_TOKENS` instruction[4]. C does not necessarily have to be hardcoded as target contract into contract A, since it can be passed as a parametric input to A (by specifying a `contract` type element in the parameter declaration of the contract A) and then used by `TRANSFER_TOKENS`. However, inputs coming from outside the blockchain are untrusted: in permissionless blockchains such as Tezos, any user can provide inputs, while potentially being anonymous. This is fine as long as the method of C that gets invoked is not redefined. Otherwise, it is possible to induce the execution of the arbitrary code placed in its redefinition. That code could move assets or currencies among contracts, in a way that was not expected.

In order to spot such dangerous code, we have implemented and applied a taint analysis [8] to Michelson code, inside MichelsonLiSA. Taint analysis is an instance of information flow analysis [9], that detects if untrusted information explicitly flows from some *source* into critical program points, called *sink*. It has been already successfully applied to different industrial contexts [8], [10], [11]. In our case, we use it to spot untrusted cross-contract invocations (UCCI). Namely, the sources are calls to `parameter_storage()` (see Sec. III-B) while the sinks are the parameter of cross-contract `TRANSFER_TOKENS` that holds the contract that receives that call.

The experimental evaluation of our analysis was performed on 1000 Michelson smart contracts containing the instruction `TRANSFER_TOKENS`, randomly retrieved from [12]. This resulted in 770060 lines of code (LoCs). The testing environment was a HP EliteBook 850 G4 equipped with an Intel Core i7-7500U at 2,70/2,90 GHz and 16 GB of RAM memory running Windows 10 Pro 64bit, Oracle JDK version 13. Readers who want to run the experiments and inspect the results can download the code[5] and follow the instructions contained in the file *README.md*.

TABLE II: Cross-contract analysis of our benchmark.

Analysis	Exec. time	Avg. time per file	# Warnings
UCCI	2 hours 32 min 8 sec	9.12 sec	2834

```
parameter address ;
storage unit ;
code {
  DUP ;
  CDR ;
  SWAP ;
  CAR ;
  DUP ;
  NIL operation ;
  SWAP ;
  CONTRACT unit ;
  { IF_NONE { PUSH
       unit Unit ;
       FAILWITH } {} }
       ;
  AMOUNT ;
  PUSH unit Unit ;
  TRANSFER_TOKENS ;
  CONS ;
  SWAP ;
  DROP ;
  PAIR
}
```

(a) Michelson smart contract

```
v0 = parameter_storage();
v1 = DUP(v0);
v2 = CDR(v1);
SWAP();
v3 = CAR(v0);
v4 = DUP(v3);
v5 = NIL();
SWAP();
v6 = CONTRACT(v4);
IF v7 = extract_value(v6) is
     None {
  v8 = PUSH("Unit");
  FAILWITH();
}
v9 = AMOUNT();
v10 = PUSH("Unit");
v11 = TRANSFER_TOKENS(v10,v9,
     v7);
v12 = CONS(v11, v5);
SWAP();
DROP();
v13 = PAIR(v12, v2);
```

(b) Michelson IR in SSA form

Fig. 6: Smart contract *expruqYPRHnQyNih8sK1vhNLRBLx-37VeuZ3T58SWaxPj5WwbCQJb2V.tz*.

[4]https://tezos.gitlab.io/michelson-reference/#instr-TRANSFER_TOKENS

[5]`git clone -branch brain2023 https://github.com/lisa-analyzer/michelson-lisa.git`

Tab. II reports the results of the experimental evaluation. In terms of time, the analysis requires less than 10 seconds per smart contract, on average. About the results, the analysis issues warnings about 2834 cross-contract invocations distributed in 781 smart contracts. Since Michelson is a low-level language, it is rather difficult to reverse-engineer the code: high-level information is lost after compilation. At the end of the analysis, MichelsonLiSA can provide an additional reports containing the analyzed CFGs in various formats (html, dot, etc.) with details about the computed abstract states. This allows one to check, for each program point, which variables the analysis infers as tainted and which it doesn't. However, for a deep manual investigation capable of identifying any over-approximations and false positives, one should manually recompute the entire execution stack for each single instruction and check if its execution in the real world can lead to a tainted value or not compared to the MichelsonLiSA report. This activity is time consuming given the poor readability of Michelson and the complexity of some contracts. For this reason, we could not manually investigate each of these files and compute the rate of true to false positives. In the following, we discuss few examples that show the strength of the analysis.

We have identified a true positive warning, that is, a dangerous UCCI in a smart contract inside our benchmark. Figure 6 shows the smart contract. MichelsonLiSA detects a flow leading to an UCCI. It begins at `v0 = parameter_storage()`, the information is propagated into `v3 = CAR(v0)` and then into `v4 = DUP(v3)`. At this point, the untrusted information flows into `v6 = CONTRACT(v4)`. The instruction `CONTRACT` allows to cast from an address to a typed contract. In terms of our analysis, when untrusted information flows into `CONTRACT`, this means that a potentially untrusted address is cast to a contract, which in turn will be untrusted thanks to the information flow propagation. Going forward, the untrusted information is propagated into `v7 = extract_value(v6)`, which finally flows into `TRANSFER_TOKENS(v10,v9,v7)`, where the analysis detects that an untrusted contract is invoked. The instruction `TRANSFER_TOKENS` allows one to transfer an amount of tokens to a target contract with its parameter. In this case, the amount of currency is loaded by `v9 = AMOUNT()`, the parameter by `v10 = PUSH("Unit")` and in both the information is propagated in a trusted way because it is not coming nor inferred from sources considered untrusted by our analysis. The target contract `v7`, as just shown, is the propagation result of the untrusted information. Then, a transaction of an amount is made to an untrusted contract.

We have identified a true negative in a smart contract in our benchmark. Figure 7 shows its code. MichelsonLiSA does not detect any untrusted flow that leads to an UCCI. The analysis starts by propagating the parameter and storage inputs in `v0 = parameter_storage()`. The untrusted value of `v0` is used only after `TRANSFER_TOKENS` (the sink for the analysis), then it cannot affect the cross-contract invocation. Indeed, this sink targets only a contract derived by the hardcoded address declared in `v3 = PUSH("tz1Rwo...Gv1ir")`, therefore the address cannot be changed by any arbitrary input, which ensures its safety against UCCIs.

```
1  parameter unit ;
2  storage unit ;
3  code {
4    CDR ;
5    NIL operation ;
6    PUSH address "tz1Rwo
         ...Gv1ir" ;
7    CONTRACT unit ;
8    IF_NONE { FAILWITH }
9    { BALANCE ;
10   UNIT ;
11   TRANSFER_TOKENS ;
12   CONS ;
13   PAIR }
14 }
```

(a) Michelson smart contract

```
v0 = parameter_storage();
v1 = CDR(v0);
v2 = NIL();
v3 = PUSH("tz1Rwo...Gv1ir");
v4 = CONTRACT(v3);
IF v5 = extract_value(v4) is
     None {
  FAILWITH();
} ELSE {
  v6 = BALANCE();
  v7 = UNIT();
  v8 = TRANSFER_TOKENS(v7,v6,
       v5);
  v9 = CONS(v8,v2);
  v10 = PAIR(v9,v1);
}
```

(b) Michelson IR in SSA form

Fig. 7: Smart contract *exprthPm93Nt4TBdDSd9LVG829YcgbK9VKE4TRDXtZiU8Fv7gFEBod.tz*.

V. Limits of the Analysis and Approximation

Abstract interpretation is based on approximations. It approximates the *concrete semantics* with an *abstract semantics*. The abstraction is a necessary step to perform analyses that detect otherwise undecidable properties, that is, abstractions trade precision for decidability. Moreover, different abstractions can be used in abstract interpretation to prove program properties. In particular, our analysis applies an over-approximating abstraction to detect UCCIs and leads to false alarms: warnings that do not correspond to any real issue. Consider the code in Fig. 8. Its untrusted input is used to index a map containing hardcoded addresses. The analysis starts by propagating the parameter and storage inputs in `v0 = parameter_storage()`. The untrusted information of `v0` flows into `v1 = CAR(v0)` and then into `v3 = GET(v1,v2)`. Given a key and a map, the instruction `GET` retrieves a value from the map. Therefore, the input parameter is used to select a hardcoded address from a map. However, our analysis propagates the untrusted information to `v3` because at least one of the two variables in `GET(v1,v2)` is untrusted. Going forward, that untrusted information propagates to `v4 = extract_value(v3)`, `v6 = CONTRACT(v4)`, and `v7 = extract_value(v6)`. From there, it flows into `TRANSFER_TOKENS(v10,v9,v7)`, where the analysis issues a warning since `v7` is untrusted. However, that warning is a false positive. Namely, the input determines the choice of the contract, but the choice is made over a read-only map of harcoded addresses and the cross-contract invocation leads to a known contract, always. In general, the precision of an analysis depends on its abstraction level, which is often inversely related to its performance. In future works, more precise abstractions will be considered.

VI. Conclusion and Related Work

This paper presents MichelsonLiSA, an abstract interpretation-based static analyzer for Tezos smart contracts.

```
parameter int ;
storage unit ;
code {
  CAR ;
  PUSH (map int address)
       {
    Elt 0 "tz1KqT...
           b7QbPE" ;
    Elt 1 "tz2VGB...
           S6rna5"} ;
  SWAP;
  GET ;
  IF_NONE { PUSH string
       "key not found" ;
        FAILWITH }
  {
    CONTRACT unit ;
    IF_NONE { PUSH
         string "invalid
          contract" ;
         FAILWITH }{};
    AMOUNT ;
    UNIT ;
    TRANSFER_TOKENS ;
    NIL operation ;
    SWAP;
    CONS ;
    UNIT ;
    SWAP;
    PAIR }
}
```

(a) Michelson smart contract

```
v0 = parameter_storage();
v1 = CAR(v0);
v2 = PUSH( map { 0 : "tz1KqT...
     b7QbPE", 1 : "tz2VGB...S6rna5
     " });
v3 = GET(v1,v2);
IF v4 = extract_value(v3) is None
     {
  v5 = PUSH ("key not found");
  FAILWITH();
}
v6 = CONTRACT(v4)
IF v7 = extract_value(v6) is None
     {
  v8 = PUSH ("invalid contract");
}
v9 = AMOUNT();
v10 = UNIT();
v11 = TRANSFER_TOKENS(v10,v9,v7);
v12 = NIL();
SWAP();
v13= CONS(v11,v12);
v14= UNIT();
SWAP();
v15=PAIR(v13,v14);
```

(b) Michelson IR in SSA form

Fig. 8: A smart contract that allows one to transfer an amount of tokens to an address that can be selected by the input parameter among those contained in a hard-coded map.

It supports non-trivial analyses that proved to be applicable on real code. Experiments show that UCCIs happen frequently in real-world contracts, an our approach can successfully identify them. In future work, other analyses will be developed and the UCCI analysis will be improved wrt. precision and efficiency. Furthermore, we will also study UCCIs for blockchain frameworks written in other programming languages and supported by LiSA, such as for Go [13].

Related Work. Other several tools exist for the verification of smart contracts, but only a few apply to Michelson. Bernardo et al. [14] define Mi-Cho-Coq, a Coq framework to verify the functional correctness of Michelson contracts. They also introduce an intermediate language called Albert, that provides a high-level stack abstraction based on linearly-typed records that can be exploited by Mi-Cho-Coq. Arrojado et al. [15] propose a Why3 prover for deductive verification of Michelson contracts. The use of Coq or Why3 relies on theorem proving, that requires formal specifications of the expected behavior of the code, such as pre- or post-conditions. Therefore, unlike MichelsonLiSA, their approach is not fully automatic. The same holds for Nishida et al. [16], who define a tool to typecheck Michelson smart contracts against a user-provided specification based on a type system, by using the Z3 solver. Reis et al. [17] describe an IR called Tezla that linearizes the stack into a store of variables and can be combined with SoftCheck for data-flow analyses. The approach is similar to ours, especially regarding IR forms, but we focus on an analyzer based on abstract interpretation instead. Bau et al. [18] present a static analyzer for Michelson based on MOPSA [19], an abstract interpretation framework. MOPSA is the major alternative to LiSA. It is designed to compute fixpoints by induction on a program's syntax and considers a program as an extensible AST that initially contains the original source code, but that can be syntactically and semantically rewritten during the analysis. Regarding IR forms, our IR is similar also to that of BC2BIR [7], that transforms Java bytecode into variable assignments, including exception flows, and is based on a symbolic stack execution. However, it is variable-based without being in SSA form. Indeed, it does not guarantee SSA of variables in linear code, it does not use ϕ-functions and variables are assigned several times before a junction point.

References

[1] P. Cousot, *Principles of Abstract Interpretation*. MIT Press, 2021.

[2] P. Ferrara, L. Negrini, V. Arceri, and A. Cortesi, "Static analysis for dummies: Experiencing lisa," in *10th International Workshop on the State Of the Art in Program Analysis*, 2021, p. 1–6.

[3] L. Negrini, "A generic framework for multilanguage analysis," PhD thesis, Ca' Foscari University of Venice, Italy, 2023.

[4] T. Parr, "ANTLR Website," https://www.antlr.org/ Accessed 07/2022.

[5] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 12–27.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, 1991.

[7] D. Demange, T. Jensen, and D. Pichardie, "A provably correct stackless intermediate representation for java bytecode," in *Asian Symposium on Programming Languages and Systems*. Springer, 2010, pp. 97–113.

[8] M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto, "Boolean Formulas for the Static Identification of Injection Attacks in java," in *20th International Conference Logic for Programming, Artificial Intelligence, and Reasoning*, vol. 9450. Springer, 2015, pp. 130–145.

[9] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, 2003.

[10] A. K. Mandal, P. Ferrara, Y. Khlyebnikov, A. Cortesi, and F. Spoto, "Cross-program taint analysis for IoT systems," in *35th Symposium on Applied Computing, online event, [Brno, Czech Republic]*. ACM, 2020, pp. 1944–1952.

[11] P. Ferrara, L. Olivieri, and F. Spoto, "Static Privacy Analysis by Flow Reconstruction of Tainted data," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 31, no. 7, pp. 973–1016, 2021.

[12] J. S. Reis, "Tezla test repository," https://github.com/joaosreis/tezla/tree/main/tests. Commit:baacf2a79f8ac1fee8b5200395ffc14d5b9922e6 Accessed 09/2022.

[13] L. Olivieri, F. Tagliaferro, V. Arceri, M. Ruaro, L. Negrini, A. Cortesi, P. Ferrara, F. Spoto, and E. Talin, "Ensuring determinism in blockchain software with golisa: an industrial experience report," in *11th International Workshop on the State Of the Art in Program Analysis*, 2022, pp. 23–29.

[14] B. Bernardo, R. Cauderlier, G. Claret, A. Jakobsson, B. Pesin, and J. Tesson, "Making tezos smart contracts more reliable with coq," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2020, pp. 60–72.

[15] L. P. Arrojado da Horta, J. Santos Reis, M. Pereira, and S. Melo de Sousa, "Whylson: Proving your michelson smart contracts in why3," *arXiv e-prints*, pp. arXiv–2005, 2020.

[16] Y. Nishida, H. Saito, R. Chen, A. Kawata, J. Furuse, K. Suenaga, and A. Igarashi, "Helmholtz: A Verifier for Tezos Smart Contracts Based on Refinement Types," *New Generation Computing*, vol. 40, no. 2, pp. 507–540, 2022.

[17] J. S. Reis, P. Crocker, and S. M. de Sousa, "Tezla, an intermediate representation for static analysis of michelson smart contracts," *arXiv preprint arXiv:2005.11839*, 2020.
[18] G. Bau, A. Miné, V. Botbol, and M. Bouaziz, "Abstract interpretation of michelson smart-contracts," in *11th International Workshop on the State Of the Art in Program Analysis*, 2022, pp. 36–43.
[19] A. Miné, A. Ouadjaout, and M. Journault, "Design of a modular platform for static analysis," in *9th Workshop on Tools for Automatic Program Analysis*, 2018.