IRSRI An Intermediate Representation for Smart Contracts

Master Thesis

Author(s):
Vogel, Frédéric Henri

Publication date:
2019-05

Permanent link:
https://doi.org/10.3929/ethz-b-000343039

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

IRSRI
An Intermediate Representation for Smart Contracts

Master's Thesis

Frédéric Henri Vogel

1st May, 2019

Advisors: Prof. Dr. M. Vechev, Dr. P. Tsankov, D. Dimitrov

Department of Computer Science, ETH Zürich

Abstract

Due to the immutability of smart contracts, it is of paramount importance to ensure their correctness before deployment . In this report we introduce IRSRI, a language-independent intermediate representation (IR) built on top of a combination of static single assignment and continuation-passing style. In addition we derive the rules of an attribute grammar to translate Solidity code into IRSRI. We showcase the strength of our IR by building a prototype symbolic execution engine on top of IRSRI and use it to prove properties of Solidity smart contracts.

Acknowledgements

I thank Prof. Dr. Martin Vechev for the opportunity of writing my master's thesis in his group.

I am grateful to my supervisors Dr. Petar Tsankov and Dimitar Dimitrov who have proved invaluable with their great expertise and helpful guidance throughout this entire project.

My thanks go to all the other members of the SRI Lab that were always there to provide support in a range of subjects. I would like to specially thank Lavrenti Frobeen, who is writing his master's thesis on a related subject, for the great collaboration.

Last but not least I extend my gratitude towards my family and friends for supporting me in every situation in any possible form.

So long, and thanks for all the fish!

Contents

Chapter 1

Introduction

June 2016: an attacker exploits a bug in the code of The DAO and steals over $50 million [1].
By switching two operations in the contract code the bug would have been prevented.

July 2017: a bug in the code of a popular wallet contract is used to steal $30 million [2].
Making a single function private would have mitigated the bug.

November 2017: $150 million are frozen after a wallet library contract is killed [3].
The library contract, implementing all the wallet functionality, was not initialised. A random user was able to get ownership, allowing him to call the `selfdestruct` method.

All these examples were bugs in the smart contracts, not the underlying blockchain. Due to the special nature of blockchains and the immutability of deployed contracts it is of paramount importance to reliably find bugs and unwanted behaviour before deployment.

Several security analysis tools have since been introduced working on top of either original source language (mostly Solidity) or on the compiled Ethereum Virtual Machine (EVM) bytecode. The main advantage of analysing the bytecode lies in its guarantees. By directly analysing the code that is being run on the EVM it is certain that no errors can get inserted "down the road", e.g. by a compiler. In addition EVM bytecode has only 114 operations, making the analysis simpler compared to the analysis of a high-level language. However there are certain important drawbacks. First and foremost a significant amount of useful information is lost (or at least difficult to access) when compiling down from a high-level language. In the case of EVM code there is an additional issue with the location of storage slots of mappings and arrays. On most systems, collections are stored in a consecutive block of memory with the index representing the offset from the start address. In contrast, Solidity stores the data located at index idx of an array A with ID id_A at the storage location

$$\text{Keccak-256}(id_A || key)$$

where || indicates concatenation. Symbolic execution can infer nothing about hash functions since the output is evenly spaced. By analysing the Solidity source code we are able to symbolically analyse mappings and reflect about them.

Notwithstanding, the overall complexity of Solidity (or any other high-level language) remains prohibitively high to analyse successfully.

Contributions. To simplify the analysis of high-level languages we have developed a new language-independent intermediate representation (IR) called IRSRI built on top of a combination of static single assignment and continuation-passing style that aims at reducing the complexity of the source code while still maintaining high-level semantics. We showcase this in a case study where we translated Solidity code into IRSRI and performed symbolic execution on it with promising results in terms of complexity and usability.

Related work. Securify [4], MAIAN [5], and Rattle [6] all perform different kinds of static analysis on the EVM bytecode as do Vandal [7] and EtherTrust [8]. Other tools (OYENTE [9] and Manticore [10]) perform symbolic execution and taint analysis on bytecode. Mythril [11] uses its backend LASER-Ethereum, a symbolic virtual machine (VM), to detect a variety of vulnerabilities performing symbolic analysis based on Solidity code or EVM bytecode.

By analysing bytecode these tools cannot precisely capture the semantics of contracts because of the challenges of correctly modelling them.

Other tools have been developed that operate on Solidity code. Slither [12] performs static analysis on Solidity code translated into their IR SlithIR. The same approach is used by SmartCheck [13], another static analysis tool for Solidity based on an XML parse tree. The Ethereum Foundation is also building a satisfiability modulo theories (SMT)-based verification module within the Solidity compiler [14].

All these tools are closely tied to Solidity. It it therefore difficult to port them to other languages and to maintain them when the Solidity specifications change.

In addition to tools for performing different kinds of security analyses other analysis tools have been introduced. EthIR [15] is a framework for decompilation of EVM bytecode and representing it in a rule-based form. Scilla [16] is an intermediate language aimed at smart contracts. It aims at reducing the complexity of high-level languages and being a solid base to perform different kinds of analyses. Gigahorse [17] is a tool to decompile EVM bytecode into a three-address code (3AC) based IR. Solhint [18] is a tool to annotate Solidity code in case it detects bad security practices.

Chapter 2

Background

In this chapter we will provide some background information on two subjects. First we will have a look at the history of smart contracts and blockchains before we introduce Ethereum's smart contract programming language of choice, Solidity.

2.1 Smart Contracts – A Brief History

According to Szabo 'the contract, a set of promises agreed to in a "meeting of the minds", is the traditional way to formalize a relationship' [19]. However, whenever there is a dispute about the semantics of a contract or an eventuality that has not been foreseen, some kind of arbitration is needed, mostly performed by courts of justice. These arbitrations thus require a trusted third party. In his paper Szabo introduces the term of smart contract to denote contracts embedded in hardware and software that are digitally controlled. They allow contract clauses to be enforced proactively and offer better observability where proactive measures cannot be implemented.

However smart contracts are still missing a framework where consensus can be achieved without trusted third parties. Another concept missing the same framework is peer-to-peer electronic cash, where double-spending must be prevented. Without framework a third party is needed to verify that no token has been spent before.

In 2008 Satoshi Nakamoto presented a truly peer-to-peer solution to double-spending: the first blockchain with the name of Bitcoin [20]. The basic principle to prevent double-spending is simple: transactions are timestamped by chaining them together using a proof-of-work based on hashes. The record formed by the chain cannot be altered without redoing the proof-of-work.

Five years later in 2013 Buterin presented a new type of blockchain with an integrated almost Turing-complete programming language, Ethereum [21].

The blockchain allows for writing decentralized applications and smart contracts whose state is stored on the blockchain. The underlying currency is called Ether and is used to pay transaction fees amongst other things. As in other blockchains the state is modified by collecting single transactions into a new block containing a reference to the previous head of the chain. In contrast to Bitcoin, Ethereum has a concept of accounts. Externally owned accounts are controlled by a private key and can send messages by creating a signed transaction. Contract accounts are controlled by the code of the contract, which is activated every time the contract account receives a message.

Due to the nature of blockchains, contracts deployed to the chain are immutable and publicly visible. Any bug in a smart code can thus neither by fixed nor is it hidden from malicious parties. This makes security analysis of code so relevant in the field of smart contracts.

2.2 Programming in Solidity

Solidity is an object-oriented, statically typed language developed to target the Ethereum Virtual Machine (EVM). It is influenced by C++, Python and JavaScript and has been developed for implementing smart contracts. Objects in Solidity are called contracts and are similar to classes in other object-oriented languages. Each contract holds the code to control a single contract account.

Once a contract is created its constructor is invoked and executed. The final part of the contract is then deployed to the blockchain.

There are three different places to store items: the storage, memory and the stack. The stack and memory are quite cheap to use but are not persistent across transactions. In contrast, the storage costs more to access but is persistent.

Contracts can inherit from other contracts; multiple inheritance and polymorphism are supported. However when a contract inherits from other contracts only a single contract is deployed to the blockchain. The code from all the base contracts is compiled into the deployed contract.

We will now delve into more detail about the language with help of the bank contract shown in Listing 2.1.

The code implements a simple banking contract. It stores the current balance of each account in the mapping `balances`. There are four available functions. `payment` allows to transfer a certain `amount` from the own bank account to another. `withdraw` drains the bank account and sends all contained funds to the calling account. `deposit` allows anyone to deposit money into their bank account. Finally there is a function without name that deposits the

```
contract Bank {
    address bank_owner = address(0xDEADBEEF);
    mapping (address => uint) private balances;

    function payment(uint amount, address recipient) public {
        uint current_balance = balances[msg.sender];
        if (amount <= current_balance) {
            balances[msg.sender] -= amount;
            balances[recipient] += amount;
        } else {
            balances[msg.sender] = 0;
            balances[recipient] += current_balance;
        }
    }

    function withdraw() public {
        uint current_balance = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(current_balance);
    }

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function() external payable {
        balances[bank_owner] += msg.value;
    }
}
```

Listing 2.1: Bank contract

sent funds into the bank account specified by `bank_owner`. This is a special function which we will discuss in more detail later.

It is important to note that the mapping `balances` doesn't actually hold any funds. These are tied in the `balance` field of the account and stored on the blockchain. The mapping's only purpose is to map how much of the funds belong to which address.

We will now look at different aspects of Solidity in turn.

2.2.1 State Variables

State variables are variables that are stored in contract storage and are thus persistent across transactions. In our example we declare the state variable `balances` as we need to persistently store each user's balance as well as `bank_owner` to hold a certain account address.

2.2.2 Functions

In Solidity functions are methods that can be invoked either by other contracts or by externally owned accounts.

In contrast to other languages there are two fundamentally different ways of invoking another function in Solidity: message calls and jumps. Whereas jumps are restricted to functions inside the current contract, message calls can further invoke functions from other contracts.

Each function invoked by a message call gets executed in the context of that call. Whenever a new message call gets issued to invoke some other function this new message call is nested inside the enclosing message call of the issuer. Jumps on the other hand do not initiate new message calls but execute in the current one.

Message calls are always executed atomically in the sense that the *entire* message call including nested message calls either succeeds or gets reverted. Functions can trigger exceptions by calling the `revert` function. This will abort the current message call. All changes made to the state will be undone and the remaining gas will be returned. Whenever an exception occurs it is propagated upwards, undoing all the changes made before returning an error to the originally calling externally owned account.

If any function invoked by a jump calls `revert` the entire message call – including the caller – will be aborted and the state reverted.

Functions take a fixed number of input arguments and can return multiple values to the caller. Although being used internally to handle multiple return values, tuples are not first-order objects in Solidity.

Function Visibility

There are different levels of visibility for functions that define who can invoke them. They are

`external`
: External functions can be called from other contracts and via transactions as they are part of the contracts interface, however they cannot be called internally.

`public`
: Public functions can be called via messages or internally.

`internal`
: Internal functions can only be called internally by the current contract and all derived contracts.

`private`
: Private functions are only accessible by the current contract (and not by derived ones).

The same visibility levels also apply for state variables (with exception of `external`) and regulate who can access them.

Our example bank contract contains three functions, each with a visibility of `public`. The state variable `balances` is declared `private` and can thus not be accessed from outside of the current contract.

Fallback Function

Each contract can have exactly one unnamed function. This function may not take arguments or return anything and is invoked on a call if no other function matches the given identifier. In our bank contract we have such a fallback function, it adds the value of received Ether to the balance specified by the address `bank_owner`.

Function Modifiers

Functions can be decorated by function modifiers that are similar to Python's function decorator. They wrap a function and have access to all symbols in the scope of the wrapped function.

State Mutability

Functions can be declared as being `pure`, `view` or `payable`. `view` functions are prohibited from changing the state, whereas `pure` functions also promise not to read from state. To be able to accept Ether, functions must be declared `payable`. In the bank contract the functions `deposit` and the fallback function are declared `payable` as they need to accept Ether from another address. As the `payment` function only changes the `balances` mapping and does not actually change anything about the account's balance it does not need to be `payable`.

2.2.3 Structs and Enums

Next to contracts there are two other ways to declare new types in Solidity, structs and enums. Structs bundle several variables together. They don't offer any further functionality besides interacting with the contained variables. Enum members can be explicitly converted to any integer type and are used to represent certain options.

2.2.4 Arrays and Mappings

There are several different types of arrays and mappings in Solidity. Fixed-size byte arrays hold a sequence of 1 up to 32 bytes. They are a value type allowing them to be compared, shifted and used in bit operators as well as being accessed by index.

Regular arrays hold elements of a common type and can either have a fixed or a dynamic size.

Finally, mappings form a key-value storage that can only be declared in the contract storage. They map elements from a key type to elements from a value type where the key type must be any of the built-in value types. They do not need to be initialized since each possible key is mapped to the default value of the value type (all zero byte representation). As explained in the introduction, the value at key k in mapping M is stored at the location given by

$$\text{Keccak-256}(id_M || k)$$

where id_i is the internal ID of object i and $||$ stands for concatenation.

This concludes the overview of the history of smart contracts and the introduction to Solidity. In the next chapter we will delve into our intermediate representation (IR).

Chapter 3

IRSRI: A New Intermediate Representation

In this chapter we introduce a new language-independent intermediate representation (IR) IRSRI. It aims at reducing the complexity of high-level languages to facilitate analysis whilst keeping the high-level concepts relevant for analysis. The overall design of the IR is similar to the one developed by MLton[1].

The underlying principle is a basic-block based control-flow graph (CFG). Besides being in static single assignment (SSA) form for variables and three-address code (3AC) for statements, our basic blocks resemble continuations usually seen in continuation-passing style (CPS): each block can take arguments and has a pointer to the next block in form of a transfer that can pass arguments.

We will now describe our IR in more detail, starting with CFGs.

3.1 Control-Flow Graph

Introduced in 1970 by Allen [22], CFGs are directed graphs representing programs where each node represents a basic block of the program. A basic block is defined as being 'a linear sequence of program instructions having one entry point [. . .] and one exit point' [22] and contains program statements that are executed sequentially without branching. A basic block can have multiple predecessors and – in case of branching – have multiple successors.

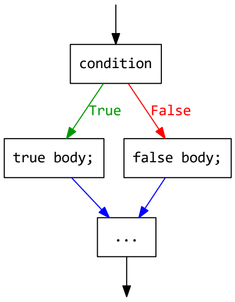
CFGs abstract away several high-level control structures and represent them in the same way. Compare the if-else statement, the while loop and the

[1] http://mlton.org/

for loop in Listings 3.1, 3.2 and 3.3. Although they have different syntaxes and elements we can represent them using the same structures, as their respective CFGs show.

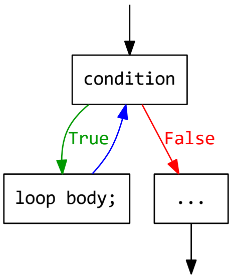
```
...
if (condition) {
    true body;
} else {
    false body;
}
...
```

Listing 3.1: If-else statement



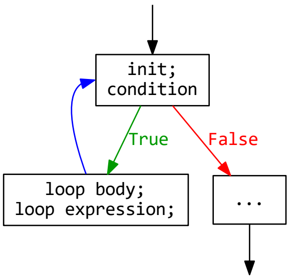
```
...
while(condition) {
    loop body;
}
...
```

Listing 3.2: While loop



```
...
for(init; condition; loop expression) {
    loop body;
}
...
```

Listing 3.3: For loop



A property often used in conjunction with CFGs is called SSA and is described next.

3.1.1 Static Single Assignment

SSA is a property in which each variable gets a value assigned to it exactly once [23]. Consider the following code example. The code in Listing 3.4 makes no use of SSA. To find out which version of a variable is used in each expression, we would need to track the definition-use chain for each

```
function foo(uint a, uint b) returns(uint) {
    a = a + b;
    b = a * a;
    uint c = b - 1;

    return c;
}
```

Listing 3.4: Without SSA

```
function foo(uint a0, uint b0) returns(uint) {
    uint a1 = a0 + b0;
    uint b1= a1* a1;
    uint c0 = b1 - 1;

    return c0;
}
```

Listing 3.5: With SSA

variable. A definition-use chain maps a single definition D_v of a variable v to all uses U_v where no other definitions D'_v exist on the path between D_v and U_v.

SSA solves this problem. In Listing 3.5 we introduce a new variable (represented as numbered versions) every time we assign to a variable. Since each version is assigned to exactly once it becomes trivial to find its uses in the following code.

An important property of SSA is that each use of a variable is strictly dominated by its definition. For many optimizations on and analyses of SSA it is in fact a necessary property [24].

A peculiarity of SSA will become visible in Listing 3.6. It represents the absolute function and works as follows. We check if the argument `a` is less than zero. If it is we assign its negative to itself, we then return the variable.

In Listing 3.7 we bring the code into SSA form. However, at the `return` statement in line 5 we do not know which version of the variable to use.

Figure 3.1 shows the CFG of the code. As one can see, depending on the condition the variable `a1` will be defined or not. When reaching the `return`

```
function abs(int a) public returns(int) {
    if (a < 0) {
        a = -a;
    }
    return a;
}
```

Listing 3.6: Branching without SSA

```
function abs(int a0) public returns(int) {
    if (a0 < 0) {
        int a1 = -a0;
    }
    return a??;
}
```

Listing 3.7: Branching with SSA

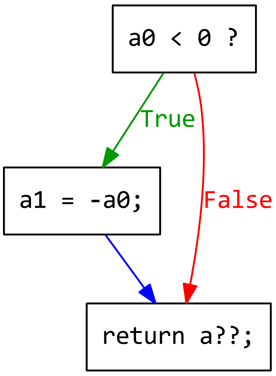


Figure 3.1: The CFG of Listing 3.7

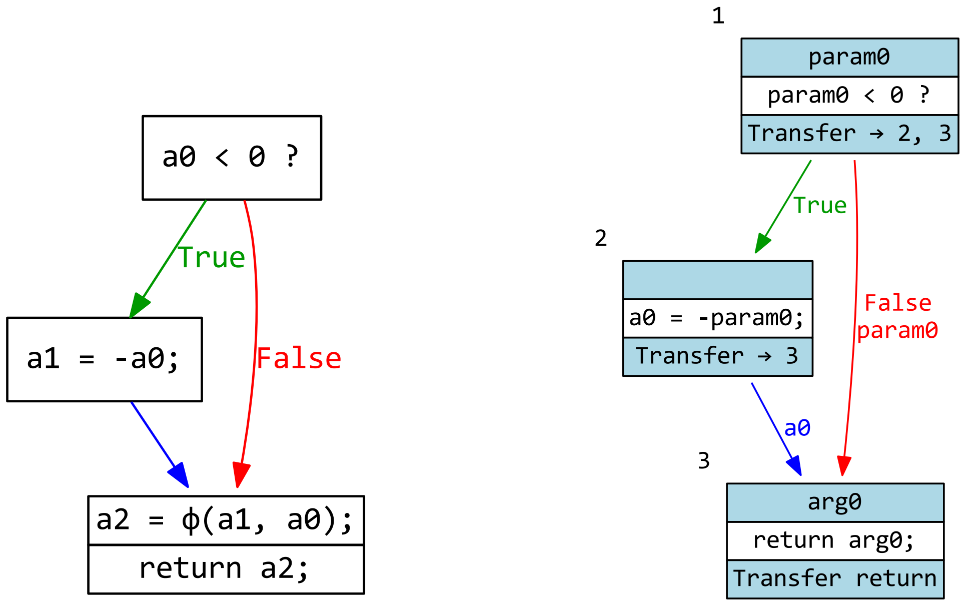
statement we need to either return a1 if the condition was true or a0 it it was false.

ϕ-Functions

Rosen et al. introduce ϕ-functions to deal with the problem of multiple possible values at the same node [23]. For any node **u** with an assignment of the form $\mathbf{V_k} \leftarrow \phi(\mathbf{V_i}, \mathbf{V_j})$ the ϕ-function is defined as follows. If control reaches **u** by the left inedge, then $\mathbf{V_k} \leftarrow \mathbf{V_i}$. If control reaches **u** by the right inedge, then $\mathbf{V_k} \leftarrow \mathbf{V_j}$. Thus the ϕ-function "magically" returns the correct value, depending from where the control reaches it.

A way to avoid ϕ-functions was introduced by the developers of the Standard ML compiler MLton[2] [25]. Their basic blocks are similar to functions,

[2] http://mlton.org/



(a) Using ϕ-functions

(b) Using arguments, the edges show the values passed

Figure 3.2: The CFGs of Listing 3.7 using ϕ-functions and arguments

in that they can take arguments. Whenever a ϕ-function would have been input in standard SSA, the basic block in MLton will take the value as an argument. In addition each basic block has a continuation which indicates where the execution continues after the current basic block. In this sense continuations are similar to a call of the next block, passing the necessary arguments.

Our IR is heavily influenced by MLton and uses the same basic design. Each basic block can take arguments and has a transfer to the next block. By being in SSA form each block has implicitly access to the variable scope of previous blocks.

Figure 3.2 shows the CFG of the same code as Figure 3.1. Figure 3.2(a) uses ϕ-functions while Figure 3.2(b) uses arguments and transfers.

Handling basic blocks as functions and giving each block a continuation to the next one are elements that come from an alternative to SSA called CPS.

It is known that SSA can be transformed into CPS and vice-versa [26]; moreover Appel makes the argument that SSA is in itself a form of functional programming [24].

Now that we have laid out the basic principles influencing our IR we will follow with a more in-depth explanation of all aspects.

Kind	IR Type	Description
CFGNodes	`SourceUnit`	Represents single source unit
	`Contract`	Represents single contract
	`Block`	Represents single basic block
	`Function`	Represents single function
	`Statement`	Represents single statement
	`Comment`	Debug node
Expressions	`Assignment`	Assignment to local variable
	`Argument`	Input to `Block`
	`Emit`	Represents the `emit` keyword
	`Const`	Constant value
	`MagicVariable`	Special Solidity variable (`msg.sender`, . . .)
	`Mapping`	Mapping variable
	`Array`	Array variable
	`MemberStore`	Store into member (`obj.member = v`)
	`MemberLoad`	Load from member (`obj.member`)
	`BinaryOp`	Binary operation
	`UnaryOp`	Unary operation
	`StateVariableLoad`	Reading value of state variable
	`StateVariableStore`	Assignment to state variable
	`ArrayLoad`	Load from array (`a[i]`)
	`ArrayStore`	Store into array (`a[i] = v`)
	`MappingLoad`	Load from mapping (`m[key]`)
	`MappingStore`	Store into mapping (`m[key] = v`)
	`Placeholder`	Represents _ in modifiers
Transfers	`Goto`	Simple transfer to other block
	`Jump`	Internal function call
	`Call`	External function call
	`Branch`	Branch
	`Return`	Function return

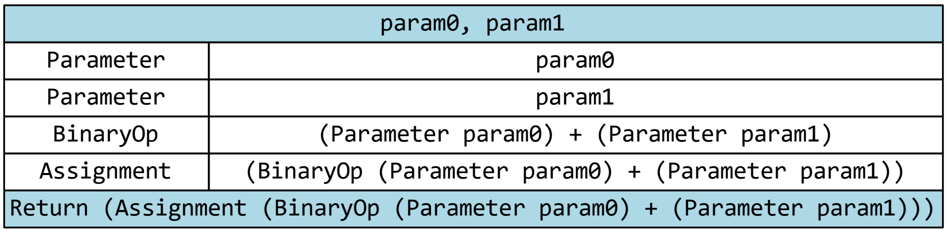
Table 3.1: Overview of IR

3.2 IRSRI in Detail

We will now further describe our IR. Table 3.1 gives an overview of all the possible IR types and a short description of each. The main element is the `Block`, which represents a basic block of the CFG. Each `Block` can have multiple `Argument` objects, a list of `Statement` objects and a `Transfer` object pointing to the next `Block`.

```
function sum(uint a, uint b) public returns(uint) {
    uint c = a + b
    return c;
}
```

Listing 3.8: Sum function



param0, param1	
Parameter	param0
Parameter	param1
BinaryOp	(Parameter param0) + (Parameter param1)
Assignment	(BinaryOp (Parameter param0) + (Parameter param1))
Return (Assignment (BinaryOp (Parameter param0) + (Parameter param1)))	

Figure 3.3: The CFG (single block) of Listing 3.8 implementing SSA using the previous objects

A Function represents a single function in Solidity. It contains a pointer to the entry Block of the function. Function modifiers are not modelled separately, their effects are inlined into the CFG of the function that use them.

All Function instances of a Solidity contract are bundled in a Contract object. Additionally, Contract objects contain Expression objects representing the declaration of state variables.

We differentiate two basic methods of representing variables. For local variables we use SSA by using an Assignment object. The ”new” variable is represented by the object itself, the value is held in the object.

For non-local variables as well as local variables of not-primitive type (arrays, members) we use the special methods *x*Load and *x*Store where *x* can be any of the following: Member, StateVariable, Array or Mapping. When one of the listed objects is accessed, we emit a new *x*Load object. When one of the listed objects is assigned to, we emit the corresponding *x*Store object.

Expressions are represented by several objects. Besides the *x*Load and *x*Store objects described previously, we define a BinaryOp object for binary operations, a UnaryOp object for unary operations and Const for constants. Additionally we introduce Array and Mapping for the variable (the pointer) of arrays and mappings (accessing those is done as explained above).

Finally we introduce Transfer objects that represent edges between Block instances. They are Goto, Jump, Call, Branch and Return. They all contain arguments and – except for the Return – have a reference to the following

Block instances. Whereas Goto objects represent a simple transfer from one Block instance to the next, Jump and Call objects are more complex. Both represent the transfer to another function; Jump objects point to functions in the same contract, Call objects to functions in another contract. Branch objects are used in conditional jumping. Each Branch contains a reference to a condition Expression and point to two Block instances: one if the condition evaluates to true, the other if the condition evaluates to false.

We will now take a closer look at Block objects to describe how the arguments work and how statements are listed.

3.2.1 The Block Object

As we have seen before, a single basic block is represented by a Block in our IR. Each Block consists of three parts: a list of arguments, a list of statements and finally a transfer. We will now describe each part in turn.

Arguments

As described in Section 3.1.1 we eliminate the use of ϕ-functions by introducing arguments to our basic blocks. We have two different kinds of arguments in our IR. The first is the Argument. It represents a general argument in a basic block and is needed after branches or function calls to hold the return values.

To determine which arguments are needed, we need to differentiate between branches and function calls. In the latter case we introduce an argument for each return value. As described in Section 2.2, Solidity handles return values as tuples, which can be nested. We perform a linearisation on nested tuples such that there is an argument for each single return value.

In the case of branching, an argument is needed for each local variable that has been changed between the branch and join. Each following reference to one of those variables will access the Argument object instead of the current assignment value.

We also introduce a subtype of Argument objects called Parameter. They are used to represent arguments to a function in the original code.

Now that the Block is initialised with its arguments, we will explain how the list of statements is structured.

Statements

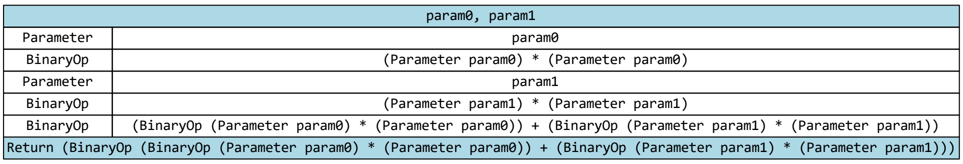
The central part of each Block is its list of statements. When executing a basic block, each statement is executed in order and sequentially. Each Statement contains exactly one Expression representing the final value of

```
function pythagoras(uint a, uint b) public returns(uint) {
    return a*a + b*b;
}
```

Listing 3.9: Without 3AC

```
function pythagoras(uint a, uint b) public returns(uint) {
    uint t0 = a;
    uint t1 = t0 * t0;
    uint t2 = b;
    uint t3 = t2 * t2;
    uint t4 = t1 + t3;
    return t4;
}
```

Listing 3.10: With 3AC



param0, param1	
Parameter	param0
BinaryOp	(Parameter param0) * (Parameter param0)
Parameter	param1
BinaryOp	(Parameter param1) * (Parameter param1)
BinaryOp	(BinaryOp (Parameter param0) * (Parameter param0)) + (BinaryOp (Parameter param1) * (Parameter param1))
Return (BinaryOp (BinaryOp (Parameter param0) * (Parameter param0)) + (BinaryOp (Parameter param1) * (Parameter param1)))	

Figure 3.4: The CFG (single block) of Listing 3.10 showcasing 3AC and SSA using the previous objects

the `Statement`. To facilitate any further analysis we keep our list of statements in 3AC form.

In 3AC form each statement is of the form `x = y op z`. If nested expressions are present in the original code, we split them up and bring them into 3AC form. An example of such a translation can be found in Listings 3.9 and 3.10. First the values are assigned, then the multiplication is handled and finally both results of the multiplications are added together.

3AC can be very easily combined with SSA as is shown in Figure 3.4. Each statement in 3AC form gets translated to a single SSA statement and each new temporary variable for intermediate results is represented by the corresponding object in our IR.

After having described `Argument` and `Parameter` objects as well as the inner workings of the `Statement` object, we are still missing the final part of each `Block`, the `Transfer`. We will present it next.

Transfers

In this section we will describe some more features of `Transfer` objects; this will conclude our description of the inner workings of the IR.

The sole purpose of `Transfer` objects is to link the basic blocks together, so to speak they form the edges of the CFG. As described previously, we introduce five different types of transfers to represent different cases. All transfers can carry arguments to the next `Block`.

The first one is the `Goto`. It represents a simple transfer to a different block and is used to join two branches back together.

Next are `Jump` and `Call` objects. They both represent a transfer to another function and differ only in their meaning. `Jump` objects are used whenever a function of the same contract is called; `Call` objects are used for external functions. Besides the destination they transfer to, both also hold a reference to the continuation block. This continuation is needed to know which block to execute after the function returns.

The next type of transfer we use is `Branch`. Essentially they represent conditional jumps. Each `Branch` object has a reference to a condition `Expression` and points to two blocks. Depending on the condition, the execution will continue in the one or in the other block. Additionally arguments are stored for each branch separately.

Last there is the `Return` transfer. It does not contain a link to another block, however it holds arguments. `Return` objects are emitted for every `return` keyword used in the code. An additional `Return` object is issued for implicit returns. The arguments contain the values being returned.

This concludes the chapter about IRSRI. We have introduced a new IR based on MLton using a combination of SSA and CPS. In the next chapter we will show how to translate from Solidity to IRSRI using attribute grammars before performing a case study that aims at demonstrating the advantages of IRSRI.

Chapter 4

Attribute Grammars

In this chapter we will introduce attribute grammars and use them to translate from Solidity code to a IRSRI representation.

One of the most used methods to translate from an abstract syntax tree (AST) to an intermediate representation (IR) is a Visitor Pattern. Introduced by the Gang of Four in 1994 [27], it allows to add (and change) functionality to a structure without changing the structure itself. In compilers the underlying structure is an AST and several visitors can be written to handle different tasks on that AST.

However Visitor Patterns not only define the *what* but also the *how*. The order of traversal is fixed as is the number of passes over the AST. Additionally, several different visitors might be needed for different tasks creating overhead.

That is why we decided to base our translation from the AST to our IR on attribute grammars. Invented in 1968 by Donald Knuth and Peter Wegner [28], they provide a way of assigning meaning to a language via attributes of the grammatical categories of the language. These attributes are defined by functions that are associated to each production rule of the language's grammar. By definition, attributes only define the *what* and do not make any statement about how the value of the attributes must be computed.

Since the evaluation of the grammar is orthogonal to the grammar itself it allows us to implement different evaluation methods without the need to change the underlying grammar. On the other other hand we can change the meaning of the grammar without changing its evaluation. Details on the evaluation method are described in [29].

4.1 Introductory Example

We will illustrate attribute grammars with an example from [28]. The example defines the meaning of the binary notation for writing numbers.

The language has the following context-free grammar:

$$\begin{aligned} B &\to \mathit{0} \\ B &\to \mathit{1} \\ L &\to B \\ L &\to LB \\ N &\to L \\ N &\to L \cdot L \end{aligned}$$

The terminal symbols of this grammar are *0*, *1* and $\cdot$, while the non-terminal symbols are B, L and N. The grammar defines strings made from a sequence of the symbols *0* and *1* potentially followed by the symbol $\cdot$ and a further sequence of the symbols *0* and *1*.

It is important to note the difference between the symbol *0* and the mathematical concept of the number zero. In itself the formal grammar only gives rules to produce certain strings.

If we wish to attach some meaning to the rules and give them certain semantics, we can add attributes to the grammar. We want the grammar to have semantics about binary numbers. Therefore we introduce the attribute $v(S)$ holding the meaning of a value to each non-terminal symbol S. We define that it is of type integer for non-terminals B and L; for N it is of type rational number. We introduce one more attribute to the non-terminal L: $l(L)$ of type integer that denotes the length of the bit sequence to which the symbol expands to.

By adding semantic rules to our formal grammar, the mere string of symbols we started with can now represent the concept of binary numbers. The attribute $v(S)$ holds the value of the symbol S while the attribute $l(L)$ holds the length of a list of bits L. The semantic rules are

$$\begin{array}{ll} B \to \mathit{0} & v(B) = 0 \\ B \to \mathit{1} & v(B) = 1 \\ L \to B & v(L) = v(B), \quad l(L) = 1 \\ L_1 \to L_2B & v(L_1) = 2v(L_2) + v(B), \quad l(L_1) = l(L_2) + 1 \\ N \to L & v(N) = v(L) \\ N \to L_1 \cdot L_1 & v(N) = v(L_1) + v(L_2)/2^{l(L_2)} \end{array}$$

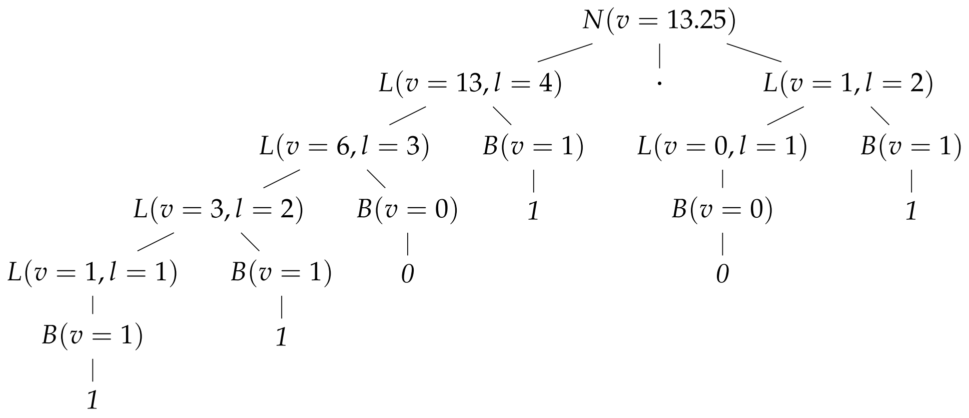


Figure 4.1: Attribute evaluation for *1101 · 01*

As an example we take the string *1101 · 01* produced by our formal grammar. By evaluating the attribute $v(\mathit{1101} \cdot \mathit{01})$ according to our semantic rules we will get the value of the string in the context of our semantics, which is $v(\mathit{1101} \cdot \mathit{01}) = 13.25$ The evaluation of the string *1101 · 01* is shown in Figure 4.1.

All the attributes used until now depend only on the descendants of the non-terminal symbols. They are called synthesized attributes.

In the context of our current semantic rules for any two symbols B_1 and B_2 the value $v(B_1) = v(B_2)$ if $B_1 = B_2$. However when considering binary numbers, the n-th closest bit to the radix point left of it represents the value 2^{n-1}. On the right hand side of the radix point, the n-th-closest bit represents the value 2^{-n} . We want to adapt our semantic rules to behave accordingly. To this end it is necessary to add some information about the distance from the radix point to each symbol B. We introduce a new attribute $s(S)$ of type integer that holds positional information in form of a scale. Starting from the radix point, the n-th bit B_n will have $s(B_n) = n$ on the left hand side and

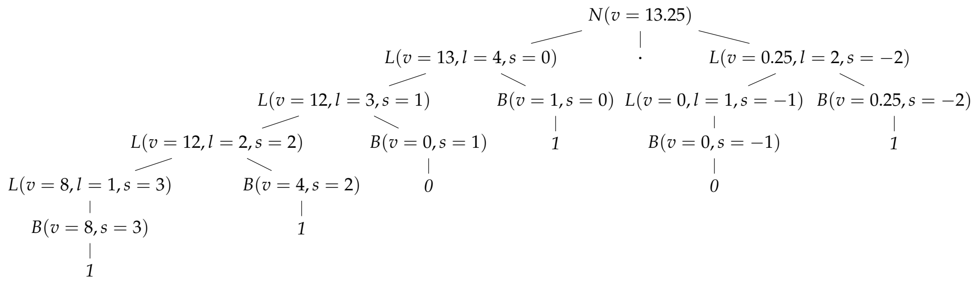


Figure 4.2: Attribute evaluation for *1101 · 01* using the new rules

$s(B_n) = -n$ on the right hand side. Our new semantic rules are thus

$B \to 0$	$v(B) = 0$	
$B \to 1$	$v(B) = 2^{s(B)}$	
$L \to B$	$v(L) = v(B)$ $s(B) = s(L)$	$l(L) = 1,$
$L_1 \to L_2 B$	$v(L_1) = v(L_2) + v(B),$ $s(L_2) = s(L_1) + 1,$	$s(B) = s(L_1),$ $l(L_1) = l(L_2) + 1$
$N \to L$	$v(N) = v(L),$	$s(L) = 0$
$N \to L_1 \cdot L_1$	$v(N) = v(L_1) + v(L_2)/2^{l(L_2)},$ $s(L_2) = -l(L_2)$	$s(L_1) = 0,$

The scale attribute $s(B)$ is defined when the non-terminal B is on the *right* side in the syntactic rule $L_1 \to L_2 B$. It therefore depends on the ancestor of the non-terminal symbol; those attributes are called inherited attributes.

To evaluate the value attribute in N we need to evaluate the length attribute from the bottom up first. We can then evaluate the scale attribute from the top down and finally evaluate the value from the bottom up again. The resulting evaluation tree for the example *1101 · 01* is shown in Figure 4.2.

This example introduced the basic idea of attributes, which are semantic information attached to a grammar. Attributes come in two forms: synthesized attributes only depend on attributes from descendent non-terminal symbols; inherited attributes only depend on ancestor attributes.

After this short introduction into attribute grammars we will now describe some attributes used in the translation from Solidity to IRSRI.

4.2 The Semantic Rules Translating Solidity

There are several concepts we need to handle during the translation from Solidity to IRSRI. We will now present the most important ones as well as showing the formal definition of some grammar rules.

4.2.1 IRSRI Representation

The primary semantics we want to extract from the AST is the representation of the Solidity code in IRSRI. We introduce an attribute `cfg` that contains the meaning of this concept. The attribute is defined for each rule in the grammar of the Solidity AST. For a node N in the AST the IRSRI representation of the subtree with root N is returned by evaluating `N.cfg`.

To translate an entire source unit into IRSRI we follow the following procedure:

1. Get the AST of the contract from the Solidity compiler `solc`[1].
2. Transform the returned JSON-object into our class-instance based representation holding the attribute grammar rules.
3. Evaluate the `cfg` attribute of the `SourceNode` object

Every attribute grammar needs to be backed by an evaluation method. When evaluating an attribute, the method works out all the dependencies between used attributes and evaluates them in a suitable order. Our evaluation method is presented in [29].

To evaluate the correct IRSRI representation we need to handle further concepts. We will explain the two concepts of the current basic block and variables next.

4.2.2 Current Basic Block

Since our IR is essentially a control-flow graph (CFG) we need to correctly identify basic blocks in the code. We do this using the following three attributes:

Attribute name	Type	Description
`current_block_pre`	inherited	Current basic block *before* the node
`current_block`	synthesized	"Working copy" for current node
`current_block_post`	synthesized	Basic block *after* the node

We show the formal rules for the current block mechanism in Table 4.1. For the production rule $B \rightarrow \emptyset$ (block without any statement) the working copy

[1] https://solidity.readthedocs.io/en/v0.5.7/using-the-compiler.html

$B \rightarrow \emptyset$	$B.current\_block = \text{clone}(B.current\_block\_pre)$ $B.current\_block\_post = B.current\_block$
$B \rightarrow S$	$B.current\_block = \text{clone}(.current\_block\_pre)$ $S.current\_block\_pre = B.current\_block$ $B.current\_block\_post = \begin{cases} B.current\_block, & \text{if type}(S) \in \mathfrak{D} \\ S.current\_block\_post & \text{otherwise} \end{cases}$
$B \rightarrow S_1, S_2, \ldots, S_n$	$B.current\_block = \text{clone}(B.current\_block\_pre)$ $S_1.current\_block\_pre = B.current\_block$ $S_{i+1}.current\_block\_pre = S_i.current\_block\_post$ $B.current\_block\_post = \begin{cases} B.current\_block, & \text{if type}(S_1) \in \mathfrak{D} \\ S_{i-1}.current\_block\_post & \text{if type}(S_i) \in \mathfrak{D} \\ S_n.current\_block\_post & \text{otherwise} \end{cases}$

Table 4.1: Attribute grammar rules for AST-node block B. Each block can have multiple statements S. $\mathfrak{D}$ is the set containing the node types return, continue and break.

`B.current_block` is defined as the clone of `B.current_block_pre`. Since there are no statements that could change the "state" of the current block `B.current_block_post` is equal to `B.current_block`.

For blocks that contain a single statement `S` we need to slightly adapt the rules. The value of the inherited attribute `S.current_block_pre` in the statement is set to `B.current_block`. This ensures S has a reference to the current basic block which it can use to e.g. add a statement to it. Since the statement could change the current basic block (if it contains a function call for example) the current basic block for following AST blocks will be different. Therefore the value of `B.current_block_post` corresponds to `S.current_block_post`. However if the type of S is any one of `return`, `continue` or `break` they break the execution flow and `B.current_block_post` takes the value of `B.current_block`.

For a block with multiple statements we chain the `current_block_post` attribute of one statement to the `current_block_pre` attribute of the next statement. Additionally we also check for each statement if it is of type return, continue or break and assign `S.current_block_post` of the previous statement to `B.current_block_post`.

4.2.3 Variables

Next to the concept of basic blocks we also need to correctly translate local variables into SSA. This is done with three attributes that behave similarly to the ones presented to handle basic blocks. The formal grammar for the three attributes `variables_pre`, `variables` and `variables_post` is shown

$B \rightarrow \emptyset$	$B.variables = \text{clone}(B.variables\_pre)$ $B.variables\_post = B.variables$
$B \rightarrow S$	$B.variables = \text{clone}(.variables\_pre)$ $S.variables\_pre = B.variables$ $B.variables\_post = \begin{cases} B.variables, & \text{if type}(S) \in \mathfrak{D} \\ S.variables\_post & \text{otherwise} \end{cases}$
$B \rightarrow S_1, S_2, \ldots, S_n$	$B.variables = \text{clone}(B.variables\_pre)$ $S_1.variables\_pre = B.variables$ $S_{i+1}.variables\_pre = S_i.variables\_post$ $B.variables\_post = \begin{cases} B.variables, & \text{if type}(S_1) \in \mathfrak{D} \\ S_{i-1}.variables\_post & \text{if type}(S_i) \in \mathfrak{D} \\ S_n.variables\_post & \text{otherwise} \end{cases}$

Table 4.2: Attribute grammar rules for handling variables at AST-node block B. Each block can have multiple statements S. $\mathfrak{D}$ is the set containing the node types return, continue and break.

$A \rightarrow E_{left}, E_{right}$	$A.variables = \text{clone}(A.variables\_pre)$ $A.variables\_post = A.variables[E_{left}.id \mapsto \mathsf{Assignment}(E_{right}.cfg)]$

Table 4.3: Attribute grammar rules for handling variables at AST-node assignment A. Each assignment has a left-hand side expression E_{left} and a right-hand side expression E_{right}. In this example we assume E_{left} to be of type identifier.

in Table 4.2.

In Table 4.3 we show the formal grammar for the attributes `variables` and `variables_post` of an assignment A of a local variable. We use $m[k]$ to denote the value stored at key k of a mapping m and $m[k \mapsto v]$ to denote a new mapping that is identical to m except for value v at key k. As described in Section 3.2 we use the current Assignment object as value for local variables. The semantic rule $A.variables\_post = A.variables[E_{left}.id \mapsto \mathsf{Assignment}(E_{right}.cfg)]$ shows how this is done in our IR. In an assignment we will create a new IR Assignment object containing a reference to the IR representation of the right hand side ($E_{right}.cfg$).

Variables can be shadowed which is why the name of the variable is unsuitable to use as definitive identifier. To keep track of scoping of variables we can use information provided in the AST by the Solidity compiler. Each identifier node holds the ID of the referenced declaration. Since each identifier referencing the same declaration actually accessed the same variable, we use this ID as key to the current assignment of that variable. In the semantic rule discussed above, the access to the ID is seen in the expression $E_{left}.id$.

We have shown in this chapter how we have established the IR of a smart contract using our evaluation framework. We will now show how to perform symbolic execution to prove post-conditions on the code.

Chapter 5

Case study: Symbolic execution using IRSRI

To test our IR we used IRSRI to perform symbolic execution of Solidity code. We will introduce and explain symbolic execution based on an example contract shown in Listing 5.1. The contract consists of two functions, `abs` and `neg`. The function `neg(a)` returns $neg(a) = -a$ and is used in the second function `abs(a)` which returns the absolute value $abs(a) = |a|$. The CFG returned by IRSRI is shown in Figure 5.1.

We will start by giving a short introduction to symbolic execution and the general idea behind it. The introduction will be followed by in-depth explanations of the path constraint (PC) and the symbolic state. Finally we will show how to use symbolic execution to prove post-conditions over code.

```
contract Foo {
    function abs(int a) public returns(int) {
        if (a < 0) {
            a = neg(a);
        }
        return a;
    }

    function neg(int a) private returns(int) {
        return -a;
    }
}
```

Listing 5.1: Implementation of absolute function in Solidity

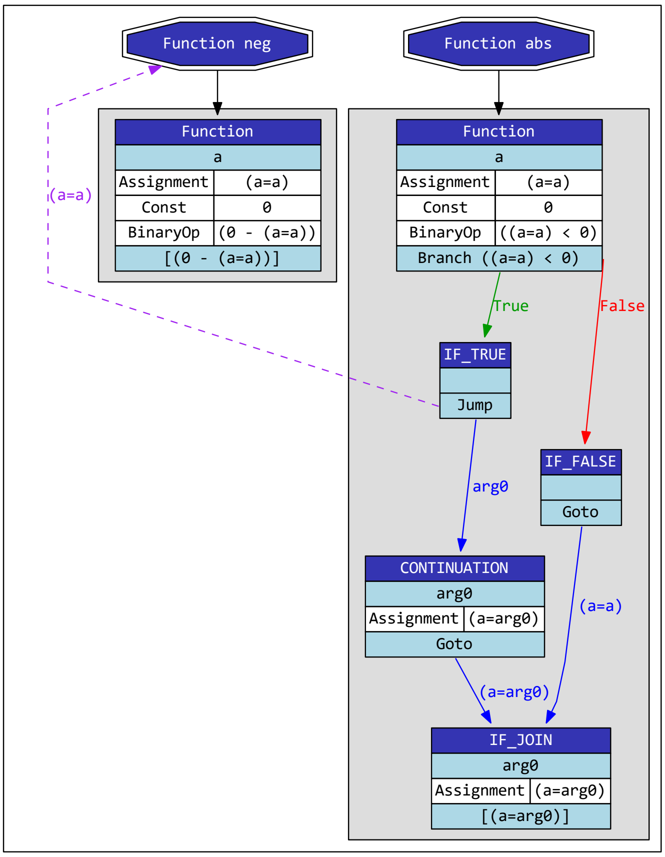


Figure 5.1: The CFG of the contract from Listing 5.1 returned by IRSRI

5.1 A Brief Overview of Symbolic Execution

Introduced by King in 1976 [30], symbolic execution in its essence consists of executing a program while using symbolic values as inputs instead of concrete ones. Normal program execution takes concrete values as input and explores a single execution path returning a certain output. By contrast, symbolic execution can explore many different execution paths. This allows us to analyse code in more detail than with usual test-case based analysis. As in regular program execution, we need to keep track of the current state of the symbolic execution.

For our example contract mentioned above, we are able to prove that $\forall a \in \mathbb{R}.\, abs(a) \geq 0$ using symbolic execution. To do so, we will symbolically execute the program and provide it with a symbolic input argument $\mathfrak{a}$.

The algorithm to symbolically execute a program and check a post-condition is shown in Algorithm 1.

Algorithm 1: Symbolic Execution

Input: entryBlock, args, acct, env, post-condition

```
foreach $argkey, arg$ in zip($entryBlock.args, args$) do
    initStore [argkey] $\leftarrow$ arg;
end
initFrame $\leftarrow$ [entryBlock, initStore ];
initStack.push($initFrame$);
pc = True;
initState $\leftarrow$ [pc, acct, env, stack];
availableStates.add($initState$);
while $\exists s \in availableStates$ do
    newStates, newFinalStates $\leftarrow$ executeBasicBlock($s$);
    availableStates.add (newStates);
    foreach $fs$ in newFinalStates do
        check($fs$, $post\text{-}condition$);
    end
end
```

After this short introduction to symbolic execution we will explore the symbolic state.

5.2 Symbolic State

Most imperative languages are stateful. When executing a program written in a stateful language the executing environment needs to keep track of the program state. symbolic execution is no exception although the state will look different. Our symbolic state is defined as

$$state := \begin{cases} path\ constraint, \\ account, \\ environment, \\ stack \end{cases}$$

First is the PC that accumulates constraint along a walked execution path. Next are the account and the environment that hold information about a contract and a transaction. These are exclusive to the analysis of smart contracts. Finally our state contains a stack containing the symbolic store.

We will now explore each of the components in more detail.

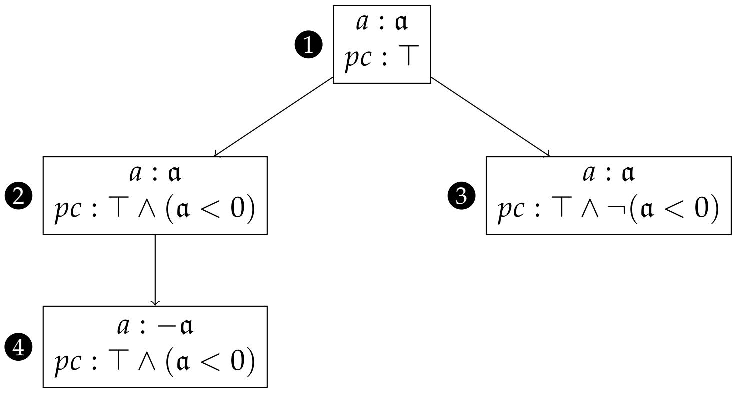

❶
$a : \mathfrak{a}$
$pc : \top$
❷
$a : \mathfrak{a}$
$pc : \top \wedge (\mathfrak{a} < 0)$
❸
$a : \mathfrak{a}$
$pc : \top \wedge \neg(\mathfrak{a} < 0)$
❹
$a : -\mathfrak{a}$
$pc : \top \wedge (\mathfrak{a} < 0)$

Figure 5.2: The symbolic tree of the contract abs function with symbolic input $\mathfrak{a}$.

5.2.1 Path Constraint

The PC contains information about the path previously traversed to arrive at the current state. We start with an unconstrained PC $pc_{start} = \top$. Each time the symbolic execution engine encounters a branch it splits into two distinct states, each associated to one of the branches. The PC of each new state will be the old PC intersected with some information about the branch. In IRSRI each branching has an associated condition. Thus the PCs of the branches will be $pc_{true} = pc \wedge condition$ for the path in which the condition is true and $pc_{false} = pc \wedge \neg condition$ for the path in which the condition is false.

In our code example in Listing 5.1 we encounter a branching statement in line 3. The statement `if (a < 0)` splits the execution path in two. We show the symbolic tree of the execution in Figure 5.1. We start at ❶, a with a value of $\mathfrak{a}$ and $pc = \top$. After the branch we have two states: ❷ and ❸. As one can see, both PCs have accumulated the corresponding constraint.

After having seen how the PC works we will now turn our attention to two special concepts in the scope of smart contracts: the Account and the Environment.

5.2.2 Account and Environment

As we have seen in our introduction to Ethereum, smart contracts are one of two types of accounts in Ethereum and are controlled by their contract code. During execution, a smart contract can access the address and the balance of the contract account. Additionally each contract has a storage where state variables and mappings are stored. The address, the balance and the storage build the account state.

In addition to the account state that persists beyond a transaction, there are

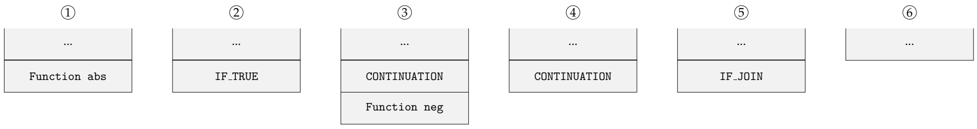


Figure 5.3: Basic blocks in symbolic stack during the execution of the code in Listing 5.1

also properties pertaining to single transactions. These include the sender of the transaction, the recipient, the value in Wei to be transferred to the recipient, the gas price and the gas limit. These make up the environment state.

A symbolic engine will need to create the account state together with the deployment of the contract as well as generating an environment state for each transaction.

5.2.3 Stack, Frames and the Symbolic Store

We proceed to the last component of the symbolic state, the stack. Each stack frame consists of a symbolic store, mapping an IR node to a z3 object, and a `Block`, the next basic block to be executed in that state

$$frame := \begin{cases} symbolic\ store, \\ Block \end{cases}$$

We will first describe the general use of the stack before going into more detail about the symbolic store.

The Stack Grows and Shrinks

Similarly to regular program execution we use a call stack to store information about the execution flow. The head of the stack points to the currently executed `Block` and its symbolic store. Whenever we transfer to another function we change the `Block` in the current frame to the continuation `Block`. We then copy the symbolic store and associate it to the jump destination in a new frame. Finally the new frame is pushed on the stack.

In Figure 5.3 we show the state of the symbolic stack during the execution of the abs function when the condition is true. When starting the execution we interpret the entry block `Function abs` which is the `Block` on top of the stack ①. After the transfer the stack looks like ②. Because we encounter a `Jump` the symbolic execution engine changes the block in the current frame to the continuation before pushing the new frame containing the destination ③. A `Return` will trigger the engine to pop the current frame from the stack and store the return values at the arguments in the symbolic store of the new

Solidity Type	z3 Object
`address`	`Int`
`address payable`	`Int`
`bool`	`Bool`
`string`	`String`
`mapping`	`Array`
`array`	`Array`
`uint`	`Int`
`uintN`	`BitVec` [N bits]
`int`	`Int`
`intN`	`BitVec` [N bits]
`byte`	`BitVec` [8 bits]
`bytesN`	`BitVec` [$8N$ bits]
`bytes`	`Array`

Table 5.1: Solidity to z3 type mapping

head of stack ④. After the `CONTINUATION` the current `Block` in the frame will be replaced by the next `Block`, `IF_JOIN` ⑤. Finally the `Return` will trigger the engine to pop the last frame before finishing the execution ⑥.

Having now seen the general principle of the stack we want to understand the symbolic store in a bit more detail.

Symbolic Store

The symbolic store maps a node from the IR to its corresponding symbolic representation.

We use two different types of mappings. The first is used in the account state as storage of state variables. It uses the variable name as key, since state variable names are unique and cannot be shadowed.

The second is stored on the execution stack. It takes an `Expression` as key and has the corresponding symbolic expression as value.

Since we use the z3 satisfiability modulo theories (SMT) solver, any IRSRI object will be mapped to a corresponding z3 expression. An important consideration when translating from IRSRI to z3 are data types. For each Solidity type we need to find a corresponding z3 type. Our mapping is shown in Table 5.1. Some of the mappings are straightforward, e.g. `bool` → `Bool`. However certain Solidity types could be represented by several different types in z3. `uint` and `int` in Solidity are syntactic sugar for `uint256` and int256 and are bit vectors of size 256 internally. We did however choose to represent uint and int as Int since z3 performs much worse when using BitVec with quantifiers[1].

Once the basic objects are mapped to z3 we can start building more complicated expressions. Due to our use of three-address code (3AC) in IRSRI we can simply go through each Statement of a Block, evaluate the Expression and store it in the symbolic store using the node as key. Whenever the Expression is used we retrieve the current z3 expression from the symbolic store using the reference as key.

There are several important details to mention. The first concerns the scope of the symbolic store; it reaches over a single basic block. It is only when a Jump or Call is encountered that we create a new symbolic store for the new stack frame. The store is then initialised with the values of the block arguments. Whenever a Return is encountered, the current frame is popped and values of the continuation arguments are set in the symbolic store of the new head.

The second detail concerns Branch instances. At each Branch instance the symbolic execution engine must create two new symbolic stores such that each path has access to the previous state but changes in one of the paths do not influence the symbolic store of the other path.

This concludes the section about the details of our case study. We will end this chapter by showing how we can use symbolic execution to check if a post-condition is valid.

5.3 Post-Condition Checking with Symbolic Execution

When trying to prove that a certain Post-Condition $\mathcal{A}$ holds for a program independently of input values we are trying to show that $\mathcal{A}$ holds for each final state of the program assuming that execution starts from an arbitrary state. Let $\mathcal{S}$ be the set of final states of a program and pc_s be the PC of a state s, then we want to show

$$\forall s \in \mathcal{S}.\, pc_s \implies \mathcal{A}$$

In general, SMT-solvers solve existentially quantified formulas. Therefore we transform the formula to the negation of an existential formula, and then try to solve that existential formula

$$\neg\exists s \in \mathcal{S}.\, pc_s \land \neg\mathcal{A}$$

[1] It is still possible to test for over- and underflows by adding assertions such as $Int < 2^{256}$. If we can find an interpretation where the assertion fails it is a possible source of overflows.

If we are unable to find an interpretation that satisfies $pc_s \land \neg\mathcal{A}$ we have shown that $pc_s \implies \mathcal{A}$ holds and, if we have tested all possible paths through a program, $\mathcal{A}$ holds for the program.

Returning to our example function in Listing 5.1, we are trying to prove the post-condition $abs(a) \geq 0$ for any a. The two final states of the program are ❸ and ❹ in Figure 5.2. We will now go through both states and try to find an interpretation satisfying $pc \land \neg(abs(\mathfrak{a}) \geq 0)$.

Let $abs(\mathfrak{a}) = \mathfrak{a}$ and $pc = \top \land \neg(\mathfrak{a} < 0)$. Then

$$\begin{aligned} & \top \land \neg(\mathfrak{a} < 0) \land \neg(\mathfrak{a} \geq 0) \\ \implies & (\mathfrak{a} \geq 0) \land (\mathfrak{a} < 0) \\ \implies & \bot \end{aligned}$$

For the second final state let $abs(\mathfrak{a}) = -\mathfrak{a}$ and $pc = \top \land (\mathfrak{a} < 0)$. Then

$$\begin{aligned} & \top \land (\mathfrak{a} < 0) \land \neg(-\mathfrak{a} \geq 0) \\ \implies & (\mathfrak{a} < 0) \land (-\mathfrak{a} < 0) \\ \implies & \bot \end{aligned}$$

Since $pc \land \neg(abs(\mathfrak{a}) \geq 0)$ leads *ad absurdum* for each state we have shown that our post-condition $abs(a) \geq 0$ is valid.

5.4 Conclusion

We have found the general design of our IR to be powerful enough to represent complex contracts whilst remaining simple enough to perform different analyses with the IR as basis. The IR contains much high-level information important for any type of analysis. The use of arguments and transfers allows to handle features like multiple return values without any additional effort.

Another major aspect of our IR that has proven to be a good choice concerns the use of IR objects as temporary variables. Both static single assignment (SSA) and 3AC require the creation of temporary variables to hold a new binding in the case of SSA and a temporary result in 3AC. By using the same type of objects, IR nodes, as temporary variables in both cases allows us to reuse them as key to the corresponding symbolic representation without additional effort. This allows us to have a simpler and cleaner translator since we get a one-to-one mapping between the IR and the symbolic representation.

In general our IR has lent itself nicely as basis for symbolic execution and static analysis (see [29]). By using 3AC and having similar structures we were able to translate most nodes to z3 with very little effort. Once a mapping from Solidity to z3 types was defined, the remaining translation was quite trivial.

This concludes our case study. Next to introducing several concepts of symbolic execution as the symbolic state, the path constraint and the symbolic store, we have shown how to perform symbolic execution of IRSRI using z3 as SMT solver. This brings us to the last part of this report, where we will give an outlook of future steps in this domain.

Chapter 6

Outlook

There are several conceivable future steps. The first ones concern the translation from Solidity to the IR. Several features of Solidity including modifiers, inline assembly and libraries are not fully functional yet. In addition, testing the correctness of a translation is currently done by comparing the original code with the produced IR manually. The development of automated means for testing the correctness of a translation could greatly improve future development speed.

Other steps would improve the reach of analysis tools built on top of IRSRI. By developing the grammatical rules for other high-level languages, our attribute grammar evaluation framework could translate code written in those languages to IRSRI. This would allow to use any analysis tool based on IRSRI to be used on code written in those languages. Additionally IRSRI could be integrated with further tools to enhance the analysis possibilities. One such example is Securify[1] which will start using IRSRI as basis for its analysis tools.

We believe that an IR based on a combination of static single assignment and continuation-passing style is a promising basis for many kinds of applications.

[1] https://securify.ch/

Bibliography

[1] M. d. Castillo. (Jun. 17, 2016). The DAO attacked: Code issue leads to $60 million ether theft, CoinDesk, [Online]. Available: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft (visited on 04/28/2019).

[2] W. Zhao. (Jul. 19, 2017). $30 million: Ether reported stolen due to parity wallet breach, CoinDesk, [Online]. Available: https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach (visited on 04/28/2019).

[3] P. Technologies. (Nov. 15, 2017). A postmortem on the parity multi-sig library self-destruct, Blockchain Infrastructure for the Decentralised Web, [Online]. Available: https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/ (visited on 04/28/2019).

[4] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bnzli, and M. Vechev, 'Securify: Practical security analysis of smart contracts', in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*, Toronto, Canada: ACM Press, 2018, pp. 67–82, ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243780.

[5] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, 'Finding the greedy, prodigal, and suicidal contracts at scale', *arXiv [cs]*, Feb. 16, 2018. arXiv: 1802.06038.

[6] R. Stortz. (Jun. 16, 2018). Rattle - an ethereum EVM binary analysis framework, Trail of Bits, [Online]. Available: https://www.trailofbits.com/presentations/rattle/ (visited on 04/28/2019).

[7] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, 'Vandal: A scalable security analysis framework for smart contracts', *arXiv [cs]*, Sep. 11, 2018. arXiv: 1809.03981.

[8] I. Grishchenko, M. Maffei, and C. Schneidewind, ‘EtherTrust: Sound static analysis of ethereum bytecode’, Jul. 2018.

[9] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, ‘Making smart contracts smarter’, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS’16*, Vienna, Austria: ACM Press, 2016, pp. 254–269, isbn: 978-1-4503-4139-4. doi: `10.1145/2976749.2978309`.

[10] Trail of Bites. Manticore: Symbolic execution for humans, [Online]. Available: `https://github.com/trailofbits/manticore` (visited on 04/28/2019).

[11] B. Mueller, ‘Smashing ethereum smart contracts for fun and real profit’, *The 9th annual HITB Security Conference*, Apr. 12, 2018.

[12] J. Feist, G. Greico, and A. Groce, ‘Slither: A static analysis framework for smart contracts’,

[13] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, ‘SmartCheck: Static analysis of ethereum smart contracts’, in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2018, pp. 9–16.

[14] L. Alt and C. Reitwiessner, ‘SMT-based verification of solidity smart contracts’, in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, T. Margaria and B. Steffen, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2018, pp. 376–388, isbn: 978-3-030-03427-6.

[15] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, ‘EthIR: A framework for high-level analysis of ethereum bytecode’, *arXiv [cs]*, 2018. arXiv: `1805.07208`.

[16] I. Sergey, A. Kumar, and A. Hobor, ‘Scilla: A smart contract intermediate-level LAnguage’, *arXiv [cs]*, Jan. 2, 2018. arXiv: `1801.00687`.

[17] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, ‘Gigahorse: Thorough, declarative decompilation of smart contracts’,

[18] Protofire. Solhint, [Online]. Available: `https://github.com/protofire/solhint` (visited on 04/28/2019).

[19] N. Szabo, ‘Formalizing and securing relationships on public networks’, *First Monday*, vol. 2, no. 9, Sep. 1, 1997, issn: 13960466. doi: `10.5210/fm.v2i9.548`.

[20] S. Nakamoto, ‘Bitcoin: A peer-to-peer electronic cash system’, Oct. 31, 2008.

[21] V. Buterin, ‘A NEXT GENERATION SMART CONTRACT & DECENTRALIZED APPLICATION PLATFORM’, 2013.

[22] F. E. Allen and F. E. Allen, 'Control flow analysis', *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, Jul. 27, 1970, ISSN: 0362-1340. DOI: 10.1145/800028.808479.

[23] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, 'Global value numbers and redundant computations', presented at the Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, Jan. 13, 1988, pp. 12–27, ISBN: 978-0-89791-252-5. DOI: 10.1145/73560.73562.

[24] A. W. Appel, 'SSA is functional programming', *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, Apr. 1, 1998, ISSN: 03621340. DOI: 10.1145/278283.278285.

[25] tephen Weeks, *Cps vs ssa*, Jan. 10, 2003. [Online]. Available: http://mlton.org/pipermail/mlton/2003-January/023054.html (visited on 04/30/2019).

[26] R. A. Kelsey and R. A. Kelsey, 'A correspondence between continuation passing style and static single assignment form', *ACM SIGPLAN Notices*, vol. 30, no. 3, pp. 13–22, Mar. 1, 1995, ISSN: 0362-1340. DOI: 10.1145/202529.202532.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0-201-63361-2.

[28] D. E. Knuth, 'Semantics of context-free languages', *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, Jun. 1, 1968, ISSN: 1433-0490. DOI: 10.1007/BF01692511.

[29] L. Frobeen, 'Static analysis using IRSRI', Master's thesis, ETH Zurich, Jun. 2, 2019.

[30] J. C. King, 'Symbolic execution and program testing', *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

IRSRI
An Intermediate Representation
for Smart Contracts

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s): Vogel

First name(s): Frédéric Henri

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date: Zurich, 1 May 2019

Signature(s): FVogel

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.