- 4 LUCA OLIVIERI, Ca' Foscari University of Venice, Italy
- <sup>5</sup> LUCA NEGRINI, Ca' Foscari University of Venice, Italy
- <sup>6</sup> VINCENZO ARCERI, University of Parma, Italy
- <sup>7</sup> THOMAS JENSEN, INRIA, France and University of Copenhagen, Denmark
- FAUSTO SPOTO, University of Verona, Italy

Once deployed in blockchain, smart contracts become immutable: attackers can exploit bugs and vulnerabilities 10 in their code, that cannot be replaced with a bug-free version. For this reason, the verification of smart contracts 11 before they are deployed in blockchain is important. However, the development of verification tools is not easy, 12 especially if one wants to obtain guarantees by using formal methods. This paper describes the development, 13 from scratch, of a static analyzer based on abstract interpretation for the verification of real-world Tezos smart 14 contracts. The analyzer is generic with respect to the property under analysis. This paper shows taint analysis 15 as a concrete instantiation of the analyzer, at different levels of precision, to detect untrusted cross-contract 16 invocations. 17

Additional Key Words and Phrases: Blockchain, Smart contracts, Program verification, Formal methods, Abstract interpretation, Tezos, Michelson, Low-level programming language, Untrusted calls

#### 20 ACM Reference Format:

18

19

21

22

23 24

25

Luca Olivieri, Luca Negrini, Vincenzo Arceri, Thomas Jensen, and Fausto Spoto. XXXX. Design and Implementation of Static Analyses for Tezos Smart Contracts. *J. ACM* 37, 4, Article 111 (August XXXX), 24 pages. https://doi.org/XXXXXXXXXXXXXXX

## 1 INTRODUCTION

In recent years, blockchain-based technologies have seen a growing interest in both academia and 26 industry. Blockchains are abstract shared data structures where data is immutable, distributed, and 27 decentralized. In this context, smart contracts are programs stored as data, that can be executed 28 within the blockchain. They were conceived as a set of promises, specified in digital form, namely 29 contracts [2]. However, their purpose is now blurred, given the generality of the software that can 30 run within modern blockchains, especially after the introduction of Turing-complete languages 31 for smart contract implementation. Once a smart contract is deployed in blockchain, it becomes 32 immutable, exactly like any other data, and it is impossible to modify its code. For this reason, 33 contract implementations must be secure against attacks and bug-free, before their deployment in 34 blockchain to avoid unexpected execution behaviors. In this context, formal verification techniques 35 allow one to analyze software with mathematical theories and ensure the presence or absence 36 of certain code properties, bugs, and vulnerabilities. However, according to Ferrara et al. [20], 37

Authors' addresses: Luca Olivieri, luca.olivieri@unive.it, Ca' Foscari University of Venice, Venice, Italy; Luca Negrini,
 luca.negrini@unive.it, Ca' Foscari University of Venice, Venice, Italy; Vincenzo Arceri, vincenzo.arceri@unipr.it, University of
 Parma, Parma, Italy; Thomas Jensen, thomas.jensen@inria.fr, INRIA, Rennes, Bretagne, France and University of Copenhagen,
 Copenhagen, Denmark; Fausto Spoto, fausto.spoto@univr.it, University of Verona, Verona, Italy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee
 provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and
 the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored.
 Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires

- prior specific permission and/or a fee. Request permissions from permissions@acm.org.
- © XXXX Association for Computing Machinery.
- <sup>7</sup> 0004-5411/XXXX/8-ART111 \$15.00
- 48 https://doi.org/XXXXXXXXXXXXXXX
- 49



Fig. 1. Development pipeline of Tezos smart contracts.

tools based on formal methods require a significant theoretical background as well as consolidated programming skills for designing and implementing a new analysis.

This paper presents an experience report on the design and implementation from scratch of MichelsonLiSA<sup>1</sup>, a static analyzer based on abstract interpretation for the verification of smart contracts executing on the Tezos blockchain, henceforth just referred to as Tezos smart contracts. It shows how LiSA [20, 33, 34] (Library for Static Analysis) facilitates this task, also for low-level languages such as Michelson [37], and how the peculiarities of smart contracts enable analyses that typically could not be applied to traditional software.

*Contributions.* This paper is an extended version of [39]. Compared to [39], it expands and clarifies all contents, providing detailed information about the design and implementation choices underlying MichelsonLiSA. Moreover, it instantiates MichelsonLiSA with taint analysis, to spot untrusted inter-contract invocations (UCCIs). Lastly, it reports experiments that demonstrate the applicability of techniques that typically do not scale for traditional software, while they do work instead on smart contracts, thanks to their conciseness.

*Paper structure.* Section 2 and Section 3 provide preliminary notions on Tezos smart contracts and blockchain software verification, respectively. Section 4 highlights design and implementation choices related to the development of MichelsonLiSA. Section 5 describes the development process of MichelsonLiSA. Section 6 uses taint analysis to detect untrusted cross-contract invocations and introduces a three levels version to improve the analysis results. Section 7 reports related work. Section 8 concludes the paper.

#### 2 TEZOS SMART CONTRACTS

Tezos [23] is a public *permissionless* blockchain based on the proof-of-stake consensus, that supports Turing-complete smart contracts. In the Tezos ecosystem, there are several frameworks for the development of smart contracts, such as Archetype [3], LIGO [27], and SmartPy [52]. Most of them exploit *meta-programming* to develop smart contracts. Meta-programming is already widely used in many blockchains [2, 10, 18, 35, 44, 54], since it allows one to develop smart contracts in different high-level languages, that all compile to a single, normally low-level target language. In this way, it

<sup>97 &</sup>lt;sup>1</sup>https://github.com/lisa-analyzer/michelson-lisa

J. ACM, Vol. 37, No. 4, Article 111. Publication date: August XXXX.

114 115 116

123

124

125

126

127

128

129

130

131 132

	Instruction	Description			
	ADDRESS	pop a contract value and push the address of that contract			
	AMOUNT	push the amount of the current transaction			
	BALANCE	push the current amount of mutez of the executing contract			
	CHAIN_ID	push the chain identifier			
	CONTRACT	replace the top of the stack after cast to a contract type			
	CREATE_CONTRACT	push a contract creation operation			
	IMPLICIT_ACCOUNT	push the address of a new implicit account			
	LEVEL	push the current block level			
	NOW	push the block timestamp			
	SELF	push the current contract			
	SELF_ADDRESS	push the address of the current contract			
	SENDER	push the contract that started the current internal transaction			
	SET_DELEGATE	push a delegation operation			
	SOURCE	push the contract that initiated the current transaction			
	TOTAL_VOTING_POWER	push the total voting power of all contracts			
	TRANSFER_TOKENS	push a transaction operation			
	VOTING_POWER	push the voting power of a contract			

Table 1. Domain-specific operations of Michelson.

is possible to switch between popular high-level languages based on the programmer's preference
and project requirements, keeping the low-level code compatible. Development frameworks for
Tezos support popular high-level programming languages (e.g., Python, OCaml, and TypeScript),
all compiled to the Michelson low-level language [37]. This is the only target language of the Tezos
blockchain (see Figure 1).

Michelson is a statically-typed domain-specific bytecode language, expressive enough to implement Turing-complete smart contracts. The memory model is stack-based and data are manipulated in a last-in-first-out (LIFO) order. Currently, Michelson consists of around 100 bytecode instructions<sup>2</sup>: for stack manipulation (PUSH, DROP, SWAP, ...), for creation and management of high-level data structures (MAP, UPDATE, SIZE, ...), for arithmetic operations (SUM, SUB, AND, ...), for control flow (IF, LOOP, ...) and blockchain-specific ones (see Table 1).

Figure 2 shows a Tezos smart contract written in SmartPy (Figure 2a), a subset of Python, and its translation into Michelson (Figure 2b) obtained with the SmartPy [52] compiler. Figure 2a shows

<sup>2</sup>https://tezos.gitlab.io/michelson-reference

```
133
                                                     1 parameter (pair %add (int %x) (int %y));
                                                     2
                                                        storage int;
134
   1
      import smartpy
                                                     3
                                                        code
135 2
                                                     4
                                                        {
     # smart contract definition
  3
136
                                                        CAR;
                                                                    # @parameter
                                                     5
     class MyAdd(smartpy.Contract):
                                                        # == add ==
137
                                                     6
   5
      def __init__(self):
                                                        # self.data.value = params.x + params.y
                                                     7
138 6
        self.init(value = 0)
                                                     8
                                                        # @parameter
  7
139
                                                     9
                                                          UNPAIR; # int : int
       @smartpy.entry_point
   8
                                                    10
                                                          ADD; # int
140
  9
       def add(self, x, y):
                                                          NIL operation; # list operation : int
                                                    11
141\,10
        self.data.value = x + y
                                                    12
                                                          PAIR; # pair (list operation) int
142
                                                    13 };
                 (a) Python code
143
                                                                    (b) Michelson code
144
145
                      Fig. 2. Meta-programming development: from Python to Michelson.
146
147
```



Fig. 3. An example of execution of the smart contract in Figure 2b.

that SmartPy contracts are defined as a class that inherits from smartpy.Contract (line 4). A 156 contract has a state and one or more entry points, annotated with @smartpy.entry\_point (line 8). 157 The constructor \_\_init\_\_ (line 5) calls self.init (an alias of self.init\_storage(arg = None, 158 \*\*kwargs)) and initializes the fields that make up the contract state (the storage of the smart 159 contract). In particular, the behavior of the program in Figure 2a is to initialize the storage with 160 the value 0 (line 6), after which blockchain users can call, through a transaction, the method 161 add(self, x, y) (line 9) to perform an addition operation between two numerical values and 162 replace the storage value with the result of the operation (line 10). Figure 2b shows the translation 163 in Michelson resulting from the SmartPy compiler. The structure of the Michelson smart contract 164 has three components: (i) an explicitly-typed parameter declaration for the input, (ii) an explicitly-165 typed storage declaration for blockchain store locations, and (iii) a code declaration that defines 166 the sequence of bytecode instructions. Technically, the input is a single value that specifies the 167 data required for running the code. However, aggregate types, such as pair and or, allow one to 168 provide more than a single input value to the contract (see line 1). 169

As already mentioned, the execution of a Michelson contract is stack-based: instructions pop and/or push values on a stack. In the Tezos blockchain, a smart contract execution request (*invocation*) specifies the address of the smart contract in the blockchain and its input.<sup>3</sup> The execution starts from a stack whose only element is the pair of the input and of the current value of the storage of the contract.

Figure 3 shows an execution of add from Figure 2b, with input Pair (5,9), assuming that the 175 current value of the contract storage is 0: the initial stack contains only one element, that is, 176 Pair (Pair (5,9),0). Note that the user provides the input, while the blockchain protocol retrieves 177 the storage value from the blockchain state. The first instruction in this example, CAR, splits the 178 pair and projects it on its first component Pair (5,9) (the input), which gets pushed on the stack 179 instead: the current storage value is discarded. The subsequent UNPAIR instruction decomposes 180 Pair (5,9) into its two components 5 and 9, that pushes on the stack instead. The ADD instruction 181 computes their sum (14), that gets pushed on the stack instead. The NIL instruction pushes an 182 empty list of operations to perform at the end of the execution and the final PAIR instruction boxes 183 the list and the result into a pair. That pair is the result of the execution. The blockchain protocol 184 will take its second component (14) and store it in the storage of the contract, for future use. 185

#### 3 BLOCKCHAIN SOFTWARE VERIFICATION

Code verification can be applied from the very beginning of the implementation of the code. For blockchain software, it is particularly important to apply it *before* code deployment in blockchain, that is, before the code becomes immutable and difficult to patch. According to Chess et al. [9], the most used approach for finding bugs is *dynamic testing*, that executes the software and compares its output with the expected result. However, dynamic testing has drawbacks. The creation of test cases is not trivial and can require a lot of effort, as developers need to compute the expected results on each input case. Namely, *unit testing* verifies small portions of code (its *units*) over

<sup>195</sup> <sup>3</sup>https://tezos.gitlab.io/michelson-reference/#execution

196

186

187

J. ACM, Vol. 37, No. 4, Article 111. Publication date: August XXXX.

153



Fig. 4. Approximation schema.

normal input or cornercases that could generate errors. Testing can observe only a finite set of
finite program executions [50, Chapt. 1.4.1]. Hence, dynamic testing can only show the presence of
bugs, never their absence [15, Part I, Chapt. 3]. In addition, dynamic testing can only be applied
from an advanced stage of development as it needs to be executed, increasing the cost of bug fixing
in case of multiple bugs.

A complementary approach to dynamic testing is *static analysis*, that automatically verifies the properties of computer programs before their execution [50]. This reduces the cost, for developers, of bug fixing, giving them the chance to fix bugs and code smells at an early stage [9]. Therefore, for full code coverage and to prove or refute a code property, such as in the case of untrusted token transfers that will be consider in this article, it is necessary to use formal methods based on mathematical frameworks, such as abstract interpretation [12].

According to Cousot [11, Chapt. 1.2], abstract interpretation [12, 13] is a unifying theory of 220 formal methods that proposes a general methodology for proving the correctness of computing 221 systems. In static program analysis, abstract interpretation is used to approximate the concrete 222 behavior of programs (their concrete semantics) with an abstract behavior (their abstract semantics). 223 It also formalizes the intuition that semantics are more or less precise depending on the abstraction 224 level. The idea behind abstract interpretation is that reasoning on the abstract properties implies 225 some reasoning on the concrete ones. The abstraction is a necessary step to perform analyses that 226 detect otherwise undecidable properties [49], that is, abstractions trade precision for decidability. 227

Abstract interpretation allows one to formalize a notion of soundness. A static analyzer is sound 228 with respect to a program and a property of interest when it considers all possible program 229 executions and is thus able to give definite guarantees on the property. Thus, if it does not issue 230 any alarm, the property is guaranteed to hold for every possible execution. In other words, sound 231 analyzers have no false negatives (situations when the property holds in at least one concrete 232 execution, but the analyzer does not detect it and no alarm is issued). In particular, soundness is 233 achieved by using over-approximations (Figure 4), that can however create *false positives* (situations 234 when the property does not hold in any concrete execution but the analyzer conservatively assumes 235 that it might hold because it considers *more* executions than the concrete ones). According to 236 Meyer [29], it is generally better to use *sound* techniques, since false negatives can lead to critical 237 issues whose mitigation might be impracticable in some contexts, such as blockchain. 238

## 4 FROM SCRATCH TO MICHELSONLISA

Static analysis based on formal methods requires a non-trivial theoretical background and development skills. In order to be able to design and implement a new analysis, it is necessary to construct an infrastructure providing its basic building blocks (parser, control flow graph (CFG) representation [1], fixpoint algorithms, etc.). Therefore, the development of even a toy static analyzer from

245

239

scratch is a big effort, unless a generic analysis infrastructure is already available, that can be used
 to reduce to development effort.

This section reports our successful experience with the design and implementation of Michelson-LiSA [39], a static analyzer for the Michelson language. In particular, it describes the challenges faced in analyzing that domain-specific language, the technologies involved for quick implementation, and the development timelines.

#### 4.1 Challenges of Tezos Smart Contract Verification

254 As reported in Section 2, the development of smart contracts for the Tezos blockchain involves differ-255 ent programming languages, both high and low-level. Switching from high to low-level languages 256 can imply a loss of information, making it difficult to understand, reverse engineer, analyze, and 257 verify blockchain software. High-level languages typically feature compact instructions, types and 258 annotations. Instead, low-level languages have a restricted instruction set and make all operations 259 performed during the execution explicit, losing expressiveness and increasing code verbosity. In 260 addition, compilation problems occur when the semantics of some high-level instruction may not 261 be easily expressed in terms of low-level instructions.

262 In this scenario, an interesting case study for Tezos is SmartPy [52]. It is a framework that allows 263 one to program smart contracts at high-level, in Python. However, Python is a general-purpose 264 language with thousands of APIs: many of them cannot be compiled into Michelson, due to its 265 domain restrictions. To overcome this problem, these functions are resolved at compile-time [53] 266 and the results are hardcoded in the compiled code. Nevertheless, this leads to other two problems. 267 The first and more immediate is that, while analyzing the Michelson bytecode, the usage of Python 268 APIs is not visible. The second is that this resolution is correct only if the API call actually returns 269 a constant, and the developer should check that. Let us explain this by means of the SmartPy smart 270 contract in Figure 5a, that gets initialized with a numerical parameter myPar1 through function 271 \_\_init\_\_ at line 6. This value can later be changed through function myEntryPoint at line 11. 272 The myEntryPoint function uses random. randint at line 13, a standard Python API that has no 273 transation into Michelson, since the latter has no instruction for generating random values, to 274 ensure deterministic execution [40, 41, 56]. But the SmartPy compiler compiles that code without a 275 single warning. Figure 5a shows the result: random.randint has been evaluated at compile-time 276 and its random return value (7) has been hardcoded in the bytecode, at line 8. When running the 277 Michelson code, it will not add a random value, as the Python programmer might naively expect, 278 but will add 7, for all executions. Moreover, the constant 7 will likely change at next compilation, 279 making the process non-deterministic. 280

In this case, code analysis at the high-level source code is not a viable choice. More generally, according to Logozzo et al. [28], the analysis of low-level code provides different advantages: (i) it is more faithful, as it analyzes the code that is actually executed (or closer to), (ii) it enables the analysis of code when source code is not available (for instance, for smart contracts already deployed in blockchain), (iii) it avoids redundant work that the compiler has already performed, such as name resolution, type checking, template/generics instantiation, and (iv) the semantics of high-level constructs is expanded by the compiler in the low-level code.

For these reasons, this paper focuses on the analysis of Michelson only.

#### 4.2 Goals, Requirements and Technologies

A static analyzer such as MichelsonLiSA is composed of at least three main components (see Figure 6):

(1) a *parsing component* that reads and interprets the code to analyze;

J. ACM, Vol. 37, No. 4, Article 111. Publication date: August XXXX.

252 253

281

282

283

284

285

286

287

288 289

290

291

292



cal background and development skills. This paper relies on the analysis engine of LiSA [20, 33, 34], a framework that provides a general infrastructure for static analysis and standard components for

abstract interpretation. It is written in Java, a popular, platform-independent, enterprise language,
that supports a wide range of tools for software development (IDEs, test frameworks, monitoring
software, debugging environments). LiSA has been successfully applied to educational [20] as well
as industrial solutions [41].

LiSA facilitates the development of analyses based on formal methods, but places some constraints, since its program model uses a representation of extensible control flow graphs where every instruction's semantics is the composition of atomic operations, agnostic w.r.t. the syntax of the source code. Moreover, LiSA is primarily designed for imperative and object-oriented languages and its application to other languages must be investigated case by case. Next sections discuss the three main components cited above and the impacts of adopting LiSA during the development of the MichelsonLiSA analyzer.

## 4.3 Code Parsing

355 356

369 370

381

382

383

384

385

386

387 388

357 A parser reads the input code, checks if it complies to the language syntax, and returns a structured 358 representation of the parsed code, to be processed in a subsequent phase. The full grammar 359 of Michelson specifies its syntax.<sup>4</sup> However, it currently lacks some syntactic sugar (such as 360 annotations, use of brackets, smart contract structure or macros) widely used in real-world Tezos 361 contracts. LiSA leaves the parsing logic to the user [33, Section 1.6]. Hence, we enriched that 362 grammar and implemented it<sup>5</sup> in the ANTLR v4 format. ANTLR [45] is a popular tool that, starting 363 from a grammar, builds a *lexer* and a *parser*. The lexer reads the input code and produces a sequence 364 of strings called *lexemes*; the parser uses an LL(\*) algorithm [46], with lexemes as input. If the code 365 complies with the grammar, the parser builds a structured *abstract syntax tree* (AST); otherwise, 366 it rejects the input code with a syntax error [11, Chapter 5] and stops. Our ANTLR grammar 367 is agnostic w.r.t. the implementation language of the lexer and parser (Java in our case): it can 368 therefore be reused in future projects written in other languages.

#### 4.4 Program Builder

371 After parsing, a program model must be built. LISA models code as a collection of CFGs (representing 372 the syntax of the input program) and provides rewriting rules of each CFG node into symbolic 373 expressions. These are an internal extensible language representing atomic semantic operations 374 (thus modeling the semantics of each instruction of the input program) [33, Section 1.6]. The ASTs 375 produced by the parser can be used as the base for building CFGs. However, according to [16], the 376 use of a stack makes it difficult to apply standard static analysis techniques. Moreover, LiSA is 377 designed for traditional high-level languages that are typically variable-based, while Michelson 378 is low-level and stack-based. Therefore, preliminary manipulations are necessary to provide a 379 program intermediate representation (IR) that matches the analysis engine. 380

MichelsonLiSA implements an IR based on static single-assignment (SSA), constructed by using a symbolic stack to translate Michelson code into variable-based code. The algorithm is inspired by BC2BIR [16] and Tezla [48]. The translation maps each Michelson instruction<sup>6</sup> into a list of MichelsonLiSA instructions (LiSA's statements expressing the syntax and semantics of the corresponding Michelson instruction), by using new fresh variables. It tracks, abstractly, the

<sup>4.4.1</sup> Intermediate Representation.

<sup>&</sup>lt;sup>389</sup> <sup>4</sup>https://tezos.gitlab.io/active/michelson.html#full-grammar

<sup>&</sup>lt;sup>390</sup> <sup>5</sup>https://github.com/lisa-analyzer/michelson-lisa/tree/master/michelson-lisa/src/main/antlr

<sup>391 &</sup>lt;sup>6</sup>https://tezos.gitlab.io/active/michelson.html#core-instructions

<sup>392</sup> 

J. ACM, Vol. 37, No. 4, Article 111. Publication date: August XXXX.

303		$v_{0} = p_{0}r_{0}r_{0}r_{0}r_{0}r_{0}r_{0}r_{0}r$
373	parameter (pair int int);	vo - parameter_storage();
394	storage int;	VI = CAR(V0);
395	code {	$v_2 = DUP(v_1);$
0,0	CAR	$v3 = get_left(v2);$
396	DIIP ·	v4 = get_right(v2);
397		v5 = COMPARE(v3, v4);
200	COMPARE .	v6 = GT(v5);
398	COMPARE,	IF(v6)
399	61;	{ # True branch
400		$v7 = get_left(v1);$
100	{ # Irue branch	$v8 = get_right(v1);$
401	UNPAIR;	v9 = ADD(v7, v8);
402	ADD;	}
100	}	{ # False branch
403	{ # False branch	v10 = get left(v1)
404	UNPAIR;	$v_{11} = get right(v_1)$
405	SUB;	$v_{11} = g_{12} (v_{11})$
105	}	V12 - 300 (V10, V11),
406		j
407	NIL operation;	$v_{13} = pn_1(v_9, v_{12});$
100	PAIR;	
408	}	v14 = NIL(operation);
409	-	v15 = PAIR(v14, v13);
410	(a) Michalson code	
110	(a) Michelson code	(b) SSA form
411		

Fig. 7. A Michelson smart contract and its translation into SSA form. 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 encapsulated in a pair, consisting of an empty list of operations and of the new storage data value.

propagation of stack values through a symbolic stack of such variables.<sup>7</sup> That is, stack elements hold symbolic names, not their exact values. Figure 7 shows the translation of a Michelson contract into SSA.

420 Instructions that push values on the stack are translated into variable assignments, with fresh 421 variables standing for stack elements, each assigned exactly once. Instructions that pop from the stack take as parameter the variables corresponding to the elements they pop. Some instructions 422 can be both producers and consumers. Figure 8 shows an example of translation in SSA for some 423 common instructions. PUSH <type> <data> pushes a constant of the declared type: it is translated 424 with a fresh new variable that gets assigned to a constant of a declared type. SUB consumes its 425 426 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 427 the stack: it is translated with a function with no return value. PAIR consumes the two topmost 428 stack elements, and packs them into a pair that pushes on the stack instead: it is translated as a 429 function with two arguments, that yields the pair. UNPAIR pops a pair, splits it, and pushes its two 430 components instead: it is translated with two functions, that select the two components and store 431 them into fresh new variables. 432

Some Michelson stack-modifying instructions perform relatively complex stack operations. Namely, SWAP exchanges the topmost two elements of the stack; DIG n shifts the stack element at depth n into the top of the stack, while DUG n does the converse. These instructions can be translated into SSA. Figure 9 shows an example for DIG 2 (the position of the elements starts at 0, which is the topmost element).

441

438

412

413

414

415 416

417

 <sup>&</sup>lt;sup>439</sup> <sup>7</sup>https://github.com/lisa-analyzer/michelson-lisa/tree/master/michelson-lisa/src/main/java/it/unive/michelsonlisa/
 <sup>440</sup> frontend/visitors/MichelsonStack.java



Fig. 10. Example of transformation of a conditional into SSA form, with a junction point. The  $\phi$ -function is written as phi.

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  $\phi$ -functions [14]. The idea is to translate instructions separately along each path, using disjoint sets of variables, and then merge the variables that stand for the same stack element along different paths at the junction point. Figure 10 shows an example.

Michelson has *stack-protecting* instructions, such as DIP n, that temporarily freeze the topmost 482 n elements of the stack, keeping them unaffected during the execution of a specified group of 483 subsequent instructions. Figure 11(a) shows a snippet of code that uses DIP 2 at line 3. There, the 484 stack holds [5, 3, 4] (from bottom to top), as reported in Figure 11(b). DIP 2 freezes its topmost 485 two elements (3 and 4) during the execution of the instructions specified inside curly braces. Namely, 486 PUSH nat 1 pushes 1 immediately below the frozen elements, instead of on top of the stack, leading 487 to the stack [5, 1, 3, 4]. Similarly, ADD pops the two topmost, unprotected stack elements 5 and 488 1 and pushes their sum immediately below the frozen elements. This behavior is reflected in the 489

111:10

474

475 476

477

478

479

480

481



Fig. 11. Michelson code that uses a DIP n instruction and its corresponding stack execution. Round brackets highlight the protected area of the stack.

SSA translation (Figure 11(d)): PUSH nat 1 becomes v1 = PUSH(nat, 5), with v1 pushed on top of the symbolic stack (Figure 11(c)). Similarly for the two subsequent PUSH instructions. At line 3, the symbolic stack will be [v1, v2, v3] and v2 and v3 will become protected. Consequently, at line 4, the PUSH instruction is translated into v4 = PUSH(nat, 1), with v4 placed below the protected area of the symbolic stack, which becomes now [v1, v4, v2, v3]. The subsequent ADD instruction operates on the unprotected elements v1 and v4 and gets translated into v5 = ADD(v1, v4), with v5 pushed immediately below the protected values.

Michelson smart contracts interact with the context of Tezos where they execute. For instance, 511 at the beginning of their execution, the stack holds a pair of the input value and of the cur-512 rent storage value. This must be made explicit in the SSA translation, as in Figure 7, with v0 =513 parameter\_storage(). Instrumentation is needed for data structures as well. Namely, Michelson 514 supports high-level data structures (sets, lists, maps, optionals) and has specific instructions to oper-515 ate on them, such as ITER, LOOP\_LEFT and IF\_CONST. These typically push additional elements on 516 the stack. For instance, ITER consumes a collection from the stack and applies a set of instructions 517 to each of its elements. These get simulated in SSA by using assignments to additional variables. 518

#### <sup>519</sup> <sub>520</sub> 4.4.2 CFG Builder.

At this point, a CFG builder can visit the IR and convert the elements into a CFG representation. 521 Thanks to this IR, the CFG needn't be specific for a low-level language. Each node corresponds to 522 a statement implementation that expresses the semantics of Michelson through symbolic expres-523 sions [33, Section 3.2.2], in order to be understandable by subsequent LiSA's analyses. Symbolic 524 expressions can be considered as an internal language of LiSA to make the semantics of a node 525 generic. The connections between one statement and another are indicated as edges. Intuitively, 526 CFGs express the syntax of the program of interest, while symbolic expressions are used to con-527 struct the semantics of CFGs by specifying the meaning of the statements in each CFG node. In 528 addition to the operations above, the builder expands the macros to analyze and handles each single 529 component separately. For instance, FAIL is a sequence of UNIT; FAILWITH to trigger a smart 530 contract failure: this is translated into two statements v = UNIT and FAILWITH(v), connected by 531 an edge. The first pushes a unit value<sup>8</sup> on the stack and the second explicitly aborts the current 532 smart contract execution and exposes the top element of the stack as the exit value of the smart 533 contract execution. 534

The output of this phase is a collection of CFGs that represent, in SSA form, the Michelson source program.

537

499

500

501 502 503

<sup>&</sup>lt;sup>538</sup> <sup>8</sup>https://tezos.gitlab.io/michelson-reference/#type-unit



Fig. 12. Gantt chart of the MichelsonLiSA development. The timeline is divided into twelve working weeks. Critical activities are shown in red, while non-critical activities are shown in blue.

## 4.4.3 Semantics of Domain Specific Operations.

Table 1 contains the current list of domain-specific operations of Michelson. Although they are 554 domain-specific, almost all of them, with the exception of CONTRACT (that can be seen as a cast), push 555 a value on the stack that depends on the run-time environment (current amount of cryptocurrency 556 in the transaction, current balance of a contract, current blockchain height, current address). In 557 static analysis, their semantics will express overapproximations of the potential run-time values, 558 that cannot be inferred statically. In general, MichelsonLiSA represents such operations as methods, 559 potentially with input parameters, that return a constant value with the return type of the operation. 560 The exact abstraction of that value will be handled at analysis time, since it has different abstractions, 561 depending on the kind of analysis. For instance, for numerical analysis, some operations could 562 return a specific numerical constant. Instead, for the UCCIs detection in Section 6.2, taintedness 563 levels will abstract the returned values, as for that of BALANCE, that gets abstracted as *clean*. 564

#### 4.5 Analysis Engine

The resulting CFGs are a program model that complies with the LiSA engine, ready to be analyzed. 567 Given the program model and additional user settings, LiSA produces an entry and an exit state 568 for each node (that is, statement) in the CFGs, containing the information inferred by the analysis. 569 These can be subsequently sent to a checker. MichelsonLiSA allows the implementation of syntactic 570 and semantic checkers. A syntactic checker performs checks that are only based on syntax (for 571 instance, check if a variable is declared). A semantic checker exploits instead both the syntactic 572 structure of the program and the semantic information produced by LiSA's analysis. In any case, it 573 is possible to save the information contained in the nodes, and generate alerts and warnings. 574

# 5 DEVELOPMENT ROADMAP

The development of a static analyzer based on abstract interpretation is an expensive task, with 577 respect to the time required and to the cost of the human resources needed to perform such 578 complex activity. An analyzer for a general-purpose language such as Java can require many years 579 of work and study to apply analyses at low-level (bytecode), supporting all language features and 580 runtimes [55]. In comparison, Michelson has a manageable number of instructions (around 100), its 581 memory model is simple and the language lacks advanced features (such as inheritance, interfaces, 582 pointers, objects, variable scoping, shadowing, concurrency). Furthermore, the use of LiSA has 583 considerably reduced the complexity of implementing MichelsonLiSA. 584

Figure 12 shows a Gantt chart of the tasks and timeline for the implementation of the first working prototype of MichelsonLiSA, supporting an information flow analysis (*taint* analysis [19, 59]) and a simple numerical analysis (*sign* analysis [12]). The activities were performed in a twelve weeks

588

549

550 551 552

553

565

566

575

111:13

window (60 person-days). The implementation was carried out by a single senior Java developer

with prior knowledge of static analysis and abstract interpretation but not of the LiSA framework. 590 After a preliminary study of Michelson, the implementation started with the definition of the 591 language grammar, using ANTLR. This activity has been marked as critical, as it is the first 592 brick of the analyzer. Moreover, it has required several refinements, given the fragmented official 593 documentation. Subsequently, lexer and parser have been implemented and tested on real-world 594 smart contracts, checking the absence of any parse error. The activities related to program building 595 have been critical because they allowed us to produce the program model on which the analyses 596 are performed. The translator into SSA form required knowledge on symbolic stack computations, 597 that in turn rely on the push/pop behavior of each individual instruction. This resulted in the 598 definition and implementation of the semantics of each instruction and, when needed, in the design 599 of additional instrumentations (such as  $\phi$ -functions and multiple pushes of values on the stack). 600 This phase required the biggest effort, in terms of time. The architecture has been developed so that 601 it can be easily expanded in the future with new instructions. For instance, the logic of the symbolic 602 stack and of the translation into SSA has been separated from the implementation of the semantics 603 of the operations, defined in terms of symbolic expressions, by using two interfaces that model, 604 abstractly, the behavior of an instruction when it pushes<sup>9</sup> or pops<sup>10</sup> stack values. Regarding the 605 semantics of the symbolic expressions, LiSA natively provides some extensible classes for the most 606 common instructions (such as numerical addition and subtraction). Moreover, across the LiSA's 607 repositories, it was possible to find several examples of the implementation of the instructions 608 semantics from which it has been possible to take inspiration, such as those from GoLiSA<sup>11</sup> and 609 PyLiSA<sup>12</sup>. This simplified the implementation task, by focusing on domain-specific issues. Analyses 610 and checks have been developed by exploiting classes, interfaces and the engine already provided 611 by LiSA. Therefore, this part was the least problematic. It was only necessary to define abstract 612 domains and checks that issue the warnings. The work terminated with testing and bug fixing on 613 several examples of real-world code. 614

#### 6 INFORMATION FLOW FOR UCCI DETECTION

Low-level code, such as that of Michelson, make blockchain software hard to understand, reverse engineer, and manually investigate. This section builds on the analysis in [39] for the detection of untrusted token transfers in Tezos smart contracts, discussing issues related to over-approximation and detection of untrusted cross-contract invocations (UCCIs). It proposes a novel version of that analysis, at different degrees of abstraction, to obtain different levels of over-approximation and consequently present warnings by priority, thus facilitating manual investigation.

#### 6.1 Untrusted Cross-Contract Invocation Problem

In the blockchain context, one of the first applications of smart contracts has been the exchange of fungible and non-fungible tokens (crypto-currencies, tickets, documents, ...). *Permissionless* blockchains such as Tezos are trustless environments composed of untrusted peers, secured by economic incentive. For this reason, a common functional requirement for smart contracts is to avoid unexpected transfers of tokens, that might happen through UCCIs, to arbitrary and potentially

637

615

616

623 624

625

626

627

628

629

630 631

 <sup>&</sup>lt;sup>632</sup> <sup>9</sup>https://github.com/lisa-analyzer/michelson-lisa/blob/master/michelson-lisa/src/main/java/it/unive/michelsonlisa/cfg/
 <sup>633</sup> statement/interfaces/StackProducer.java

 <sup>&</sup>lt;sup>10</sup>https://github.com/lisa-analyzer/michelson-lisa/blob/master/michelson-lisa/src/main/java/it/unive/michelsonlisa/cfg/
 statement/interfaces/StackConsumer.java

<sup>&</sup>lt;sup>635</sup> <sup>11</sup>https://github.com/lisa-analyzer/go-lisa/tree/master/go-lisa/src/main/java/it/unive/golisa/cfg/expression



Fig. 14. Extortionware attack model exploiting UCCIs [7, 8].

untrusted peers. In terms of the Smart contract Weakness Classification (SWC) registry, these issues can be classified as *delegatecall to untrusted callee* (SWC-112) [32].

Namely, cross-contract invocations (CCIs) are delegate calls to external contracts, thus allowing smart contracts to execute the code of other contracts in blockchain. An example is for the exchange of fungible and non-fungible tokens. In this regard, Michelson provides the TRANSFER\_TOKENS instruction (Figure 13) to withdraw tokens from the current contract balance and send them to a peer's account or to another contract. TRANSFER\_TOKENS requires three parameters:

- the target contract *c* to transfer tokens to, typed as contract *ty*, where *ty* is the type of the contract parameter;
- the tokens *z* to transfer, typed as mutez, which is a specific type for manipulating tokens;
- the value parameter *d* of the contract *c*, that must have type *ty*.

CCIs are useful, but their naive use might introduce UCCIs that a malicious agent can exploit to inject arbitrary target values, that will be executed in blockchain, leading for instance to extor-tionware attacks [7, 8]. UCCIs occur when the contract to call is parameterized with untrusted input (that is, data from outside the blockchain). Users can provide any input, also anonymously. In Michelson, input and current storage value are implicitly pushed onto the stack at the beginning of each smart contract execution. Hence, an UCCI may happen whenever one of them is used as target contract of a TRANSFER\_TOKENS.

686

665 666 667

668

669

670

671

672

673

674

675

676

677

```
687
                                                                      parameter (contract unit);
                                                                   1
                                                                   2
                                                                                 address;
       import smartpy as sp
                                                                      storage
688 1
                                                                   3
   2
                                                                      code
689
   3
      @sp.module
                                                                   4
                                                                      {
690 4
                                                                        UNPAIR:
      def main():
                                                                   5
         class Proxy(sp.Contract):
                                                                   6
                                                                        NIL operation;
   5
691
           def __init__(self, owner):
                                                                   7
                                                                        SWAP:
692
   7
             self.data.owner = owner
                                                                   8
                                                                        BALANCE:
693 8
                                                                   9
                                                                        UNIT:
   9
           @sp.entrypoint
                                                                  10
                                                                        TRANSFER_TOKENS;
694
   10
           def forward(self, callee):
                                                                  11
                                                                         CONS:
695
  11
             sp.transfer((), sp.balance, callee)
                                                                  12
                                                                        PAIR;
696
                                                                  13
                                                                      3
                          (a) Python code
```

(b) Michelson code

Fig. 15. Tezos contract containing an UCCI.

Consider for instance the attack schema in Figure 14. A blockchain user might naively deploy a 703 contract containing a vulnerable TRANSFER\_TOKENS instruction and use it to handle assets. After 704 contract deployment, its source code will remain immutable and exposed in the blockchain. An 705 attacker could discover the vulnerability of the contract and exploit it to steal the contract's assets. 706 Specifically, the attacker could redirect the target contract of the TRANSFER\_TOKENS instruction to 707 his own malicious contract, and subsequently demand a ransom or permanently take possession of 708 the stolen assets. 709

Figure 15 shows a concrete example of UCCI, actually exploitable by an attacker. The contract 710 is a proof of concept of a proxy implementation, inspired by the SWC-112 samples in the SWC 711 registry [32]. In general, proxy contracts such as those using proxy upgrade patterns [42], are 712 managed by special users called contract admin or contract owner. In Figure 15, the proxy contract 713 allows one to set up a contract owner at initialization time (line 7 of Figure 15a) and to transfer 714 the cryptocurrency in the smart contract through function forward (line 11). Every user can call 715 forward, not just the owner, hence every user can provide the callee and transfer the crypto to it, 716 leading to an UCCI. A fix requires a conditional statement to guarantee that only the owner may 717 call forward (see [32]). 718

#### Taint Analysis for UCCIs Detection 6.2 720

Previous work [39] expressed the detection of UCCIs as a taintedness problem. Taint analysis is an 721 instance of information flow analysis [17, 51], that allows one to detect if untrusted information 722 flows, explicitly, from some sources into critical program points, called sinks. In this context, program 723 variables (denoted by  $\mathbb{V}$ ) are partitioned into *tainted* (denoted by  $\mathbb{T}$ ) and *clean* (denoted by  $\mathbb{C}$ ), where 724  $\mathbb{V} = \mathbb{T} \cup \mathbb{C}$  and  $\mathbb{T} \cap \mathbb{C} = \emptyset$ . The variables in  $\mathbb{T}$  are those that *may* contain untrusted information, 725 while those in  $\mathbb C$  do not contain tainted values across all possible program executions. The analysis 726 identifies flows of information (in the form of value propagations) from variables in  $\mathbb{T}$  to variables 727 in  $\mathbb{C}$ . 728

For UCCIs detection, sources are statements providing untrusted user input. For the speicific case 729 of Michelson, the untrusted input is on the stack at the beginning of the smart contract execution, 730 as a pair of the input value and of the current storage value. Since there is no real function or 731 instruction that pushes the input on the stack, MichelsonLiSA models the presence of such a 732 pair by always prefixing the code with a call to function parameter\_storage(). To consider the 733 result of such function as tainted, parameter\_storage() is considered as a source for the UCCI 734

735

719

697

698 699

L. Olivieri, L. Negrini, V. Arceri, T. Jensen, F. Spoto



analysis. Instead, sinks are statements performing CCIs. For Michelson, they are the parameters of TRANSFER\_TOKENS instructions.

#### 6.2.1 Running Example.

111:16

During the analysis of the code in Figure 15b, MichelsonLiSA detects the explicit flow leading to an 771 UCCI and issues a true positive warning on the TRANSFER\_TOKENS instruction. Figure 16a shows 772 the flow, whose source is highlighted with a blue box and whose sink is highlighted with a red 773 box. Tainted information is propagated in gray. Namely, the analysis begins after the computation 774 of the SSA form. It identifies sources and sinks at line 1 and line 8, respectively: Tainted data is 775 propagated from  $v0 = parameter_storage()$  to  $v2 = get_right(v0)$  at line 3. Then, at line 8 776 it reaches TRANSFER\_TOKENS through v2. The program has assigned nil, the current balance of 777 the contract, and a unit value to v3, v4 and v5, respectively. These variables are clean since such 778 values are not related to the user input: they are constants or, in the case of BALANCE, they are a 779 value that is not controlled by the user. 780

At the end of taint analysis, MichelsonLiSA issues a warning at line 8 because the sink, that is, the third parameter of TRANSFER\_TOKENS, is tainted. Therefore, the user will be able to directly identify the critical point of the program highlighted by the warning, inspect the CFG with the

784

767

768 769

Analysis Exec. time Avg. time per file # Warnings 785 Taint UCCI 2h 32m 8s 9.12s 2834 786 787

Table 2. Taint analysis for UCCI detection in Michelson smart contracts.

analysis information produced by MichelsonLiSA (a simplified view is in Figure 16b), understand the data propagation path and conclude that it is a true positive.

#### Experimental Results. 6.2.2

788

789

790 791

792

806

The goal of this experimental evaluation is to test applicability and performance of MichelsonLiSA, 793 for the detection of UCCIs on Tezos smart contracts. The chosen artifact set is the same used 794 in [39], i.e. 1000 Michelson smart contracts (770060 lines of code) containing the instruction 795 TRANSFER\_TOKENS. They have been randomly retrieved from [47], a repository containing 6983 796 Michelson contracts coming from a Tezos testnet. This set has been chosen for its code diversity, in 797 terms of size and complexity, and for its many CCIs. The same would be difficult to retrieve from 798 the Tezos blockchain through its public APIs. 799

Experiments have been executed on a HP EliteBook 850 G4 equipped with an Intel Core i7-7500U 800 at 2,70/2,90 GHz and 16 GB of RAM memory, running Windows 10 Pro 64bit, with Oracle JDK 801 version 13 to run the analyzer. 802

Table 2 reports the results of the experimental evaluation. In terms of time, the analysis requires 803 less than nine seconds per smart contract, on average. The analysis issues warnings about 2834 804 cross-contract invocations distributed in 781 smart contracts. 805

#### 6.2.3 Discussion. 807

In general, the precision of an analysis depends on its abstraction level, which is often inversely 808 related to its computational cost. In particular, traditional taint analysis only tracks binary in-809 formation (taint/clean) across program variables. This makes the analysis scalable to software 810 of industrial size (between 100KLOCs and 1MLOCs) [22, 57]. Our taint analysis implementation 811 applies over-approximations to guarantee soundness. This means that clean variables definitely hold 812 trusted values, while *tainted* variables *might* contain untrusted values, being sound entiles false 813 positives, that must be disambiguated by manual investigation. However, manual investigation is 814 challenging. As Section 4.1 reports, Michelson is a low-level language and it is rather difficult to 815 reverse-engineer its code, where high-level information is lost after compilation. At the end of the 816 analysis, MichelsonLiSA provides an additional report containing the analyzed CFGs in various 817 formats (such as html and dot), with details about the computed abstractions. This allows one to 818 check, for each program point, which variables the analysis infers as tainted or clean. However, in 819 order to spot over-approximations and false positives by manual investigation, one should manually 820 recompute the entire execution stack for every single instruction and check if its execution might 821 lead to a tainted value or not, exploiting MichelsonLiSA's report. This activity is time-consuming, 822 given the complexity of some contracts. 823

Consider for instance the code in Figure 17. Its untrusted input is used to index a map containing 824 hardcoded addresses. The analysis starts by propagating the parameter and storage inputs in v0 825 = parameter\_storage(). The untrusted information of v0 flows into v1 = CAR(v0) and then 826 into  $v_3 = GET(v_1, v_2)$ . Given a key and a map, the instruction GET retrieves a value from the 827 map. Therefore, the input parameter is used to select a hardcoded address from a map. How-828 ever, our analysis propagates the untrusted information to v3 because at least one of the two 829 variables in GET(v1, v2) is untrusted. Going forward, that untrusted information propagates to 830  $v4 = extract_value(v3)$ , v6 = CONTRACT(v4), and  $v7 = extract_value(v6)$ . From there, 831 it flows into TRANSFER\_TOKENS(v10, v9, v7), where the analysis issues a warning since v7 is 832

L. Olivieri, L. Negrini, V. Arceri, T. Jensen, F. Spoto

```
834 1
       parameter int :
<mark>835</mark> 2
       storage unit :
   3
       code {
836 4
                                                                     v0 = parameter_storage();
         CAR ;
                                                                     v1 = CAR(v0);
         PUSH (map int address) {
837 5
                                                                     v2 = PUSH(map \{ 0 : "tz1KqT...b7QbPE",
838 6
         Elt 0 "tz1KqT...b7QbPE" ;
                                                                          " : "tz2VGB...S6rna5" });
           Elt 1 "tz2VGB...S6rna5"
    7
                                                                     v3 = GET(v1, v2);
839
   8
         };
                                                                     IF v4 = extract_value(v3) is None {
840 9
         SWAP;
                                                                       v5 = PUSH ("key not found");
<mark>841</mark>10
          GET :
                                                                       FAILWITH();
   11
          IF_NONE { PUSH string "key not found" ;
842
                                                                     }
               FAILWITH }
                                                                     v6 = CONTRACT(v4)
84312
         {
                                                                     IF v7 = extract_value (v6) is None {
            CONTRACT unit ;
844 13
                                                                       v8 = PUSH ("invalid contract");
845<sup>14</sup>
            IF_NONE { PUSH string "invalid contract" ;
                                                                     }
                  FAILWITH }{};
                                                                     v9 = AMOUNT();
84615
            AMOUNT ;
                                                                     v10 = UNIT();
<mark>847</mark> <sup>16</sup>
            UNIT :
                                                                     v11 = TRANSFER_TOKENS(v10, v9, v7);
   17
            TRANSFER_TOKENS ;
848
18
                                                                     v12 = NIL();
            NIL operation ;
                                                                     SWAP();
84919
            SWAP;
                                                                     v13= CONS(v11,v12);
<mark>850</mark> 20
            CONS :
                                                                     v14= UNIT();
            UNIT ;
  21
851<sub>22</sub>
                                                                     SWAP();
            SWAP:
                                                                     v15=PAIR(v13,v14);
85223
            PAIR
853<sup>24</sup>
         }
                                                                               (b) Michelson IR in SSA form
   25
       }
854
855
                     (a) Michelson smart contract
856
```

Fig. 17. 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.

untrusted. However, it is not very intuitive to label this as a false positive and spot the overapproximation, given the reduced readability of low-level languages. Untrusted information originating from parameter\_storage() at line 1 does not determine, explicitly, the target contract of TRANSFER\_TOKENS(v10, v9, v7) at line 17, that comes instead from a read-only map (declared at line 5) containing two hardcoded contract addresses. Hence, the CCI transfers tokens to known contracts, always, and is not untrusted.

Given the complexity of the low-level code under analysis, we could not manually investigate all files and compute true positive and false positive rates. However, next section considers more precise abstractions, to spot possible over-approximations and prioritize warnings, hence easing manual investigation.

# 6.3 Taint Analysis with Three Levels

Taint analysis allows the analyzer to scale to software of industrial size. However, as shown by the 873 benchmark of [39] and as also empirically evident for popular blockchains such as Ethereum [38], 874 smart contracts are typically small (few hundreds/thousands of lines of code). Therefere, it becomes 875 possible to collect more than binary information during taintness propagation without incurring 876 into scalability issues. For instance, it is possible to design a taint analysis based on three sets of 877 variables:  $\mathbb{T}, \mathbb{C}$  and  $\mathbb{P}$ , where  $\mathbb{V} = \mathbb{T} \cup \mathbb{C} \cup \mathbb{P}$  and  $\mathbb{T} \cap \mathbb{C} = \emptyset$ ,  $\mathbb{T} \cap \mathbb{P} = \emptyset$  and  $\mathbb{C} \cap \mathbb{P} = \emptyset$ . The variables 878 in  $\mathbb{T}$  are definitely tainted; those in  $\mathbb{C}$  are definitely clean; those in  $\mathbb{P}$  are *possibly tainted* (due to 879 over-approximation). In this way, the analyzer can issue warnings of different priority, when the 880 information that flows in a sink is tainted or possibly tainted. 881

871

Analysis	Exec. time	Avg. time per file	#T Warnings	<b>#PT Warnings</b>
Taint 3-level UCCI	2h 49m 18s	10.16s	2045	789

Table 3. Taint analysis with three levels for UCCIs detection of Michelson smart contracts.

#### 6.3.1 Experimental Results.

Table 3 reports the results of executing our three-levels taint analysis on the same benchmark used 889 in Section 6.2.2: **#T Warnings** is the number of warnings triggered in sinks where the information 890 was marked as tainted; **#PT Warnings** is the number of warnings triggered in sinks where the 891 information was marked as possibly tainted. The execution time is around ten seconds per smart 892 contract, on average. The analysis issues warnings about 2834 cross-contract invocations: 2045 893 are related to tainted information, while 789 are related to possibly tainted information. They are 894 distributed into 680 and 219 smart contracts, respectively (there are smart contracts that contains 895 both kinds of warnings). 896

#### 898 6.3.2 Discussion.

883 884

885 886 887

888

897

The proposed analysis distinguishes when an explicit flow from an untrusted input to a transfer token invocation *definitely* happens (when the parameter is *tainted*) or *might* happen (when the parameter is *possibly tainted*). The former are situations when there is a direct non-overapproximating flow of information from source to sink. The latter, instead, models situations when either there might be multiple execution paths leading to the sink with different taintedness, or when overapproximation has been applied. This helps the user investigate the warnings, giving priority to those that represent definite vulnerabilities.

Figure 18 shows simplified CFGs with analysis information produced by MichelsonLiSA for 906 the code in Figure 17. In general, the result of taint analysis is a set of warnings for potentially 907 vulnerable program points, with the indication of the sink parameter that is reached by tainted data. 908 909 Hence, manual investigation starts from the sinks. In Figure 18a, the only way for checking the 910 correctness of the warning is to reconstruct the flow, backwards from the sinks, because there is no 911 indication of possible overapproximations. In Figure 18b, instead, it is apparent where variables 912 are added to the set  $\mathbb{P}$ : the user can check whether that is an overapproximation or not, without 913 investigating all flows backwards.

Comparing the experimental results of the two approaches to taint analysis, the three-levels one detects around 27% of the total warnings as overapproximations. The analysis requires slightly more time than traditional taint analysis, a second more, on average. This might seem negligible, but is actually a 10% increase, on average. In industrial code analysis, this is a big difference in terms of time, assuming that there are enough resources to handle the information collected by the analysis. Instead, in the blockchain context, it is possible to design analyses that typically would not apply to traditional software, hence opening new development scenarios.

#### 6.3.3 Other limitations of Taint Analysis.

Information flow analysis understands how information flows inside a program during its execution. Information flows can be of three different categories (see Figure 19):

- *explicit flows* are those when the information in variable x is explicitly transferred to y;
- *implicit flows* are those when the information in variable y implicitly depends on the information in variable x; (for instance, an assignment guarded by x)
- *side channels* are observable properties of the execution that depend on the information in variable x (for instance, the amount of computational resources consumed).
- 930 931

922

923

924

925

926

927

928



mation.

969 970

971 972 973

979 980 three levels.

Fig. 18. Simplified CFGs, as reported by MichelsonLiSA.

Taint analysis can only detect explicit flows. Nevertheless, it is a good compromise between 974 performance and quality of the results. Historically, static information flow analysis for security 975 focused on explicit flows in order to be cheaper and have fewer false positives. Implicit flows are 976 harder to explain and understand, and their use by attackers remains a theoretical possibility only. 977 Side channels are outside the scope of this paper, as information disclosure cannot introduce UCCIs. 978

111:20



Fig. 19. Example of (a) explicit, (b) implicit, and (c) side channel flows, where h and 1 represent secret and public variables, respectively.

#### 7 RELATED WORK

987

988

993

994

995

996

997

998

999

1000

1001

This section is divided in two parts. The first part presents related work about other analyzers and frameworks for the design and implementation of new static analyses for smart contracts. This first comparison does not consider the specific analyses that have been implemented for each analyzer and framework, but the structure and domain of application of such tools only. Fistly, the specific analyses are not relevant in this comparison; secondly, the implemented analyses for each framework are often not clarified in the description of the framework or tools.

The second part of this section instead reports other analyses that are somehow related to the use of information flow for UCCIs detection, which is the new analysis introduced here.

Many smart contract verification tools have emerged since the dawn of blockchain technol-1002 ogy [58]. Most of them apply to Ethereum [2] since, historically, it was the first successful blockchain 1003 to introduce a Turing-complete language for smart contracts. Regarding Tezos, it is a relatively 1004 new blockchain, hence it does not boast a large coverage of verification tools. Mi-Cho-Coq [6], 1005 ConCert [30], and WhylSon [4] allow one to verify the functional correctness of Michelson contracts 1006 through proof assistants. They all rely on theorem proving, which requires formal specifications of 1007 the expected behavior of the code, such as pre- or post-conditions. Therefore, unlike MichelsonLiSA, 1008 their use is not fully automatic. The same holds for Helmholtz [36], that type-checks Michelson 1009 smart contracts against a user-provided specification based on a type system, by using the Z3 solver. 1010 Reis et al. [48] propose SoftCheck for data-flow analyses of Michelson code providing an IR called 1011 Tezla that linearizes the stack into a store of variables. The approach is similar to ours, especially 1012 regarding the IR form, but we followed the abstract interpretation approach instead. Bau et al. [5] 1013 extends MOPSA [31] to perform static analyses for Michelson. MOPSA is an abstract interpretation 1014 framework and the major alternative to LiSA. It is designed to compute fixpoints by induction on a 1015 program's syntax and considers a program as an extensible AST that initially contains the original 1016 source code, but that can be syntactically and semantically rewritten during the analysis. 1017

Regarding UCCIs detection, several techniques are applied for other blockchains. Contract-1018 Fuzzer [24] generates fuzzing inputs and defines test oracles to detect security vulnerabilities, in-1019 cluding problems related to UCCIs in Solidity. Mythril [43] bases the analysis on symbolic execution 1020 and concrete execution techniques to discover vulnerabilities, including UCCIs. It combines static 1021 execution with dynamic execution to improve path coverage and accuracy. SMARTSHIELD [61] 1022 dynamically highlights state changes and alterations after CCIs. Wang et al. [60] propose a general 1023 platform for defect detection in smart contracts, including UCCI issues. The platform analyzes 1024 smart contracts and obtains the semantic description of corresponding functions and variables. 1025 Hence, it generates assertions to detect the defects of smart contracts. However, as the authors 1026 acknowledge, there are still some problems that need further research and improvement, especially 1027 related to the construction of assertions, where human intervention is required. 1028

#### 1030 8 CONCLUSION

1031 Smart contract verification is challenging. Every day new blockchains are born with new program-1032 ming languages that require formal verification tools to avoid bugs and critical vulnerabilities. 1033 This paper describes how to implement and design from scratch an abstract interpretation-based 1034 static analyzer for Tezos smart contracts, relying on LiSA. This is a useful resource that supports 1035 developers, providing standard components for software verification and allowing one to reduce the 1036 implementation time. Furthermore, this paper investigated the use of taint analysis with different 1037 levels of abstraction for the detection of UCCIs. The results show, empirically, that it is possible 1038 to use abstractions not normally applicable to traditional code, opening new opportunities for 1039 the verification of smart contracts. Future work will develop other analyses and investigate other 1040 ways to improve taint analysis results, such as the introduction of backflow reconstruction on taint 1041 graphs [21]. Moreover, given the multi-language nature of LiSA [34], we will investigate blockchain 1042 interoperability to design a cross-blockchain taint analysis, involving other LiSA analyzers such as 1043 GoLiSA [40, 41].

Although this paper applies LiSA to Michelson smart contracts, the same technique can be used, in principle, on other programming languages for smart contracts. For instance, the idea of using information flow to identify UCCIs is not bound to Tezos in any way. However, the specific technicalities will change from language to language. For instance, the preliminary SSA transformation of Michelson code might not be useful for other programming languages. LiSA reduces the overhead of implementing new static analyses, but taking care of such distinguishing technical details still requires some effort and expertise.

#### 1052 ACKNOWLEDGMENTS

Work partially supported by SERICS (PE00000014), iNEST (ECS 00000043) projects funded by
 PNRR NextGeneration EU, and Bando di Ateneo per la Ricerca 2022, funded by University of
 Parma, (MUR\_DM737\_2022\_FIL\_PROGETTI\_B\_ARCERI\_COFIN, CUP: D91B21005370003), "Formal
 verification of GPLs blockchain smart contracts".

# <sup>1058</sup> REFERENCES

- 1059 [1] Frances E. Allen. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization* (Urbana-Champaign, Illinois). Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/800028.808479
- 1062 [2] A. M. Antonopoulos and G. Wood. 2018. Mastering Ethereum: Building Smart Contracts and Dapps. O'Reilly.
- 1063 [3] ArcheType. 2023. https://archetype-lang.org/ Accessed 04/2023.
- [4] Luís Pedro Arrojado da Horta, João Santos Reis, Mário Pereira, and Simão Melo de Sousa. 2020. WhylSon: Proving your Michelson Smart Contracts in Why3. arXiv e-prints (2020), arXiv-2005.
- [5] G. Bau, A. Miné, V. Botbol, and M. Bouaziz. 2022. Abstract interpretation of Michelson smart-contracts. In 11th ACM
   SIGPLAN International Workshop on the State Of the Art in Program Analysis. 36–43.
- [6] Bruno Bernardo, Raphaël Cauderlier, Guillaume Claret, Arvid Jakobsson, Basile Pesin, and Julien Tesson. 2020. Making
   Tezos smart contracts more reliable with Coq. In *International Symposium on Leveraging Applications of Formal Methods*.
   Springer, 60–72.
- [7] Alessandro Brighente, Mauro Conti, and Sathish Kumar. 2022. Extorsionware: Exploiting Smart Contract Vulnerabilities for Fun and Profit. ArXiv abs/2203.09843 (2022).
- [8] Christian Cattai. 2022. Extorsionware: Bringing Ransomware Attacks to Blockchain Smart Contracts. Master thesis.
   University of Padua, Italy.
- [9] Brian Chess and Jacob West. 2007. Secure programming with static analysis. Addison-Wesley Professional.
- [10] CosmWasm. 2023. CosmWasm Book. https://book.cosmwasm.com/ Accessed 11/2023.
- [11] Patrick Cousot. 2021. Principles of Abstract Interpretation. MIT Press.
- P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In 4th ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. ACM, 238–252.
- 1078

J. ACM, Vol. 37, No. 4, Article 111. Publication date: August XXXX.

- [13] P. Cousot and R. Cousot. 1979. Systematic Design of Program Analysis Frameworks. In 6th Annual ACM Symposium
   on Principles of Programming Languages, San Antonio, Texas, USA, January 1979. ACM Press, 269–282.
- [14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13 (1991), 451–490.
  - [15] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.). 1972. *Structured Programming*. Academic Press Ltd., Gbr.
- [16] Delphine Demange, Thomas Jensen, and David Pichardie. 2010. A provably correct stackless intermediate representa tion for Java bytecode. In Asian Symposium on Programming Languages and Systems. Springer, 97–113.
- 1085 [17] D. E. Denning. 1976. A Lattice Model of Secure Information Flow. Commun. ACM 19, 5 (1976), 236-243.
- [18] EOS.IO. 2023. EOS.IO Developer Manual. https://developers.eos.io/welcome/v2.1/manuals/index Accessed 04/2023.
- [19] Michael D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto. 2015. Boolean Formulas for the Static Identification of Injection Attacks in Java. In 20th International Conference Logic for Programming, Artificial Intelligence, and Reasoning (Lecture Notes in Computer Science, Vol. 9450). Springer, 130–145.
- [20] P. Ferrara, L. Negrini, V. Arceri, and A. Cortesi. 2021. Static Analysis for Dummies: Experiencing LiSA. In *10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Virtual, Canada) (SOAP 2021). Association for Computing Machinery, New York, NY, USA, 1–6.
- [21] Pietro Ferrara, Luca Olivieri, and Fausto Spoto. 2020. Backflow: Backward Context-Sensitive Flow Reconstruction of Taint Analysis Results. In Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings (New Orleans, LA, USA). Springer-Verlag, Berlin, Heidelberg, 23–43. https://doi.org/10.1007/978-3-030-39322-9\_2
- [22] P. Ferrara, L. Olivieri, and F. Spoto. 2021. Static Privacy Analysis by Flow Reconstruction of Tainted Data. Int. J. Softw.
   Eng. Knowl. Eng. 31, 7 (2021), 973–1016.
- [23] L.M Goodman. 2014. Tezos a self-amending crypto-ledger (White paper). https://tezos.com/whitepaper.pdf Accessed 04/2023.
- [24] Bo Jiang, Ye Liu, and W.K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In 2018
   33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). 259–269. https://doi.org/10.1145/
   3238147.3238177
- [25] Nomadic Labs. 2020. Michelson Reference TRANSFER\_TOKENS. https://tezos.gitlab.io/michelson-reference/#instr-TRANSFER\_TOKENS Accessed 04/2023.
- [26] Nomadic Labs. 2020. Michelson Reference Typing and Semantics Rules. https://tezos.gitlab.io/michelson-reference/
   #typing-and-semantics-rules Accessed 04/2023.
- 1104 [27] LIGO. 2023. LIGO Documentation. https://ligolang.org/ Accessed 04/2023.
- 1105[28] Francesco Logozzo and Manuel Fähndrich. 2008. On the Relative Completeness of Bytecode Analysis Versus Source1106Code Analysis. In Compiler Construction, Laurie Hendren (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–212.
- [29] Bertrand Meyer. 2019. Soundness and Completeness: With Precision. BLOGCACM, https://cacm.acm.org/blogs/blogcacm/236068-soundness-and-completeness-with-precision/fulltext Accessed: 04/2023.
- [30] Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. 2022. Finding Smart Contract Vulnerabilities with
   ConCert's Property-Based Testing Framework. In 4th International Workshop on Formal Methods for Blockchains (FMBC
   2022) (Open Access Series in Informatics (OASIcs), Vol. 105), Zaynah Dargaye and Clara Schneidewind (Eds.). Schloss
   Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:13. https://doi.org/10.4230/OASIcs.FMBC.2022.2
- Bagstuni Leioniz-Zehrum für informatik, Dagstuni, Germany, 2:1-2:13. https://doi.org/10.4230/OASICS.1MBC.2022.2
   [31] A. Miné, A. Ouadjaout, and M. Journault. 2018. Design of a modular platform for static analysis. In 9th Workshop on Tools for Automatic Program Analysis.
- [11] [32] Dominik Muhs. 2023. SWC Registry Delegatecall to Untrusted Callee. https://swcregistry.io/docs/SWC-112/
   Accessed: 10/2023.
- [33] Luca Negrini. 2023. A generic framework for multilanguage analysis. Ph. D. Dissertation. Universitá Ca' Foscari Venezia.
- 1116
   [34] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. 2023. LiSA: A Generic Framework for Multilanguage Static Analysis. Springer Nature Singapore, Singapore, 19–42. https://doi.org/10.1007/978-981-19-9601-6\_2

   1117
   Image: Static Analysis Springer Nature Singapore, Singapore, 19–42. https://doi.org/10.1007/978-981-19-9601-6\_2
- [35] Neo Team. 2023. NEO Documentation Smart Contracts. https://neo.org/technology#smart-contracts Accessed
   04/2023.
- [119 [36] Y. Nishida, H. Saito, R. Chen, A. Kawata, J. Furuse, K. Suenaga, and A. Igarashi. 2022. Helmholtz: A Verifier for Tezos
   Smart Contracts Based on Refinement Types. New Generation Computing 40, 2 (2022), 507–540.
- [37] Nomadic Labs. 2023. Michelson: the language of Smart Contracts in Tezos. https://tezos.gitlab.io/active/michelson. html#michelson-the-language-of-smart-contracts-in-tezos Accessed 04/2023.
- [38] G. A. Oliva, A. E. Hassan, and Z. M. Jiang. 2020. An Exploratory Study of Smart Contracts in the Ethereum Blockchain
   Platform. *Empirical Software Engineering* 25, 3 (2020), 1864–1904.
- [39] Luca Olivieri, Thomas Jensen, Luca Negrini, and Fausto Spoto. 2023. MichelsonLiSA: A Static Analyzer for Tezos. In
   2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events
   (PerCom Workshops). 80–85. https://doi.org/10.1109/PerComWorkshops56833.2023.10150247
- 1127

111:24

- 1128 [40] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. 2023. Information Flow Analysis for Detecting Non-Determinism in Blockchain. In 37th European Conference on Object-1129 Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263). Schloss Dagstuhl 1130 - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1-23:25. https://doi.org/10.4230/LIPIcs.ECOOP.2023.23 1131
- [41] L. Olivieri, F. Tagliaferro, V. Arceri, M. Ruaro, L. Negrini, A. Cortesi, P. Ferrara, F. Spoto, and E. Talin. 2022. Ensuring 1132 determinism in blockchain software with GoLiSA: an industrial experience report. In 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. 23-29. https://doi.org/10.1145/3520313.3534658 1133
- [42] OpenZeppelin. 2023. Proxy Upgrade Pattern. https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies Accessed: 1134 10/2023.1135
- [43] Nikhil Parasaram. 2020. Mythril Wiki Page. https://github.com/ConsenSys/mythril/wiki Accessed: 04/2023. 1136
- [44] Parity Technologies. 2023. Ink! Documentation. https://paritytech.github.io/ink-docs/why-rust-for-smart-contracts 1137 Accessed 04/2023.
- [45] T. Parr. 2023. ANTLR Website. https://www.antlr.org/ (Accessed 04/2023). 1138
- [46] Terence Parr and Kathleen Fisher. 2011. LL(\*): The Foundation of the ANTLR Parser Generator. In Proceedings of the 1139 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (San Jose, California, USA) (PLDI 1140 '11). Association for Computing Machinery, New York, NY, USA, 425-436. https://doi.org/10.1145/1993498.1993548
- 1141 [47] João Santos Reis. 2022. Tezla Test Repository. https://github.com/joaosreis/tezla/tree/main/tests. Com-1142 mit:baacf2a79f8ac1fee8b5200395ffc14d5b9922e6 Accessed 04/2023.
- [48] João Santos Reis, Paul Crocker, and Simão Melo de Sousa. 2020. Tezla, an Intermediate Representation for Static 1143 Analysis of Michelson Smart Contracts. In 2nd Workshop on Formal Methods for Blockchains (FMBC 2020) (OpenAccess 1144 Series in Informatics (OASIcs), Vol. 84). 4:1-4:12. https://doi.org/10.4230/OASIcs.FMBC.2020.4 1145
- [49] Henry Gordon Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the 1146 American Mathemathical Society 74 (1953), 358-366. https://doi.org/10.1090/s0002-9947-1953-0053041-6
- 1147 [50] Xavier Rival and Kwangkeun Yi. 2020. Introduction to static analysis: an abstract interpretation perspective. Mit Press.
- [51] A. Sabelfeld and A. C. Myers. 2003. Language-based information-flow security. IEEE 7. Sel. Areas Commun. 21, 1 (2003), 1148 5 - 19. 1149
- [52] SmartPy. 2023. https://smartpy.io/docs/ Accessed 04/2023. 1150
- [53] SmartPy. 2023. SmartPy Reference Constants vs Expressions. https://smartpy.io/reference.html Accessed: 04/2023. 1151
- [54] Solana. 2023. Solana Getting Started With Solana Development. https://solana.com/news/getting-started-with-1152 solana-development Accessed 04/2023.
- [55] Fausto Spoto. 2016. The Julia Static Analyzer for Java. In Static Analysis, Xavier Rival (Ed.). Springer Berlin Heidelberg, 1153 Berlin, Heidelberg, 39-57. 1154
- [56] Fausto Spoto. 2020. Enforcing Determinism of Java Smart Contracts. In Financial Cryptography and Data Security -1155 FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, 1156 Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12063). Springer, 568-583. https://doi.org/10.1007/978-1157 3-030-54455-3 40
- [57] F. Spoto, E. Burato, M. D. Ernst, P. Ferrara, A. Lovato, D. Macedonio, and C. Spiridon. 2019. Static Identification of 1158 Injection Attacks in Java. ACM Trans. Program. Lang. Syst. 41, 3, Article 18 (jul 2019), 58 pages. 1159
- [58] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal 1160 Specification and Verification. ACM Comput. Surv. 54, 7, Article 148 (2021), 38 pages. 1161
- [59] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis 1162 of web applications. In Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and 1163 Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009, Michael Hind and Amer Diwan (Eds.). Acm, 87-97. https://doi.org/10.1145/1542476.1542486 1164
- [60] Xiaoqiang Wang, Jianhua Li, and Xuesen Zhang. 2022. A Semantic-Based Smart Contract Defect Detection General 1165 Platform. In 2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications 1166 (AEECA). 34-37. https://doi.org/10.1109/AEECA55500.2022.9918903
- 1167 [61] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. SMARTSHIELD: Automatic Smart Contract 1168 Protection Made Easy. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). 23-34. https://doi.org/10.1109/SANER48275.2020.9054825 1169
- 1170
- 1171
- 1172
- 1173
- 1174
- 1175

J. ACM, Vol. 37, No. 4, Article 111. Publication date: August XXXX.