Abstract interpretation of Michelson smart-contracts

Guillaume Bau, Antoine Miné, Vincent Botbol, Mehdi Bouaziz

Abstract Interpretation of Michelson Smart-Contracts

Guillaume Bau
Sorbonne Université, CNRS, LIP6, Nomadic Labs
Paris, France
guillaume.bau@nomadic-labs.com

Antoine Miné
Sorbonne Université, CNRS, LIP6
Paris, France
antoine.mine@lip6.fr

Vincent Botbol
Nomadic Labs
Paris, France
vincent.botbol@nomadic-labs.com

Mehdi Bouaziz
Nomadic Labs
Paris, France
mehdi.bouaziz@nomadic-labs.com

Abstract

Static analysis of smart-contracts is becoming more widespread on blockchain platforms. Analyzers rely on techniques like symbolic execution or model checking, but few of them can provide strong soundness properties and guarantee the analysis termination at the same time. As smart-contracts often manipulate economic assets, proving numerical properties beyond the absence of runtime errors is also desirable. Smart-contract execution models differ considerably from mainstream programming languages and vary from one blockchain to another, making state-of-the-art analyses hard to adapt. For instance, smart-contract calls may modify a persistent storage impacting subsequent calls. This makes it difficult for tools to infer invariants required to formally ensure the absence of exploitable vulnerabilities.

The *Michelson* smart-contract language, used in the Tezos blockchain, is strongly typed, stack-based, and has a strict execution model leaving few opportunities for implicit runtime errors. We present a work in progress static analyzer for *Michelson* based on Abstract Interpretation and implemented within MOPSA, a modular static analyzer. Our tool supports the *Michelson* semantic features, including inner calls to external contracts. It can prove the absence of runtime errors and infer invariants on the persistent storage over an unbounded number of calls. It is also being extended to prove high-level numerical and security properties.

CCS Concepts: • **Security and privacy** → *Logic and verification*; • **Software and its engineering** → **Automated static analysis**.

Keywords: static analysis, abstract interpretation, smart-contract, blockchain, Michelson, Tezos

1 Introduction

1.1 Blockchains and Smart-Contracts

Blockchains are distributed immutable ledgers organized in peer to peer networks, allowing participants to securely transfer tokens without a central authority. Some blockchains allow programmable transactions in the form of computer programs. These *smart-contracts* define complex transactions between blockchain participants and maintain a persistent state across runs. They can be viewed as a novel way for multiple users to securely exchange, build value sharing or distribution applications without a trusted third-party. Applications include auction sales, decentralized exchanges, collective organizations, investment funds, etc.

Smart-contracts are relatively small, and not resource intensive as they have to be executed on all blockchain network nodes. However, compared to usual programming languages, they have an unconventional execution model tightly tied to the blockchain implementation, thus are non-intuitive to program. Compounded with the inability to update smart-contracts on the (immutable) blockchain, and applications manipulating large sums of money, this leads to costly errors. Notable vulnerability examples include: a reentrancy issue in *The DAO* [10], a smart-contract implementing a venture capital fund, that allowed a user to steal $60 million; the *Parity* wallet bug [29], which froze $150 million by allowing unauthenticated users to call restricted functions; and the Proof-of-Weak-Hands attack, that allowed attackers to steal $800,000 overnight [5], and $2.3 million then after, abusing an integer overflow. Thereby, there is a high motivation to statically detect potential misbehavior or prove their absence when possible.

1.2 Motivating Example

We focus on the verification by static analysis of smart-contracts for the Tezos blockchain programmed in the Michelson [34, 35] language. Our goal is to analyze *Dexter* [7], an important smart-contract implementing a decentralized exchange with alternate blockchain currencies (*bitcoins*, *usdtz*). Its initial version featured a vulnerability [20] allowing an attacker to steal parts of the contract funds. As this is a work in progress, we report on a preliminary analysis of a simplified version only.

Consider the contract in Fig. 1 inspired from Dexter and written in an ML-style pseudo-code. It implements a simple wallet, allowing users to deposit to or withdraw from their personal account some amount in mutez (the currency on

```
1  storage : (address, mutez) map
2  entry deposit () {
3    let owned = match Map.get $storage $sender with
4      | None -> 0
5      | Some v -> v in
6    (Map.add $sender (owned + $amount) $storage, [])
7  }
8  entry withdraw (asked : mutez, dest : address) {
9    assert ($amount == 0);
10 // Fix to ensure proper access control:
11 // if dest != $sender then failwith "unauthorized";
12   let owned = match Map.get $storage dest with
13     | None -> failwith "empty account"
14     | Some v -> v in
15   if asked > owned then
16     failwith "not enough tokens";
17   (Map.add dest (owned - asked) $storage,
18    [transfer $sender asked])
19 }
```

Figure 1. A smart-contract with incorrect authentication

the Tezos blockchain). A *map* keeps track of user accounts: it maps users, identified by their blockchain *address*, to the deposited amount. The map is kept on the blockchain, in a so-called *storage*, updated after each transaction. An execution of the smart-contract starts with the following variables:

- $storage is the storage value currently on the blockchain.
- $sender is the address of the initiator of the contract call (a user, or another smart-contract).
- $amount is the amount of mutez transferred to the contract. Every call is a transfer, possibly with a 0 amount.

A contract can define several independently callable entry points. They allow splitting functionalities sharing the same storage. In our case:

- *deposit* allows a user to deposit an amount. The $amount sent to the contract is actually recorded to belong to the user by updating his balance in the map (Map.add, line 6).
- *withdraw* allows a user to transfer back some amount from the contract, unless his account in the map does not hold sufficient funds (if asked > owned, line 15). The map is updated (Map.add, line 17) and a transfer back to the sender is generated (transfer, line 18).

Michelson features a purely functional execution model: the new value of the storage as well as any effect (e.g., additional transfers) are provided in the return value of the call.

This example actually contains a logic error: it allows a user to transfer to himself mutez that were owned by someone else. Indeed, the *dest* parameter used as key in the map in withdraw is controlled by the user who calls the contract. A fix is provided in a comment line 11: it ensures that dest equals the caller $sender. In this example, we want to verify the high-level property stating that:

$$(key = \$sender) \vee (new\_value \geq old\_value)$$

whenever the map is updated on key from old_value to new_value, i.e., a user can only add funds to another user's account and can subtract funds only from his own account.

1.3 Related Work

A number of tools have been designed to help smart-contract developers catch bugs or vulnerabilities. Most of them target the *Ethereum* platform. This includes symbolic execution tools, like Maian [28], Manticore [26], Oyente [22], Zeus [18], Securify [36], Mythril [27]. Some tools rely on exhaustive state exploration, via model checking or SMT solving, sometimes leading to a slow analysis [1], timeouts [31], or lack of results [1]. [18] relies on Abstract Interpretation for a preliminary analysis, and uses an SMT solver to check properties on inferred invariants. [11, 12, 14, 37] report that many existing tools fail to detect some issues, i.e., report false negatives. [14, 33, 37] affirm that some tools can fail to prove properties because their analyses are unsound.

Some tools focus on low-level properties affecting the popular Ethereum platform, like reentrancy issues [21, 32] or overflows [13]. By contrast, Michelson [34, 35], the language of the Tezos blockchain, which is the focus of our work, has fewer opportunities for runtime errors and a stricter execution model eliminating reentrancy issues; low-level properties are less of an interest. Checking higher-level properties is feasible using proof assistants, but requires a large effort to prove even simple properties. Mi-cho-Coq [4] provides a Michelson Coq embedding allowing the certification of smart-contract properties, but requires manual developments of proofs, and small changes in a contract require new proof developments. The Micse project [6] allows for automated static analysis, using the Z3 SMT solver. The Tezla [30] project allows translating the Michelson instructions into a suitable intermediate representation for dataflow analysis.

1.4 The MOPSA Static Analyzer

MOPSA [17] is a modular and extensible static analyzer based on Abstract Interpretation [8]. It features a C analyzer detecting runtime errors and invalid preconditions when calling the C library, as well as Python type, value, and uncaught exception analyses. Its modular design allows sharing and reusing abstract domains across multiple analyses. Its *AST* structure can be extended to support novel languages, keeping a high-level representation without static translation.

MOPSA strongly relies on domain cooperation. An analysis is defined as a combination of small domain modules, including value abstractions and syntax iterators that can be plugged in or out, depending on the target language and properties. It provides a common set of domains to build value analyses with intervals, relational domains like octagons [23] or polyhedra [9] to infer linear relations, recency abstraction [2] for memory blocks, etc. In addition to reductions, domains cooperate through expression rewriting. For instance, a domain handles C arrays by rewriting array accesses dynamically as accesses into scalar variables

```
1  storage  nat;
2  parameter  nat;
3  code {  UNPAIR;
4        ADD;
5        NIL  operation;
6        PAIR; }
```

Figure 2. Simple Michelson smart-contract

representing array cells. Expression rewriting helps writing small, independent, and reusable domains, that rely only on the manipulation of variables the state of which is managed by other, lower-level domains.

1.5 Contribution

We have developed an analysis of Michelson programs [34, 35] based on Abstract Interpretation. This analysis is built on MOPSA [17]. It reuses domains provided by MOPSA and provides novel domains to support the semantics of the Michelson language. This includes support for specific Michelson, ML-like types, such as pairs, unions, sets, maps, etc. as well as iterators to handle the execution model for contracts on the Tezos platform, including contract interactions. Our tool can currently statically detect runtime errors like *overflows*, *shift overflows*, and Michelson contracts always terminating in a failure state. We also demonstrate its potential to prove higher-level correctness properties on the example contract from Fig. 1. By using abstract interpretation, our analysis is sound and efficient, but can raise false alarms.

Section 2 presents a basic set of abstract domains that are sufficient to cover the complete semantic of Michelson instructions and achieve an initial, sound, low-precision analysis; Sect. 3 presents the support for the Tezos transaction execution model, including contracts calling external contracts and inferring invariants on unbounded sequences of calls to contracts; Sect. 4 presents more involved abstractions necessary to prove the correctness of Fig. 1; Sect. 5 presents our experimental evaluation; Sect. 6 concludes.

2 Michelson Value Analysis

For convenience, Fig. 1 presented a smart-contract example in a high-level, ML-like syntax. Michelson [34], the language actually executed on the Tezos blockchain, is a high-level *stack-based* language that takes inspiration from Forth [25] or Joy [19], while including many aspects from functional languages: strong static typing, immutable values, anonymous functions, algebraic data-types, functional *list*, *set*, and *map* types. This section presents the abstract domains added to MOPSA to handle the stack, data-types, and instructions.

2.1 The Michelson Language

To simplify, we consider here a much simpler contract, in Fig. 2, that performs an addition into an accumulator stored on the blockchain.

In Michelson, there are no explicit variables. All values are stored on a stack, implicitly manipulated through dedicated instructions such as PUSH, DROP, DUP, and using operator instructions (e.g., ADD to perform an addition) replacing arguments at the top of the stack with the operator result.

When the contract execution starts, the stack contains a single element: a pair containing the value of the parameter it has been called with and the value stored on the blockchain for the contract. When the execution ends, the contract should leave on the stack a single value: a pair containing a list of operations to perform after the contract execution (such as calling other contracts, or performing a transfer) and the new value to be stored on the blockchain. Operations will be discussed in details in Sect. 3. For now, the operation list output by a contract will be empty. The initial value of the storage is specified when the contract is deployed on the blockchain. Subsequent executions of the contract update the storage value.

As the language is statically typed, a contract declares the type of its storage and parameter. This corresponds to lines 1–2 in Fig. 2. In the example, both the storage and parameter have type nat (i.e., a non-negative integer), but more complex data structures can be used. For instance, Fig. 1 uses a map as storage and its parameter is a union to model different possible entry points.

When executed, the code from Fig. 2 proceeds as follows:
- UNPAIR pops the topmost (and only) element from the stack: a pair with the storage and the parameter, which are pushed as the first and second items on the stack;
- ADD, pops two elements, adds them, and pushes the result;
- NIL operation builds an empty list of operations, and pushes it on the stack;
- PAIR pops the addition result and the empty list, and pushes a pair, resulting in a stack with a single pair element.

Thus, each call to the contract will simply add the integer passed as parameter to the integer stored on the blockchain.

2.2 Dynamic Translation into Variables

MOPSA models the memory as a map from variables to values and supports instructions, such as assignments and tests, involving expressions over variables. This is a common assumption for abstract interpreters as well as domain libraries (such as APRON [16], used in MOPSA) and especially useful for relational analyses. One possibility to handle stack-based languages is to translate them to variable-based environments and expressions beforehand, in a pre-processing phase, as performed for instance in the Sawja framework [15] for Java bytecode as well as Tezla [30] for Michelson.

Instead of a static translation, we extended MOPSA's AST with a native support for Michelson instructions and relied on the ability for domains to rewrite statements and expressions as part of the abstract execution. We developed a domain that introduces variables to represent stack positions

and translates Michelson instructions into assignments on-demand. Non-scalar data-types, such as pairs, give rise to several variables per stack position, as detailed in Sect. 2.3. A dynamic translation can potentially use information about the current precondition to optimize the translated instructions [17], although this is not currently the case for Michelson.

2.3 Michelson Data-Types

Michelson supports several integer kinds: arbitrary precision integers (`int`), natural integers (`nat`), dates, and unsigned 63-bit integers (`mutez`). Some operations, such as overflows on `mutez`, as well as shift overflows, are checked runtime errors that halt the contract execution. A specific domain in MOPSA handles these types, checking all possible runtime errors and representing their possible values in standard numeric domains such as intervals and polyhedra.

Michelson supports simple algebraic types *à la ML* through pairs (`(a,b)`), option types (`Some a` or `None`), and tagged unions with two variants (`Left a` or `Right b`). The type of `a` and `b` is arbitrary, and algebraic types can be nested. MOPSA features domains to handle these types. They create and manage additional variables for each component of a pair, an option, or a sum, delegate the abstraction of their value to the domain of the components' type, and translate operations on algebraic types (such as `PAIR`, `CAR`, etc.) into operations on component variables. Domains handling scalar values, such as numeric domains, ultimately work on environments mixing components from different algebraic values, making it possible to infer relations between values that appear inside pairs or options. This technique is similar to that of Bautista et al. [3], but we support recursive types and do not partition with respect to which variant is used by each variable.

Michelson has a native support for immutable containers: lists, sets, and maps. We propose simple, general-purpose, and efficient, but coarse abstractions to handle them. Lists are abstracted using a summary variable to represent the union of all list elements, and a numeric variable representing its size. Like algebraic types, list elements can have arbitrary type. List operations are translated into operations on the variables (e.g., weak updates of summary variables, size incrementation) and delegated to the domain appropriate for the type. List iteration `ITER` is handled, as usual in abstract interpretation, using a fixpoint. Sets are abstracted similarly to lists with slight adjustments as they cannot contain duplicate elements. Maps are abstracted using a summary variable to represent keys and a summary variable to represent values. A more involved, property-specific abstraction of maps will be discussed in Sect. 4.3.

2.4 Addresses

Michelson has a domain-specific type for addresses, representing participants on the blockchain: either users (identified by a public key) or smart-contracts (identified by a hash). Some addresses play a special role during contract execution, and can be accessed using dedicated instructions. As detailed in Sect. 3, a contract execution can be triggered by the execution of another contract. `SOURCE` represents the user at the origin of a chain of calls, while `SENDER` is the immediate caller of the contract. These variables play an important role in access control and thus the security of contracts, as demonstrated by the fix proposed line 11 of Fig. 1 for our incorrect wallet implementation.

We use a reduced product of two domains for addresses: a powerset of address constants – useful to precisely handle addresses hard-coded in a contract – and a domain that maintains whether the address equals $sender or not. It is useful to handle precisely access control by comparison with $sender, which is not a literal constant.

3 Execution Model and Analysis

The previous section presented domains sufficient to handle the execution of arbitrary Michelson code on an input stack. In this section, we take into account the execution in its context on the blockchain. A contract can be executed multiple times, making its storage evolve during time. Additionally, one execution can trigger additional contract executions through the operation list it returns.

3.1 Execution Context

Once deployed (originated) on the blockchain, smart-contracts are available for any user to call. A call must provide a parameter as well as an entry point for the contract. As different entry points execute very different code, MOPSA performs a case analysis: for each entry point, the contract is analyzed on an initial stack for this entry point with a corresponding abstract parameter value modeling any possible actual value in the parameter type; the results are then joined after execution. The execution context also sets up special variables, such as $sender, modeling the contract caller and initialized with a symbolic value in the address domain (Sect. 2.4).

An analysis of the contract on an initial, empty storage would only model the very first execution of the contract, which is not sound. For instance, in Fig. 1, an empty storage means that the *withdraw* entry point always fails. Alternatively, starting with an abstract storage representing all possible concrete values in its type could be imprecise. Section 3.3 will propose another solution where a sound abstraction of the storage is inferred through fixpoint computation.

3.2 Operation List

In addition to the updated storage, Michelson contracts can return a list of operations to execute after they finish. These operations can be some calls to smart-contracts, which entails executing these contracts with the updated storage. These

can, in turn, append new operations to the operation list. The list is traversed in depth-first order until there are no more operations to execute. Note that a contract cannot call another contract in the middle of its execution and expect a return value; moreover, the execution of the operation list is atomic: a runtime error at any point reverts all modifications to the storage of the contracts involved. This unusual execution model makes reentrancy bugs, such as the one plaguing as *The DAO* [10], less likely on Tezos.

MOPSA has partial support for this model. We do compute the operation list and iterate contract execution in a fixpoint, using updated storage and inferred entry points and arguments, as mandated. This includes the cases where a contract calls itself, or another contract. However, our coarse abstraction of lists using summary variables (Sect. 2.3) makes the analysis impractical when a contract calls more than one contract. It should be addressed in future work.

3.3 Multiple Calls Analysis

Analyzing a unique call to a smart-contract provides some insights on the possible runtime errors, but it does not take into account all possible executions over its whole lifetime. We developed an analysis to over-approximate an infinite number of calls to a smart-contract, from different callers and to multiple entry points.

Let *Addr* be the set of all addresses, *Entrypoints* the set of entry points for the contract and P_e the semantic function computing the new storage after executing entry point e of contract P. The next storage S_{i+1} of the contract as a function of its current storage S_i (assuming, for the simplicity, an empty list of operations) is:

$$S_{i+1} = \exists addr \in Addr, \exists e \in Entrypoints, call(P_e, addr, S_i)$$

Using the (classic) technique of iterations with widening, with \$sender being an abstract value of our address reduced product from 2.4, our analysis computes an abstraction of the fixpoint:

$$lfp_{S_0}\left(\lambda S : \bigsqcup_{\forall e \in Entrypoints} call(P_e, \$sender, S)\right)$$

which models arbitrary sequences of executions of the contract from the initial storage S_0. It thus outputs all possible runtime errors. It also returns an invariant over storage values, which could be inspected by the user for additional insight on the behavior of the smart-contract. On the example of Fig. 1, we discover that the *deposit* entry point will fill an initially empty map and allow some *user* to call the *withdraw* entry point without entering the *failure* state. This is not sufficient yet to prove our property of interest, that "only owners can decrease the amount of tokens in the map."

4 High-Level Domains

We now present additional abstract domains, bringing more precision necessary to analyze our motivating example.

```
1  // assuming a stack containing values  x :: y;
2  DIP { DUP } // x :: y :: y
3  DUP;        // x :: x :: y :: y
4  DUG 2;      // x :: y :: x :: y
5  COMPARE;    // pops 2 items, push -1, 0 or 1
6  EQ;         // boolean test if -1, 0 or 1 equals to 0
7  IF          // pops boolean and branches accordingly
8    { } // x = y
9    { } // x != y
```

Figure 3. Comparison in Michelson

4.1 Symbolic Expressions

In Michelson, there is no direct comparison operator. Consider the example in Fig. 3 that executes different branches when the topmost stack elements x and y are equal, and when they are different. The COMPARE polymorphic instruction pushes −1 (resp. 0, 1) on the stack when one operand is smaller than (resp. equal to, greater than) the other. Then, an integer operation such as EQ compares the result to 0 and pushes a boolean on the stack, which is consumed by IF. To be precise, an analysis must track this sequence of instructions. Using the domains presented in Sect. 2, our analysis is only able to infer that *true* or *false* is pushed on the stack and immediately consumed, inferring no information on the topmost stack values x and y inside the branches.

As an alternative to developing a complex relational domain, we implemented the symbolic constant abstract domain proposed in [24]. This domain assigns to each variable a value v from the set of symbolic expressions $\mathbb{E}$, or $\top$ to represent no information: $v \in \{e, \top\}, e \in \mathbb{E}$. The mapping is updated through assignments, building more complex expressions by substitution. This domain allows reconstructing dynamically high-level expressions from low-level stack-based evaluation, without requiring a static pre-processing phase as done for instance by [15] on Java bytecode. In our example, just before the IF instruction, the top of the stack contains the expression *eq(compare(x, y))*, which allows IF to apply flow-sensitive constraints on the *x* and *y* values.

4.2 Equality Domain

A stack-based execution model entails pushing copies of existing values from the stack (using DUP), to be consumed later by operators while leaving the original values intact for future use. This can be seen, for instance, in Fig. 3. In this context, maintaining information about variable equalities is critical for precision: it allows any information inferred on one copy to be propagated to other copies. The symbolic expression domain from the last section helps to a degree, as it allows substituting x with y after an assignment x := y, which is sufficient for the case in Fig. 3. However, this substitution mechanism is unidirectional and can fail when the symmetry or the transitivity of equality is required. Equalities can be tracked by numerical abstract domains, such as polyhedra, but this is limited to numeric values, while we require tracking the equality of values of complex types (such as maps, for the example from Fig. 1). To solve this problem,

Table 1. Experimental evaluation

Analysis	intv	poly	intv+exp	poly+exp
total contracts	2931	2833	1579	1549
mutez overflow	2824	1967	411	308
shift overflow	10	10	9	9
always fail	32	33	32	33
min. time	0.076s	0.15s	0.17s	0.16s
max. time	71.02s	568.25s	581.34s	590.86s
avg. time	3.31s	29.8s	26.43s	21.89s

we developed a simple domain able to infer variable equalities. It maintains a set of equivalence classes for variables that are known to be equal. It proved to be more reliable than symbolic expressions for the specific purpose of tracking equalities on non-numeric variables.

4.3 Symbolic Maps

On the example Fig. 1, we want to prove that only the owner of an account stored in the storage map can reduce its amount. This requires inferring a numerical property about the contents of a map. However, this is not a uniform property: the property on the value depends on whether the key associated to it equals the $sender address or not. Hence, the simple summarization abstraction of Sect. 2.3 is not expressive enough. We propose a map abstraction of the form: $\{sender \mapsto amount, \neg sender \mapsto namount\}$ that uses two variables per map: *amount* represents the value associated to the key equal to $sender for this call; *namount* summarizes all the values associated to other keys.

Like previous abstractions, *amount* and *namount* are variables, the values of which are abstracted in mutez domain. Using a relational domain, it is possible to even track relations between different versions and copies of the map from the storage. All map operations are translated into operations on these variables, depending on whether the key used to access the map equals $sender or not, which can be precisely tested using our symbolic address domain (Sect. 2.4).

In our example, when updating the value of *namount*, we check that the new value is greater than or equal to the previous one. This is always the case for *deposit* (even if the *key* address was specified in parameter), as transferred amounts are always non-negative. As for *withdraw*, updating the old value with the value (owned − asked) triggers an error for the original version without the fix. For the fixed version, we are able to prove that the property is correct because the value of *namount* is unchanged in *withdraw*.

5 Experimental Results

We performed two kinds of experiments. Firstly, we analyzed a large set of existing contracts for non-functional correctness (e.g., absence of overflows) to assess the practicality and scalability of our method. Secondly, we analyzed

more specifically the example from Fig. 1 for our functional specification: only the owner of an account can decrease its amount. We used a Xeon E5-2650 CPU with 128GB memory. Our prototype can be found at https://gitlab.com/baugr/mopsa-analyzer at commit tag soap22.

We selected the Carthagenet test network containing 2935 contracts with size ranging from 1 to 3604 lines, and analyzed them with arbitrary storage. The results, using different domain combinations, is presented in Table 1: *intv* uses the domains from Sect. 2 and the interval domain; *poly* adds the polyhedra domain; *intv+exp* and *poly+exp* add the domains from Sect. 4. The first line indicates the number of successful analyses (not all domains can support all contracts due to the prototype nature of our implementation). The lines *mutez overflow* and *shift overflow* indicate the number of runtime errors detected, whereas *always fail* is the number of contracts always terminating in a failing state. The last three lines indicate the minimal, maximal and average runtime per contract. We expect that a large number of *mutez overflows* are actually false positive as the analysis assumes that arbitrary 63-bit amounts can be stored and transferred but, in fact, the total number of mutez in circulation is far smaller.

Our prototype can check the functional correctness of our motivating example from Fig. 1 in 0.273s. As for other examples, it raises spurious overflows in mutez computations.

6 Conclusion

We have proposed a new sound and efficient static analysis based on Abstract Interpretation for the Michelson smart-contract language. Our prototype implemented in MOPSA is already able to analyze realistic smart-contracts for runtime errors, and higher-level functional properties for toy contracts using realistic authentication patterns.

Future work include strengthening our implementation to analyze more contracts, as well as our support of operation lists for inter-contract analysis. We will also focus on analyzing functional correctness properties, closing the gap between our simplified example and the actual Dexter implementation, and considering other smart-contracts and properties. This entails developing more expressive domains, e.g. extending our non-uniform map abstraction to arbitrary value types, and supporting more complex authentication patterns, such as cryptographic signatures. Finally, we plan to exploit the value analysis to perform a gas consumption (*i.e.* timing) analysis.

References

[1] Binod Aryal. 2021. *Comparison of Ethereum Smart Contract Vulnerability Detection Tools.* Master's thesis. University of Turku. https://urn.fi/URN:NBN:fi-fe2021110453759

[2] Gogul Balakrishnan and Thomas Reps. 2006. Recency-Abstraction for Heap-Allocated Storage. In *Static Analysis: 13th International Symposium, SAS 2006, August 29-31* (Seoul, Korea). Springer, Berlin, Heidelberg, 221–239. https://doi.org/10.1007/11823230_15

[3] Santiago Bautista, Thomas Jensen, and Benoît Montagu. 2020. Numeric Domains Meet Algebraic Data Types. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains* (Virtual, USA) *(NSAD 2020)*. Association for Computing Machinery, New York, NY, USA, 12–16. https://doi.org/10.1145/3427762.3430178

[4] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. 2019. Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts. In *Formal Methods. FM 2019 International Workshops*. Springer, Cham, 368–379. https://doi.org/10.1007/978-3-030-54994-7_28

[5] Bitburner. 2018. *Proof of Weak Hands (PoWH) Coin hacked, 866 eth stolen.* Retrieved 2022-03-19 from https://steemit.com/cryptocurrency/@bitburner/proof-of-weak-hands-powh-coin-hacked-866-eth-stolen

[6] Jisuk Byun and Heewoong Jang. 2020. *MicSE: The Michelson Symbolic vErifier.* Retrieved 2022-03-21 from https://github.com/kupl/MicSE

[7] CamlCase. 2020. *Dexter: A decentralized exchange for Tezos, on Gitlab.* Retrieved 2022-03-19 from https://gitlab.com/camlcase-dev/dexter/

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* 238–252. https://doi.org/10.1145/512950.512973

[9] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. Association for Computing Machinery, New York, NY, USA, 84–96. https://doi.org/10.1145/512760.512770

[10] Michael del Castillo. 2016. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. *Saatavissa (viitattu 13.2. 2017)* 3 (2016). https://www.coindesk.com/markets/2016/06/17/the-dao-attacked-code-issue-leads-to-60-million-ether-theft/

[11] Bruno Dia, Naghmeh Ivaki, and Nuno Laranjeiro. 2021. An Empirical Evaluation of the Effectiveness of Smart Contract Verification Tools. In *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*. 17–26. https://doi.org/10.1109/PRDC53464.2021.00013

[12] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering.* 530–541. https://doi.org/10.1145/3377811.3380364

[13] Jianbo Gao, Han Liu, Chao Liu, Qingshan Li, Zhi Guan, and Zhong Chen. 2019. EASYFLOW: Keep Ethereum Away from Overflow. *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (May 2019). https://doi.org/10.1109/ICSE-Companion.2019.00029

[14] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 415–427. https://doi.org/10.1145/3395363.3397385

[15] Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. 2011. Sawja: Static Analysis Workshop for Java. In *Formal Verification of Object-Oriented Software*. Springer, 92–106.

[16] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

[17] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. 2019. Combinations of reusable abstract domains for a multilingual static analyzer. In *Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19)* (New York, USA) *(Lecture Notes in Computer Science (LNCS), Vol. 12031)*. Springer, 1–18. https://doi.org/10.1007/978-3-030-41600-3_1

[18] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09

[19] La Trobe University. 2013. *Joy Programming Language.* Retrieved 2021-11-15 from https://www.latrobe.edu.au/humanities/research/research-projects/past-projects/joy

[20] Nomadic Labs. 2021. *Dexter Flaw Discovered; Funds are Safe.* Retrieved 2022-03-20 from https://research-development.nomadic-labs.com/dexter-flaw-discovered-funds-are-s

[21] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68. https://doi.org/10.1145/3183440.3183495

[22] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* 254–269. https://doi.org/10.1145/2976749.2978309

[23] Antoine Miné. 2006. The octagon abstract domain. *High. Order Symb. Comput.* 19, 1 (2006), 31–100. https://doi.org/10.1007/s10990-006-8609-1

[24] Antoine Miné. 2006. Symbolic methods to enhance the precision of numerical abstract domains. In *Proc. of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)* (Charleston, South Carolina, USA) *(Lecture Notes in Computer Science (LNCS), Vol. 3855)*. Springer, 348–363. https://doi.org/10.1007/11609773_23

[25] Charles H Moore and Geoffrey C Leach. 1970. Forth–a language for interactive computing. *Amsterdam: Mohasco Industries Inc* (1970). http://www.ultratechnology.com/4th_1970.html

[26] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[27] Bernhard Mueller. 2017. *Introducing Mythril: A framework for bug hunting on the Ethereum blockchain.* https://medium.com/hackernoon/introducing-mythril-a-framework-for-bug-hunting

[28] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference.* 653–663. https://doi.org/10.1145/3274694.3274743

[29] Santiago Palladino. 2017. The parity wallet hack explained. *OpenZeppelin blog, https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7* (2017).

[30] João Santos Reis, Paul Crocker, and Simão Melo de Sousa. 2020. Tezla, an Intermediate Representation for Static Analysis of Michelson Smart Contracts. In *2nd Workshop on Formal Methods for*

Blockchains (FMBC 2020) (OpenAccess Series in Informatics (OA-SIcs), Vol. 84), Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:12. https://doi.org/10.4230/OASIcs.FMBC.2020.4

[31] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Cheng-nian Sun, Huizhong Li, and Yan Cai. 2021. Empirical Evaluation of Smart Contract Testing: What is the Best Choice?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 566–579. https://doi.org/10.1145/3460319.3464837

[32] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2018. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. *CoRR* abs/1812.05934 (2018). arXiv:1812.05934 http://arxiv.org/abs/1812.05934

[33] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM*

SIGSAC Conference on Computer and Communications Security. 621–640. https://doi.org/10.1145/3372297.3417250

[34] Tezos protocol developers. 2014–2022. *Michelson: the language of Smart Contract in Tezos*. Retrieved 2022-03-21 from https://tezos.gitlab.io/active/michelson.html

[35] Tezos protocol developers. 2020–2022. *Michelson Reference*. Retrieved 2022-03-21 from https://tezos.gitlab.io/michelson-reference/

[36] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 67–82. https://doi.org/10.1145/3243734.3243780

[37] Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui, and Yinxing Xue. 2020. *Clairvoyance: Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts*. Association for Computing Machinery, New York, NY, USA, 274–275. https://doi.org/10.1145/3377812.3390908