EthIR: A Framework for High-Level Analysis of Ethereum Bytecode[⋆]

Elvira Albert, Pablo Gordillo, Benjamin Livshits,
Albert Rubio, and Ilya Sergey

Abstract. Analyzing Ethereum bytecode, rather than the source code from which it was generated, is a necessity when: (1) the source code is not available (e.g., the blockchain only stores the bytecode), (2) the information to be gathered in the analysis is only visible at the level of bytecode (e.g., gas consumption is specified at the level of EVM instructions), (3) the analysis results may be affected by optimizations performed by the compiler (thus the analysis should be done ideally *after* compilation). This paper presents EthIR, a framework for analyzing Ethereum bytecode, which relies on (an extension of) Oyente, a tool that generates CFGs; EthIR produces from the CFGs, a *rule-based representation* (RBR) of the bytecode that enables the application of (existing) high-level analyses to infer properties of EVM code.

1 Introduction

Means of creating distributed consensus have given rise to a family of distributed protocols for building a replicated transaction log (a *blockchain*). These technological advances enabled the creation of decentralised cryptocurrencies, such as Bitcoin [9]. Ethereum [12], one of Bitcoin's most prominent successors, adds Turing-complete stateful computation associated with funds-exchanging transactions—so-called *smart contracts*—to replicated distributed storage.

Smart contracts are small programs stored in a blockchain that can be invoked by transactions initiated by parties involved in the protocol, executing some business logic as automatic and trustworthy mediators. Typical applications of smart contracts involve implementations of multi-party accounting, voting and arbitration mechanisms, auctions, as well as puzzle-solving games with reward distribution. To preserve the global consistency of the blockchain, every transaction involving an interaction with a smart contract is replicated across the system. In Ethereum, replicated execution is implemented by means of a uniform execution back-end—Ethereum Virtual Machine (EVM) [12]—a stack-based operational formalism, enriched with a number of primitives, allowing contracts to call each other, refer to the global blockchain state, initiate sub-transactions, and even create new contract instances dynamically. That is, EVM provides a convenient *compilation target* for multiple high-level programming languages for implementing Ethereum-based smart contracts. In contrast with prior low-level languages for smart contract scripting, EVM features mutable persistent state

[⋆] This work was funded partially by the Spanish MECD Salvador de Madariaga Mobility Grants PRX17/00297 and PRX17/00303, the Spanish MINECO projects TIN2015-69175-C4-2-R and TIN2015-69175-C4-3-R, the CM project S2013/ICE-3006 and by the UCM CT27/16-CT28/16 grant. Sergey's research was supported by a generous gift from Google.

that can be modified, during a contract's lifetime, by parties interacting with it. Finally, in order to tackle the issue of possible denial-of-service attacks, EVM comes with a notion of *gas*—a cost semantics of virtual machine instructions.

All these features make EVM a very powerful execution formalism, simultaneously making it quite difficult to formally analyse its bytecode for possible inefficiencies and vulnerabilities—a challenge exacerbated by the mission-critical nature of smart contracts, which, after having been deployed, cannot be amended or taken off the blockchain.

Contributions. In this work, we take a step further towards *sound* and *automated* reasoning about high-level properties of Ethereum smart contracts.

– We do so by providing EthIR, an open-source tool for precise decompilation of EVM bytecode into a high-level representation in a rule-based form; EthIR is available via GitHub: `https://github.com/costa-group/ethIR`.
– Our representation reconstructs high-level control and data-flow for EVM bytecode from the low-level encoding provided in the CFGs generated by Oyente. It enables application of state-of-the-art analysis tools developed for high-level languages to infer properties of bytecode.
– We showcase this application by conducting an automated resource analysis of existing contracts from the blockchain inferring their loop bounds.

2 From EVM to a Rule-based Representation

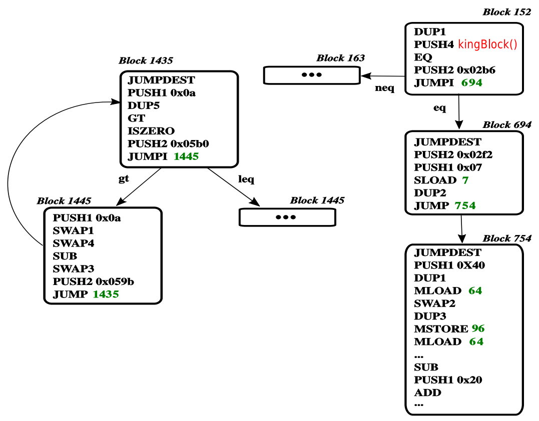
The purpose of decompilation –as for other bytecode languages (see, *e.g.*, the Soot analysis and optimization framework [11])– is to make explicit in a higher-level representation the *control flow* of the program (by means of rules which indicate the continuation of the execution) and the *data flow* (by means of explicit variables, which represent the data stored in the stack, in contract fields, in local variables, and in the blockchain), so that an analysis or transformation tool can have this control flow information directly available.

2.1 Extension of Oyente to Generate the CFG

Given some EVM code, the Oyente tool generates a set of blocks that store the information needed to represent the CFG of such EVM code. However, when the jump address of a block is not unique (depends on the flow of the program), the blocks generated by Oyente sometimes only store the last value of the jump address. We have modified the structure of Oyente blocks in order to include all possible jump addresses, so that the whole CFG is reconstructed. As an example, Fig. 1 shows the Solidity source code for a fragment of a contract (left), and the CFG generated from it (right). Observe that in the CFGs generated by our extension of Oyente, the instructions SSTORE or SLOAD are annotated with an identifier of the contract field they operate on (for instance, a SSTORE operation that stores a value on the contract field 0 is replaced by SSTORE 0). Similarly, the EVM instructions MSTORE and MLOAD instructions are annotated with the memory address they operate on (such addresses will be transformed into variables in the RBR whenever possible). These annotations cannot be generated when the memory address is not statically known, though,

```
contract BlockKing {
  ...
  uint public warriorBlock;
  uint public kingBlock;
  ...
  function kingBlock(){
    uint var = kingBlock;
    ...
  }

  function process_payment() {
    uint singleDigit  = warriorBlock;
    ...
    while (singleDigit  > 10) {
      singleDigit  -= 10;
    }
    ...
  }
}
```



```
Block 152
DUP1
PUSH4 kingBlock()
EQ
PUSH2 0x02b6
JUMPI 694

Block 694
JUMPDEST
PUSH2 0x02f2
PUSH1 0x07
SLOAD 7
DUP2
JUMP 754

Block 754
JUMPDEST
PUSH1 0X40
DUP1
MLOAD 64
SWAP2
DUP3
MSTORE 96
MLOAD 64
...
SUB
PUSH1 0x20
ADD
...

Block 1435
JUMPDEST
PUSH1 0x0a
DUP5
GT
ISZERO
PUSH2 0x05b0
JUMPI 1445

Block 1445
PUSH1 0x0a
SWAP1
SWAP4
SUB
SWAP3
PUSH2 0x059b
JUMP 1435
```

Fig. 1. Solidity code (left), and EVM code for `process_payment` within CFG (right).

(for instance, when we have an array access inside a loop with a variable index). In such cases, we annotate the corresponding instructions with "?".

Finally, when we have Solidity code available, we are able to retrieve the name of the functions invoked from the hash codes (see e.g. Block 152 in which we have annotated in the second bytecode kingBlock, the name of the function to be invoked). This allows us to statically know the continuation block.

2.2 From the CFG to Guarded Rules

The translation from EVM into our *rule-based representation* is done by applying the translation in Def. 1 to each block in a CFG. The identifiers given to the rules –*block_x* or *jump_x*– use x, the PC of the first bytecode in the block being translated. We distinguish among three cases: (1) if the last bytecode in the block is an unconditional jump (JUMP), we generate a single rule, with an invocation to the continuation block, (2) if it is a conditional jump (JUMPI) we produce two additional *guarded* rules which represent the continuation when the condition holds, and when it does not, (3) otherwise, we continue the execution in block $x{+}s$ (where s is the size of the EVM bytecodes in the block being translated). As regards the variables, we distinguish the following types:

1. *Stack variables*: a key ingredient of the translation is that the stack is flattened into variables, *i.e.*, the part of the stack that the block is using is represented, when it is translated into a rule, by the explicit variables $s_0, s_1, \ldots$, where s_1 is above s_0, and so on. The initial stack variables are obtained as parameters $s_0, s_1, \ldots, s_n$ and denoted as $\bar{s}_n$.
2. *Local variables*: the content of the local memory in numeric addresses appearing in the code, which are accessed through MSTORE and MLOAD with the given address, are modelled with variables $l_0, l_1, \ldots, l_r$, denoted as $\bar{l}_r$, and are passed as parameters. For the translation, we assume we are given a map `lmap` which associates a different local variable to every numeric address memory used in the code. When the address is not numeric, we represent it using a fresh variable local to the rule to indicate that we do not have information on this memory location.

3. *Contract fields*: we model fields with variables $g_0, \ldots, g_k$, denoted as $\bar{g}_k$, which are passed as parameters. Since these fields are accessed using SSTORE and SLOAD using the number of the field, we associate g_i to the ith field. As for the local memory, if the number of the field is not numeric because it is unknown (annotated as "?"), we use a fresh local variable to represent it.
4. *Blockchain data*: we model this data with variables $\overline{bc}$, which are either indexed with $md_0, \ldots, md_q$ when they represent the message data, or with corresponding names, if they are precise information of the call, like the gas, which is accessed with the opcode GAS, or about the blockchain, like the current block number, which is accessed with the opcode NUMBER. All this data is accessed through dedicated opcodes, which may consume some offsets of the stack and normally place the result on top of the stack (although some of them, like CALLDATACOPY, can store information in the local memory).

The translation uses an auxiliary function τ to translate each bytecode into corresponding high-level instructions (and updates the size of the stack m) and τ_G to translate the guard of a conditional jump. The grammar of the resulting RBR language into which the EVM is translated is given in Fig. 2. We optionally can keep in the RBR the original bytecode instructions from which the higher-level ones are obtained by simply wrapping them within a nop functor (*e.g.*, nop(DUPN)). This is relevant for a gas analyzer to assign the precise gas consumption to the higher-level instruction in which the bytecode was transformed.

Definition 1. *Given a block B with instructions $b_1, \ldots, b_i$ in a CFG starting at PC x, and local variables map* `lmap`*, the generated rules are:*

if $b_i \equiv$ *JUMP* p	
$block\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q)$	$\Rightarrow \tau(b_1, \ldots, b_{i-1}), call(block\_p(\bar{s}_{m-1}, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$
if $b_i \equiv$ *JUMPI* p	
$block\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q)$	$\Rightarrow \tau(b_1, \ldots, b_{c-1}), call(jump\_x(\bar{s}_m, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$
$jump\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q)$	$\Rightarrow \tau_G(b_c, \ldots, b_{i-2}) \| call(block\_p(\bar{s}_m, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$
$jump\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q)$	$\Rightarrow \neg\tau_G(b_c, \ldots, b_{i-2}) \| call(block\_(x+s)(\bar{s}_m, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$
if $b_i \not\equiv$ *JUMP and* $b_i \not\equiv$ *JUMPI*	
$block\_x(\bar{s}_n, \bar{g}_k, \bar{l}_r, \bar{bc}_q)$	$\Rightarrow \tau(b_1, \ldots, b_i), call(block\_(x+i)(\bar{s}_m, \bar{g}_k, \bar{l}_r, \bar{bc}_q))$

where functions τ and τ_G for some representative bytecodes are:

τ(*JUMPDEST*)	$= \{\};\ m := m$	
τ(*PUSHN* v)	$= \{s_{m+1} = v\};\ m := m+1$	
τ(*DUPN*)	$= \{s_{m+1} = s_{m+1-N}\};\ m := m+1$	
τ(*SWAPN*)	$= \{s_{m+1} = s_m, s_m = s_{m-N}, s_{m-N} = s_{m+1}\};\ m := m$	
τ(*ADD*|*SUB*|*MUL*|*DIV*)	$= \{s_{m-1} = s_m + \| - \| * \| / s_{m-1}\};\ m := m-1$	
τ(*SLOAD*|*MLOAD* v)	$= \{s_m = g_v \| l_{lmap(v)}\};\ m := m$	*if v is numeric*
	$= \{gl \| ll = s_m, s_m = fresh()\};\ m := m$	*otherwise*
τ(*SSTORE*|*MSTORE* v)	$= \{g_v \| l_{lmap(v)} = s_{m-1}\};\ m := m-2$	*if v is numeric*
	$= \{gs_1 \| ls_1 = s_{m-1}, gs_2 \| ls_2 = s_m\};\ m := m-2$	*otherwise*
...		
τ_G(*GT,ISZERO*)|τ_G(*GT*)	$= leq(s_m, s_{m-1}) \| gt(s_m, s_{m-1});\ m := m-2$	
τ_G(*EQ,ISZERO*)|τ_G(*EQ*)	$= neq(s_m, s_{m-1}) \| eq(s_m, s_{m-1});\ m := m-2$	
...		

RBR	$\rightarrow (B \mid J)\ \ RBR \mid \epsilon$
B	$\rightarrow block\_id\ (\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc}) \Rightarrow Instr\ \ (Call \mid \epsilon)$
J	$\rightarrow jump\_id\ (\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc}) \Rightarrow InstrJ$
$Instr$	$\rightarrow S\ \ Instr \mid \epsilon$
S	$\rightarrow s = Exp$
Exp	$\rightarrow num \mid x \mid x+y \mid x-y \mid x*y \mid x/y \mid x\%y \mid x^y$
	$\mid and(x,y) \mid or(x,y) \mid xor(x,y) \mid not(x)$
$Call$	$\rightarrow call(block\_id(\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc})) \mid call(jump\_id(\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc}))$
$InstrJ$	$\rightarrow Guard$ "|" $call(block\_id(\overline{i_n}, \overline{g_k}, \overline{l_r}, \overline{bc}))$
$Guard$	$\rightarrow eq(x,y) \mid neq(x,y) \mid lt(x,y) \mid leq(x,y) \mid gt(x,y) \mid geq(x,y)$

Fig. 2. Grammar of the RBR into which the EVM is translated

- *c is the index of the instruction, where the guard of the conditional jump starts. Note that the condition ends at the index $i-2$ and there is always a PUSH at $i-1$. Since the pushed address (that we already have in p) and the result of the condition are consumed by the JUMPI, we do not store them in stack variables.*
- *m represents the size of the stack for the block. Initially we have $m := n$.*
- *variables gs_1, gs_2 and gl, and ls_1, ls_2 and ll, are local to each rule and are used to represent the use of SLOAD and SSTORE, and MLOAD and MSTORE, when the given address is not a concrete number. For SLOAD and MLOAD we also use fresh(), to denote a generator of fresh variables to safely represent the unknown value of the loaded address.*

Example 1. As an example, an excerpt of the RBR obtained by translating the three blocks on the right-hand side of Fig. 1 is as follows (selected instructions keep using nop annotations the bytecode from which they have been obtained):

$block152(s_0, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$	$block694(s_0, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$	$s_5 = s_4$
$s_1 = s_0$ nop(DUP1)	$s_1 = 754$ nop(PUSH2)	$s_4 = s_2$
$s_2 = 6584849474$ nop(PUSH4)	$s_2 = 7$ nop(PUSH1)	$s_2 = s_5$ nop(SWAP2)
$call(jump152(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc})$	$s_2 = g_7$ nop(SLOAD)	$s_5 = s_2$ nop(DUP3)
nop(EQ) nop(PUSH2) nop(JUMPI)	$s_3 = s_1$ nop(DUP2)	$l_1 = s_4$ nop(MSTORE)
$jump152(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$	$call(block754(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc})$	$s_3 = l_0$ nop(MLOAD)
$eq(s_2, s_1)$	nop(JUMP)	$\ldots$
$call(block694(s_0, \overline{g_{11}}, \overline{l_8}, \overline{bc})$	$block754(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$	$s_3 = s_4 - s_3$ nop(SUB)
$jump152(\overline{s_2}, \overline{g_{11}}, \overline{l_8}, \overline{bc}) \Rightarrow$	$s_3 = 64$ nop(PUSH1)	$s_4 = 32$ nop(PUSH1)
$neq(s_2, s_1)$	$s_4 = s_3$ nop(DUP1)	$s_3 = s_4 + s_3$ nop(ADD)
$call(block163(s_0, \overline{g_{11}}, \overline{l_8}, \overline{bc})$	$s_4 = l_0$ nop(MLOAD)	$\ldots$

3 Case Study: Bounding Loops in EVM using SACO

To illustrate the applicability of our framework, we have analyzed quantitative properties of EVM code by translating it into our intermediate representation and analyzing it with the high-level static analyzer SACO [3]. SACO is able to infer, among other properties, *upper bounds* on the number of iterations of loops. Note that this is the first crucial step to infer the gas consumption of smart contracts, a property of much interest [4]. The internal representation of

SACO (described in [2]) matches the grammar in Fig. 2 after minor syntactic translations (that we have solved implementing a simple translator that is available in github, named `saco.py`). As SACO does not have bit-operations (namely `and`, `or`, `xor`, and `not`), our translator replaces such operations by fresh variables so that the analyzer forgets the information on bit variables. After this, for our running example, we prove termination of the 6 loops that it contains and produce a linear bound for those loops. We have included in our github other smart contracts together with the loop bounds inferred by SACO for them. Other high-level analyzers that work on intermediate forms like Integer transition systems or Horn clauses (*e.g.*, AProVE, T2, VeryMax, CoFloCo) could be easily adapted as well to work on our RBR translated programs.

4 Related Approaches and Tools

In the past two years, several approaches tackled the challenge of fully formal reasoning about Ethereum contracts implemented directly in EVM bytecode by modeling its rigorous semantics in state-of-the-art proof assistants [5, 6]. While those mechanisations enabled formal machine-assisted proofs of various safety and security properties of EVM contracts [5], none of them provided means for fully *automated* sound analysis of EVM bytecode.

Concurrently, several other approaches for ensuring correctness and security of Ethereum contracts took a more agressive approach, implementing automated toolchains for detecting bugs by symbolically executing EVM bytecode [8, 10]. However, low-level EVM representation poses difficulties in applying those tools immediately for analysis of more high-level properties. For instance, representation of EVM in Oyente, a popular tool for analysis of Ethereum smart contracts [1] is too low-level to implement analyses of high-level properties, *e.g.*, loop complexity or commutativity conditions. Zeus, a tool for analysing Ethereum smart contracts via symbolic execution *wrt.* client-provided *policies*, operates directly on Solidity sources [7]. Soundness of Zeus as an analysis approach, thus, depends on the semantics of Solidity, which is not formally defined.

References

1. Oyente: An Analysis Tool for Smart Contracts, 2018. https://github.com/melonproject/oyente.
2. E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, and G. Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *STVR*, 25(3):218–271, 2015.
3. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *TACAS*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.
4. J. Chow. Ethereum, Gas, Fuel & Fees, 2016. Published online on June 23, 2016. https://media.consensys.net/ethereum-gas-fuel-and-fees-3333e17fe1dc.
5. I. Grishchenko, M. Maffei, and C. Schneidewind. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST*, volume 10804 of *LNCS*, pages 243–269, 2018.
6. Y. Hirai. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. 2017.
7. S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, 2018. To appear.
8. L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *CCS*, pages 254–269. ACM, 2016.
9. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
10. I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *CoRR*, abs/1802.06038, 2018.
11. R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON*, 1999.
12. G. Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.