Tezla, an Intermediate Representation for Static Analysis of Michelson Smart Contracts

João Santos Reis
Release Lab, Nova-Lincs, University of Beira Interior, Portugal
joao.reis@ubi.pt

Paul Crocker
Release Lab, C4, University of Beira Interior, Portugal
crocker@di.ubi.pt

Simão Melo de Sousa
Release Lab, C4, Nova-Lincs, University of Beira Interior, Portugal
desousa@di.ubi.pt

Abstract

This paper introduces Tezla, an intermediate representation of Michelson smart contracts that eases the design of static smart contract analysers. This intermediate representation uses a store and aims to preserve the semantics, flow and resource usage of the original smart contract. This enables properties like gas consumption to be statically verified. We provide an automated decompiler of Michelson smart contracts to Tezla. In order to support our claim about the adequacy of Tezla, we develop a static analyser that takes advantage of the Tezla representation of Michelson smart contracts to prove simple but non-trivial properties.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Intermediate representation, Static analysis, Tezos blockchain, Michelson

Digital Object Identifier 10.4230/OASIcs.FMBC.2020.4

Related Version `https://arxiv.org/abs/2005.11839`

Supplementary Material Source code of the implementation available at: `https://gitlab.com/releaselab/fresco/tezla`.

Funding This work was supported and funded by the Tezos Foundation by the project FRESCO (Formal Verification of Tezos Smart Contracts).

1 Introduction

The term “smart contract” was proposed by Nick Szabo as a way to formalize and secure relationships over public networks [26]. In a blockchain, a smart contract is an application written in some specific language that is embedded in a transaction (hence the program code is immutable once it is on the blockchain). Some examples of smart contracts applications are the management of agreements between parties without resorting to a third party (escrow) and to function as a multisignature account spending requirement. Smart contracts have the ability to transfer/receive funds to/from users or from other smart contracts and can interact with other smart contracts.

There are reports of bugs and consequently attacks in smart contracts that have led to losses of millions of dollars worth of assets. One of the most famous and most costly of these attacks was on the Distributed Autonomous Organization (DAO), on the Ethereum blockchain [8]. The attacker managed to withdraw approximately 3.6 million ethers from the contract.

2nd Workshop on Formal Methods for Blockchains (FMBC 2020).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 4; pp. 4:1–4:12
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Given the fact that a smart contract in a blockchain cannot be updated or patched, there is an increasing interest in providing tools and mechanisms that guarantee or potentiate the correctness of smart contracts and to verify certain properties. However, current tools and algorithms for program verification that are based, for example, on deductive verification and static analysis, are usually designed for classical store-based languages in contrast with Michelson [15], the smart contract language for the Tezos Blockchain [11], which is stack-based.

To facilitate the usage of such tools to verify Michelson smart contracts, we present Tezla, a store-based intermediate representation language for Michelson, and its respective tooling. We provide an automated translator of Michelson smart contracts to Tezla. The translator was designed and implemented in a way that aims to preserve the semantics, flow, and resource usage of the original smart contract, so that properties like gas consumption can be faithfully verified at the Tezla representation level. To support our work, we present a case study of a demo platform for the static analysis of Tezos smart contracts using the Tezla intermediate representation alongside with an example analysis.

The paper is structured as follows. In section 2, we introduce the Tezla intermediate representation and the translation mechanism of Michelson code to Tezla. Section 3 addresses the static analysis platform case study that targets Tezla-represented smart contracts. In section 4, we talk about the related work. Finally, section 5 concludes with a general overview of this contribution and future lines of work.

2 Tezla

Tezla aims to facilitate the adoption of existing static analysis tools and algorithms. As such, Tezla is an intermediate representation of Michelson code that uses a store instead of a stack, enforces the Static Single-Assignment Form (SSA) [20] and aims to preserve information about gas consumption. We will see in the next section how such characteristics ease the translation of a Tezla program into its Control Flow Graph (CFG) forms and the construction of data-flow equations.

Compiled languages (like Albert [5], LIGO [1], SmartPy [16], Lorentz [25], etc.) also provide a higher-level abstraction over Michelson. However, as it happens with most compiled languages, the produced code may not be as concise or compact as expected which, in the case of smart contracts, may result in a higher gas consumption and, consequentially, undesired costs. Tezla was designed to have a tight integration with the Michelson code to be executed, not as a language that compiles to Michelson neither as a higher level language to ease the writing of Michelson smart contracts.

Tezla adapts the Michelson syntax and semantics in order to transform the stack usage to a traditional store usage. As such, we encourage the reader to head to the Michelson documentation [14] for more information about the Michelson language and its syntax and semantics.

Due to its large extent, the full syntax and semantics of the Tezla representation are not presented here but can be found at `https://gitlab.com/releaselab/fresco/tezla-semantics`.

In a general way, Michelson push-like instructions are translated into variable assignments, whereas instructions that consume stack values are translated to expressions that use as arguments the variables that match the values from the stack. Furthermore, lists, sets and maps deconstruct and lifting of `option` and `or` types that happen implicitly are represented through explicit expressions added to Tezla.

Since the operational effect of stack manipulation is transposed into variable assignments, we also expose in a Tezla represented contract the stack manipulation as instructions that act as no-op instructions in the case of a semantics that do not take resource consumption into account[1]. In the case of a resource aware semantics, these instructions will semantically encode this consumption.

The following section describes in detail the process of transforming a Michelson smart contract to a Tezla representation.

2.1 Push-like instructions and stack values consumption

Instructions that push N values to the stack are translated to N variable assignments of those values. The translation process maintains a Michelson program stack that associates each stack position to the variable to which that position value was assigned to. When a stack element is consumed, the corresponding variable is used to represent the value. A very simple example is provided in listings 1 and 2.

Listing 1 Stack manipulation example – Michelson code.

```
PUSH nat 5;
PUSH nat 6;
ADD;
```

Listing 2 Stack manipulation example – Tezla code.

```
v1 := PUSH nat 5;
v2 := PUSH nat 6;
v3 := ADD v1 v2;
```

The block on listing 1 is translated to the Tezla representation shown in listing 2.

From the previous example, we can also observe that Michelson instructions that consume N stack variables are translated to an expression that consumes those N values. Concretely, the instruction `ADD` that consumes two values (say, `a` and `b`), from the stack is translated to `ADD a b`.

2.2 Branching and deconstructions

Michelson provides developers with branching structures that act on different conditions. As Tezla aims at being used as an intermediate representation for static analysis, there are some properties we would like to maintain. One such property is static single-assignment form (SSA-form) [20], so that we obtain data flow information in a way that simplifies analyses and code optimization. This is guaranteed as Tezla-represented smart contracts are, by construction, in SSA-form, since each assignment uses new variables.

In order to deal with branching, the Tezla representation makes use of ϕ-functions (see [20]) that select between two values depending on the branch. As an illustration consider the Michelson example in listing 3.

The `IF_CONS` instruction tests if the top of the stack is a non-empty list, and deconstructs the list in the true branch by putting the head and the tail of the list on top of the stack.

In this example, the `IF_CONS` instructions checks the top of the stack and if it is a non-empty list it inserts the sum of a `int` value already on the stack with the head of the list at the lists's head. If the list is empty, it inserts the value into the empty list. Here, each branch of the `IF_CONS` instruction will result in a stack with a list of integers, whose values depends on which branch was executed. This translates to the Tezla representation presented on listing 4.

[1] This is the case of the semantics presented in this paper.

Listing 3 Branching example – MICHELSON code.

```
IF_CONS
  { DUP ;
    DIP { CONS ; SWAP } ;
    ADD ; CONS }
  { NIL int ; SWAP ; CONS } ;
  DUP ;
  PAIR;
```

Listing 4 Branching example - TEZLA code.

```
IF_CONS v1
{
  v2 := hd v1;
  v3 := tl v1;
  v4 := DUP v2;
  v5 := CONS v2 v3;
  SWAP;
  v6 := ADD v4 v0;
  v7 := CONS v6 v5
}
{
  v8 := NIL int;
  SWAP;
  v9 := CONS v0 v8
};
v10 := ϕ(v7, v9);
v11 := DUP v10;
v12 := PAIR v11 v10;
```

The variable `v10` will receive its value through a ϕ-function that returns the value of `v7` if the true branch is executed, or the value of `v9` otherwise.

From this example, it is possible to observe that the deconstruction of a list is explicit through two variable assignments. This is also the behaviour of `IF_NONE` and `IF_LEFT` instructions, where the unlifting of `option` and `or` types happens explicitly through an assignment.

2.3 Loops, maps and iterations

MICHELSON also provides language constructs for looping and iteration over the elements of lists, sets and maps. Sets in MICHELSON are defined as ordered lists, whereas maps are defined as lists of key-value pairs ordered by key. These are treated using the same ϕ-functions mechanism in order to preserve SSA-form. We can observe this on the example listing 5.

This example uses a `LOOP_LEFT` (loop with an accumulator) to sum 1 to a `nat` (starting with the value 0) until that value becomes greater than 100 and casts the result to an `int`. This example translates to the code presented in listing 6.

Note that the `LOOP_LEFT` variable is assigned to the value of `v1` if it is the first time that the loop condition is checked, or `v12` if the program flow comes from the loop body. Moreover, notice that the same explicit deconstruction of an `or` (union type) variable is applied here, where `v3` gets assigned the value of the unlifting of the loop variable in the beginning of the loop body and `v13` at the end of the loop. Similar behaviour applies to the other looping and iteration instructions.

2.4 Parameter and Storage

We now present an example of a complete MICHELSON smart contract (listing 7).

The contract takes an `int` as parameter and adds 1 to that value, which is later put in the storage. This contract translates to the TEZLA code of figure 8.

Listing 5 Loop example – MICHELSON code.

```
PUSH nat 0 ;
LEFT nat ;
LOOP_LEFT
 { DUP ;
   PUSH nat 100 ;
   COMPARE ;
   GE ;
   IF
    { PUSH nat 1 ;
      ADD ; LEFT nat }
    { RIGHT nat } } ;
INT ;
```

Listing 6 Loop example – TEZLA code.

```
v0 := PUSH nat 0;
v1 := LEFT nat v0;
LOOP_LEFT v2 := ϕ(v1, v12)
{
  v3 := unlift_or v2;
  v4 := DUP v3;
  v5 := PUSH nat 100;
  v6 := COMPARE v5 v4;
  v7 := GE v6;
  IF v7
  {
    v8 := PUSH nat 1;
    v9 := ADD v8 v3;
    v10 := LEFT nat v9;
  }
  {
    v11 := RIGHT nat v3;
  }
  v12 := ϕ(v10, v11);
}
v13 := unlift_or v2;
v14 := INT v13;
```

Listing 7 Example contract – MICHELSON code.

```
parameter (int) ;
storage (int) ;
code { CAR ;
      PUSH int 1;
      ADD;
      NIL operation ;
      PAIR; }
```

Listing 8 Example contract – TEZLA code.

```
v0 := CAR parameter_storage;
v1 := PUSH int 1;
v2 := ADD v1 v0;
v3 := NIL operation;
v4 := PAIR v3 v2;
return v4;
```

In this example, we can observe that a MICHELSON contract has a parameter and storage. The initial stack of any MICHELSON smart contract is a stack that contains a single pair whose first element is the input parameter and second element is the contract storage. As such, we introduce a variable called `parameter_storage` that contains the value of that pair.

The final stack of any MICHELSON smart contract is also a stack that contains a single pair whose first element is a list of internal operations that it wants to emit and whose second element is the resulting storage of the smart contract. We identify the variable containing this pair through the addition of a `return` instruction.

3 Building static analyses for Tezla smart contracts

In this section, we present the experiments conducted in order to test and demonstrate the applicability of the TEZLA intermediate representation to perform static analysis.

3.1 SoftCheck

We build and organise these static analyses upon a generic data-flow analysis platform called SOFTCHECK [18]. SOFTCHECK provides an internal and intermediate program representation, called SCIL, rich enough to express high-level as well as low-level imperative programming constructs and simple enough to be adequately translated into CFGs.

SoftCheck is organised upon a generic monotone framework [12] that is able to extract a set of data-flow equations from (1) a suitable representation of programs and; (2) a set of monotone functions; and then to solve them. SoftCheck is written in OCaml and makes use of functor interfaces to leverage its genericity (see figure 1).

By generic we mean that, given a translation from a programming language to SCIL. SoftCheck gives the ability to instantiate its underlying monotone framework by means of a functor interface. Then all defined static analyses are automatically available for the given programming language.

On the other hand, once written as a set of properties that define the domain of the analysis and the monotone functions on that domain, a particular static analysis can be incorporated (again, through instantiating a functor) as an available static analysis for all interfaced programming languages.

SoftCheck offers several standard data-flow analysis such as very busy expressions, available expressions, tainted analysis etc.

We propose in the next sections to detail how we have interfaced Tezla with SCIL, how we have designed a simple but useful data-flow analysis within SoftCheck and how we have tested this analysis on the Michelson smart contracts running in the Tezos blockchain.

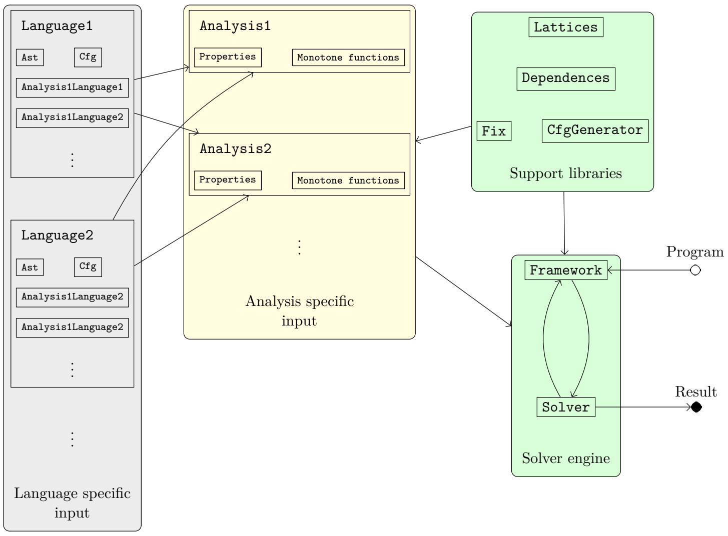

Figure 1 SoftCheck in a picture (adapted from [21]).

3.2 Constructing a Tezla Representation of a Contract

To obtain the Tezla representation of a smart contract, we first developed a parser to obtain an abstract syntax representation of a Michelson smart contract. This parser was implemented in OCaml and Menhir and respects the syntax described in the Tezos documentation [15]. It allows us to obtain a data type that fully abstracts the syntax (with the exception of annotations). The reason behind the implementation of our own parser was to obtain a data type that would better suit and ease the adoption of the integration with SoftCheck. Therefore, to improve the integration between these two forms, Tezla data types were built upon the data types of Michelson.

Control-flow graphs are a common representation among static analysis tools. We provide a library for automatic extraction of such representation from any Tezla-represented smart contract. This library is based upon the control-flow generation template present in

SoftCheck. As such, control-flow graphs generated with this library can be used with SoftCheck without further work. To instantiate the control-flow graph generation template, we simply provided the library with a module with functions that describe how control flows between each node.

3.3 Sign Detection: An Example Analysis

At this point, the SoftCheck platform is ready to be used to develop data flow analyses targeting Tezla represented smart contracts.

Here we devise an example of a static analysis for sign detection. The abstract domain consists of the following abstract sign values: 0 (zero), 1 (one), 0+ (zero or positive), 0- (zero or negative), + (positive), - (negative), ⊤ (don't know) and ⊥ (not a number). These values are organised according to the lattice on figure 2.

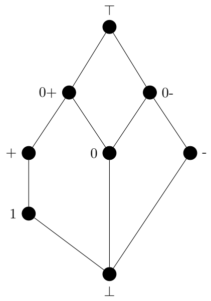

Figure 2 Sign lattice.

Using SoftCheck, we implemented a simple sign detection analysis of numerical values. By definition, `nats` have a lowest precision value of 0+, while `ints` can have any value. Every other data type has a sign value of ⊥.

This implementation does not propagate information to non-simple types (`pair`, `or`, etc.), but it does perform some precision refinements on branching.

To implement such an analysis, we provided SoftCheck, in addition to the previously defined Tezla control-flow graph library, a module that defines how each instruction impacts the sign value of a variable. Then, using the integrated solver mechanism based on the monotone framework, we are able to run this analysis on any Tezla represented smart contract.

We now present an example. Listings 9 and 10 show the code of a smart contract and its Tezla representation. This contract multiplies its parameter by -5 if the parameter is equal to 0, or by -2 otherwise, and stores the result in the storage. Figure 3 shows the control-flow graph of representation of that contract.

Running this analysis on the previously mentioned contract produced the results shown in Figure 4. In these results we can observe the known sign value of each variable at the exit of each block of the control-flow graph in Figure 3. For brevity, we omitted non-numerical variables from the result.

It it possible to observe from the results that the analysis takes into account several details. For instance, the sign of values of type `nat` are, by definition, always zero or positive. The analysis also refines the sign values on conditional branches according to the test. In this case, we can observe that in blocks 6 and 7 (true branch) the sign value of `v1` must be `0`, as the test corresponds to `0 == v1`. Complementary to this, in blocks 8 and 9 the value of `v1` assumes the sign value of `+`, since being a `nat` value its value must be `0+` and we know that its values is not zero because the test `0 == v1` failed.

Listing 9 Example contract for sign analysis – Michelson code.

```
parameter nat ;
storage int ;
code { CAR ;
       DUP ;
       PUSH nat 0 ;
       COMPARE ;
       EQ ;
       IF { PUSH int -5 ; MUL }
          { PUSH int -2 ; MUL } ;
       NIL operation ;
       PAIR }
```

Listing 10 Example contract for sign analysis – Tezla code.

```
v0 := CAR parameter_storage;
v1 := DUP v0;
v2 := PUSH nat 0;
v3 := COMPARE v2 v1;
v4 := EQ v3;
IF v4
{
    v5 := PUSH int -5;
    v6 := MUL v5 v0;
}
{
    v7 := PUSH int -2;
    v8 := MUL v7 v0;
};
v9 := phi(v6, v8);
v10 := NIL operation;
v11 := PAIR v10 v9;
return v11;
```

Due to the Tezla nature, we were able to take advantage of existing tooling, such as the SoftCheck platform, and effortlessly design the run a data-flow analysis. This enables and eases the development of static analysis that can be used to verify smart contracts but also to perform code optimisations, such as dead code elimination. Albeit simple, the sign analysis can be used to instrument such dead code elimination procedure.

3.4 Experimental Results and Benchmarking

Tezla and all the tooling are implemented in OCaml and are available at [13]. Tezla accepts Michelson contracts that are valid according to the Tezos protocol 006 Carthage. We conducted experimental evaluations that consisted in transforming to Tezla and running the developed analyses on a batch of smart contracts.

To do so, we implemented a tool that allows the extraction of smart contracts available in the Tezos blockchain. With that tool, we extracted 142 unique smart contracts. We tested these unique contracts alongside 21 smart contracts we have implemented ourselves.

We successfully converted all smart contracts with a coverage result of all Michelson instructions except for 9 instructions that were not used in any of these 163 contracts. On those, we ran the available analyses and obtained the benchmarks presented on table 1. These experiments were performed on a machine with an Intel i7–8750H (2.2 GHz) processor with 6 cores and 32 GB of RAM.

In the absence of an optimisation tool that takes advantage of the information computed by the analysis, we do not produce any optimisations from the analyses results. To do so, currently one must manually inspect the reports produced by the analysis. These reports, the source code of contracts under evaluation, as well as the respective analysis results and other performed static analyses are available at [22, 19].

4 Related Work

Albert [5] is an intermediate language for the development of Michelson smart contracts. This language provides an high-level abstraction of the stack and some of the language datatypes. This language can be compiled to Michelson through a compiler written in Coq that targets Mi-Cho-Coq [4], a Coq specification of the Michelson language.

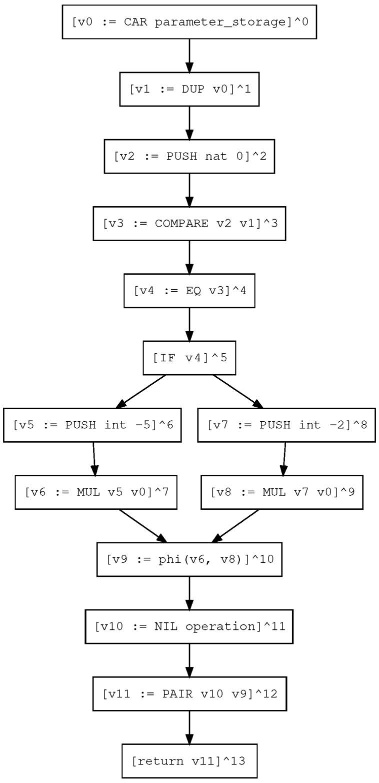

Figure 3 Generated CFG, by the SoftCheck tool.

Table 1 Benchmark results.

Average time	0.48 s	Worst-case (number of instructions)	2231 (6.08 s)
Worst-case (time)	9.87 s (926 instructions)	Average time per instrucion	0.0009

Several high-level languages [1, 2, 16, 7, 25] that target Michelson have been developed. Each one presents a different mechanism that abstracts the low-level stack usage. However, a program analysis tool that would target one of these languages should not be easily reusable to programs written in the other languages.

Scilla [23, 24] is an intermediate language that aims to be a translation target of high-level languages for smart contract development. It introduces a communicating automata-based computational model that separates the communication and programming aspects of a contract. The purpose of this language is to serve as a basis representation for program analysis and verification of smart contracts. We believe that Tezla is at a different level than Scilla, as we could use a Tezla representation to be mid step between having a Scilla representation and the Michelson code.

Slither [10], presented in 2019, is a static analysis framework for Ethereum smart contract. It uses the Solidity smart contract compiler generated Abstract Syntax Tree to transform the contract into an intermediate representation called SlithIR. This representation also uses a

```
0: {
    v0: 0+
}
1: {
    v0: 0+,
    v1: 0+
}
2: {
    v0: 0+,
    v1: 0+,
    v2: 0
}
3: {
    v0: 0+,
    v1: 0+,
    v2: 0,
    v3: 0-
}
4: {
    v0: 0+,
    v1: 0+,
    v2: 0,
    v3: 0-
}
5: {
    v0: 0+,
    v1: 0+,
    v2: 0,
    v3: 0-
}
6: {
    v0: 0,
    v1: 0,
    v2: 0,
    v3: 0-,
    v5: -
}
7: {
    v0: 0,
    v1: 0,
    v2: 0,
    v3: 0-,
    v5: -,
    v6: 0
}
8: {
    v0: +,
    v1: +,
    v2: 0,
    v3: 0-,
    v7: -
}
9: {
    v0: +,
    v1: +,
    v2: 0,
    v3: 0-,
    v7: -,
    v8: -
}
10: {
    v0: 0+,
    v1: 0+,
    v2: 0,
    v3: 0-,
    v5: -,
    v6: 0,
    v7: -,
    v8: -,
    v9: 0-
}
11: {
    v0: 0+,
    v1: 0+,
    v2: 0,
    v3: 0-,
    v5: -,
    v6: 0,
    v7: -,
    v8: -,
    v9: 0-
}
12: {
    v0: 0+,
    v1: 0+,
    v2: 0,
    v3: 0-,
    v5: -,
    v6: 0,
    v7: -,
    v8: -,
    v9: 0-
}
13: {
    v0: 0+,
    v1: 0+,
    v2: 0,
    v3: 0-,
    v5: -,
    v6: 0,
    v7: -,
    v8: -,
    v9: 0-
}
```

Figure 4 Generated report for the sign analysis.

SSA form and a reduced instruction set to facilitate the implementation of program analyses of smart contracts. However, the Slither intermediate representation is not able to accurately model some low-level information like gas computations, which we took into account when designing TEZLA. Also, this work does not contemplate a formal semantics of SlithIR.

Solidifier [3] is a bounded model checker for Ethereum smart contracts that converts the original source code to Solid, a formalisation of Solidity that runs on its own execution environment. Solid is translated to Boogie, an intermediate verification language that is used by the bounded model checker Corral, which is then used to look for semantic property violations.

Durieux et. al [9] presented a review on static analysis tools for Ethereum smart contracts. This work presents an extensive list of 35 tools, of which 9 respected their inclusion criteria: the tool is publicly available and supports a command-line interface; takes as input a Solidity contract; requires nothing but the source code of the contract; the tool claims to be able to identify vulnerabilities and bad practices in the contract. The authors then used those tools to test several vulnerabilities on a sample set of 47,587 smart contracts. This work presents some interesting results, as it was able to detect 97% of the smart contracts as vulnerable, as well as identify two categories of DASP10 as not able to be detect by the tools.

5 Conclusion

To the best of our knowledge, this is the first work towards a static analysis framework for Tezos smart contracts. TEZLA positions itself as an intermediate representation obtained from a Michelson smart contract, the low-level language of Tezos smart contracts. This representation abstracts the stack usage through the usage of a store, easing the adoption of mechanisms and frameworks for program analysis that assume this characteristic, while maintaining the original semantics of the smart contract.

We have presented a case study on how this intermediate representation can be used to implement a static analysis by using Tezla along side the SoftCheck platform. This has shown how effortlessly one can perform static analysis on Michelson code without forcing developers to use a different language or implement `ad hoc` static analysis tooling for a stack-based language.

Michelson smart contracts have a mechanism of contract level polymorphism called entrypoints, where a contract can be called with an entrypoint name and an argument. This mechanism takes the form of a parameter composed as nesting of `or` types with entrypoint name annotations. This parameter is then checked at the top of the contract in a nesting of `IF_LEFT` instructions, running the desired entry point this way. This mechanism is optional and transparent to smart contracts without entry points. As such, they are also transparent to Tezla. We therefore plan to extend Tezla to deal with entrypoints and generate isolated components for each entrypoint of a smart contract, which allow us to obtain clearer control flow graphs and analysis results. This allows us to analyse each entry point separately and possibly obtain more fine-grained results.

5.1 Future Work

At the moment of this paper writing, there is an initial work on an static analysis of Tezla represented smart contracts to detect potentially costly loops.

Future plans include a proof of correctness of the Michelson to Tezla transformation through a proof of equivalence of the Tezla semantics in respect to Michelson semantics. We aim to do so by developing a Tezla semantics using the Why3 deductive program verification platform and using the work done in WhylSon [6] to prove the semantic equivalence of Michelson and Tezla. Furthermore, this semantics should be accountable of gas consumption, so that we can provide a sound Tezla resource analysis in respect to the original Michelson code. This will also make way to the development of a platform for principled static analysis of Michelson smart contracts.

We plan to study which problems and properties are of interest so that we can integrate existing tools and algorithms for code optimization, resource usage and security analysis and correctness verification.

Another direction to tackle is the interfacing of Tezla with other static analysis platforms such as those provided by the MOPSA project [17] which, among other capabilities, provides a means to integrate static analyses. The integration with different static analysis platforms makes way to a more diverse universe of possible static analysis. Furthermore, it reinforces the statement that Tezla is an intermediate representation suitable not only for SoftCheck but for other platforms.

References

1 Gabriel Alfour. LIGO. URL: https://ligolang.org/.

2 Stephen Andrews and Richard Ayotte. Fi - Smart coding for smart contracts. URL: https://fi-code.com/.

3 Pedro Antonino and A. W. Roscoe. Formalising and verifying smart contracts with Solidifier: A bounded model checker for Solidity. *arXiv:2002.02710 [cs]*, February 2020. arXiv:2002.02710.

4 Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts. In *1st Workshop on Formal Methods for Blockchains*, September 2019. arXiv:1909.08671.

5 Bruno Bernardo, Raphaël Cauderlier, Basile Pesin, and Julien Tesson. Albert, an intermediate smart-contract language for the Tezos blockchain. In *4th Workshop on Trusted Smart Contracts*, January 2020. arXiv:2001.02630.

6 Luís Pedro Arrojado da Horta, João Santos Reis, Mário Pereira, and Simão Melo de Sousa. WhylSon: Proving your Michelson Smart Contracts in Why3. *arXiv:2005.14650 [cs]*, May 2020. arXiv:2005.14650.

7 DaiLambda. SCaml. URL: https://gitlab.com/dailambda/scaml.

8 Michael del Castillo. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft, June 2016. URL: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft.

9 Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *42nd International Conference on Software Engineering (ICSE '20)*, February 2020. doi:10.1145/3377811.3380364.

10 Josselin Feist, Gustavo Greico, 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 '19, pages 8–15, Montreal, Quebec, Canada, May 2019. IEEE Press. doi:10.1109/wetseb.2019.00008.

11 L M Goodman. Tezos-a self-amending crypto-ledger White paper. Technical report, Tezos, 2014.

12 John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977. doi:10.1007/BF00290339.

13 RELEASE Lab. FRESCO - formal verification and static analysis of tezos compliant smart contracts, gitlab. URL: https://gitlab.com/releaselab/fresco.

14 Nomadic Labs. Michelson Reference. URL: https://michelson.nomadic-labs.com.

15 Nomadic Labs. Michelson: The language of Smart Contracts in Tezos. URL: http://tezos.gitlab.io/whitedoc/michelson.html.

16 Francois Maurel and Smart Chain Arena. SmartPy. URL: https://smartpy.io/.

17 A. Miné, A. Ouadjaout, and M. Journault. Design of a modular platform for static analysis. In *Proc. of 9h Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, Lecture Notes in Computer Science (LNCS), page 4, August 2018. URL: http://www-apr.lip6.fr/~mine/publi/mine-al-tapas18.pdf.

18 João Reis. Softcheck, a generic and modular data-flow analyser. URL: https://gitlab.com/joaosreis/softcheck.

19 João Reis. TezCheck Analysis Results - GitLab. URL: https://gitlab.com/releaselab/fresco/tezcheck-results.

20 B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '88*, pages 12–27, San Diego, California, United States, 1988. ACM Press. doi:10.1145/73560.73562.

21 João Santos Reis. *SoftCheck, uma plataforma de construção de análises estáticas para a segurança de programas, genérica e composicional.* PhD thesis, University of Beira Interior, Covilhã, Portugal, July 2018.

22 João Santos Reis. TezCheck - softcheck interface for tezos, gitlab. URL: https://gitlab.com/releaselab/fresco/tezcheck.

23 Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: A Smart Contract Intermediate-Level LAnguage. *arXiv:1801.00687 [cs]*, January 2018. arXiv:1801.00687.

24 Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with Scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, October 2019 . doi:10.1145/3360611.

25 Serokell and TQ Group. Lorentz. URL: https://gitlab.com/morley-framework/morley/-/tree/master/code/lorentz.

26 Nick Szabo. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9), September 1997. doi:10.5210/fm.v2i9.548.