1. **INTRODUCTION**

xxxxxx

1. **YYY**
2. **REWRITE RULES**

We start by formalizing solidity, a subset of the complete language which we use to describe our rewriting rules.

* 1. **Language: Solidity**

For ease of exposition, we assume that a solidity program is an expression (i.e., unlike JavaScript we do not distinguish between statements and expressions). Solidity expressions include:

* ***basic constants***of the form JavaScript that represent integers, strings etc.,
* ***field reads*** *of* the from *e1*[*e2*], where *e1*is an expression that evaluates to the object whose field is being read, and *e2* is an expression that evaluates to the name of the field being read.
* ***Unary operations*** of the form *e1* op that show primitive operations like increment, decrement operation.
* ***binary operations***of the form *e1* op *e2* that include primitive operations like integer addition, string concatenation etc.
* ***object literals*** of the form {*f1*:*e1*…} that map a set of fields *f1*… to a set of objects represented by *e1*… respectively.
* ***variable assignments***of the form *x* = *e*; the assignment updates *x* and evaluates to the object that *e* evaluate to.
* ***field assignments***of the form *e1*[*e2*] = *e3,* where *e1* evaluates to the object whose field is updated, *e2* evaluates to (a string naming) the field being written, and *e3* is the expression whose value the field is updated with; field-assignments evaluate to the object that *e3* evaluates to.
* ***branches***of the form if *e1*, *e2*, *e3*: a branch expression evaluates to the trivial null object. *e1 stands for condition in if statement; e2 stands for the code when e1 is true; and e3 represents the code when e1 is false.*
* ***functions***of the form fun(*x1*…){*e*} where *x1*… are the formals of the function and *e* the function’s body (the function returns the value of *e*); in our encoding, methods are functions with a “*this*” parameter, that are bound to the fields of objects.
* ***function calls***of the form *e*(*e1*…) where e evaluates to the callee and *e1*… to the arguments; we encode method calls as function calls made through a field, and for which the target object is passed as the first parameter (for example *x.f*(*x*, …)).
* ***loop (for, while)***of the form **loop**(*i* = *first* **to** *last*) **do** *e* where “*last – first*” represents the execution count of this loop; and *e* the loop’s body.
* ***The start and end position of basic block:*** in addition, we add two expressions to present the position of basic block.There are ***“****block\_st*” and “*block\_ed*” which are representing the start and end point of basic block respectively. Our main approach is to ensure execution flow by tracking control flow of contracts at runtime.
  1. **ERC20 interface in Solidity**

ERC20 is a standard interface for tokens. The standard provides basic functionality to transfer tokes, as well as allow tokens to be approved so they can be spent by another on-chain third party. The reason to suggest it is to allows any tokens on Ethereum to be re-used by other applications: from wallets to decentralized exchanges. Solidity 0.4.17 (above) supports ERC20 functions [[1](https://eips.ethereum.org/EIPS/eip-20)].

* 1. **Rewriting Algorithm**

**Table** 1. summarizes our rewriting procedure RW (*x* op *y*), which takes as input smart contracts made by Solidity. The table is given as a series of rules of the form:

RW (*x* op *y*) ≅ *e`*

where each rule describes how an expression that matches *e* is rewritten to the expression *e*`. We use *italics* to denote meta variables that range over expressions and typewriter to denote our guard functions. At a high level, our rewriting has two goals. First, to ensure the execution flow integrity that complies smart contract developer’s intent. This makes smart contract safe for reentrancy attack. Second, to ensure the specific value range about an object. Ethereum specifies that the block gas limit, transactions can only consume a certain amount of gas.

if there is a loop that does not have a fixed number of iterations, the loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. To avoid the state, therefore, our tool monitors that the iteration does not stall the service at runtime by leveraging code instrumentation .

Next, we describe how the rewriting procedure achieves these goals by describing how it tracks and checks different kinds of flows.

**Assignments:** xxx

**Example**

**Branches:** yyy

**Example**

**Function Call:** zzz

**Example:**

* 1. **Policy enforcement**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Rewrite Procedure** | **No.** | **Rewrite Procedure** |
| **1** | **RW** (*e*; at “block\_st”) 🡪  Check(cur\_block\_no);  *e*; | **6** | **RW** (**loopfor** (*x* = 0 **to** *y*) *e*) 🡪  loopfor (*x* = 0 to *y*)  {  if (checkGasLimit(*y* - 0, cur\_block\_no))  *e*;  else revert();  } |
| **2** | **RW** (*e1*; at “block\_ed”) 🡪  Record(cur\_block\_no);  *e1* can be any expression except for break, return, func\_call(). | **6-1** | **RW** (**loop**do-while (*x* > *y*) *e*) 🡪  loopdo-while {  *e;*  if (!checkGasLimit(*y* - x, cur\_block\_no) revert();  } |
| **3** | **RW** (*e2*; at “block\_ed”) 🡪  Record(cur\_block\_no);  *e2*;    *e2* can be break, return, or func\_call(). | **8** | **RW** (if *e1, e2, e3*) 🡪 where *e2 is x2-1* op *x2-2*; and *e3* is *break*  if (*e1 is* ***TRUE****)* {  if (checkIntOp(*x2-1*, *x2-2*))  *e2 =* Intop(*x2-1*, *x2-2*);  else revert();  }  else //***FALSE*** part {  Record(cur\_block\_no); *e3;*  } |
| **4** | **RW** (*a* = *x1*[*y1*]op *x2*) 🡪  if (checkIntop(*x1*[*y1*], *x2*))  a = Intop(*x1*[*y1*]*, x2*);  else revert(); | **9** | **RW** (*z* = func\_call(*x*…) op *y*)  *Temp* = func\_call(*x*…);  if (checkIntOp(*Temp*,*y*))  *z* = Intop(*Temp*, *y*);  else revert(); |
| **4-1** | **RW** (*a* = *x1* op *x2* op *x3*) 🡪  if (checkIntop(*x1,x2*))  if (checkIntop(Intop(*x1,x2*),*x3*))  *a* = Intop(Intop(*x1,x2*),*x3*);  else revert();  else revert(); | **9-1** | **RW** (*z* = *x*.func\_call(*y*.fun()) op *w*)  *Temp* = *x*.func\_call(*y*.fun());  if (checkIntOp(*Temp*, *w*))  *z* = Intop(*Temp*, *w*);  else revert();  -----------------------------------------------------------------  The rule 9 and 9-1 separate the assignment statement to external fun() and some operations. Moreover, in this case, we should apply the rules according to an order of priority.  So, we need to clarify the priority to apply our rewrite rule. |
| **5** | **RW** (*x* op) 🡪  if (checkXcrementOp(*x*)) *x--*;  else revert(); |

**Example Legends**

**------------------------**

\_ 🡪 Any expression

Uop 🡪 Unary decrement, and increment operations (++, --)

Bop 🡪 Binary Arithmetic operations (+,-,%,/)

Comp 🡪 Comparison operators (e.g., >, <, => …)

Call 🡪 Any external call

**Protection functions**

**----------------------------**

Check(Current Block Number) 🡪 Check the execution flow integrity.

Record(Current Block Number) 🡪 Save current block number that is will be executed right next.

CheckGasLimit(Loop Count, Current Block Number) 🡪 calculate the approximate gas price for this basic block and check the gas limitation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Basic rules – to ensure execution flow integrity** | | | | |
| **Block Start** | | **Expression** | **Rewritten code** | **Comment** |
| \_; | Check(cur\_block\_no);  \_; | Inject Check(); before the user code. |
| **Block End** | | **Expression** | **Rewritten code** | **Comment** |
| \_; (except for break, return, call) | \_;  Record(cur\_block\_no); | Inject Record(); after the user code. |
| break or return or call; | Record(cur\_block\_no);  break or return or call; | Inject Record(); before the user code where the end of code is break or return or call(). |
| **Extended rules – to ensure value range and integer overflow** | | | | |
| - | **Statement** | | **Rewritten code** | **Comment** |
| a = x +(Bop) y; | | If (CheckIntAdd (x,y))  a = IntAdd(x,y); // SafeMath  else revert(); | This rule supports other arithmetic operations (+,/, %, and so on). |
| a = x +(Bop) y +(Bop) z; | | if (CheckIntAdd (x,y))  if (CheckIntAdd(IntAdd(x,y),z))  a = IntAdd(IntAdd(x,y),z);  else revert();  else revert(); | How to handle this case? |
| a--(Uop); | | If (CheckDecrementOp(a)) a--(Uop);  else revert(); | This rule also supports prefix increment operation(++). |
| for (int x = 0; x < y; x++)  {  \_;  } | | for (int x = 0; x <(comp) y; x++(Uop))  {  if (checkGasLimit(y,cur\_block\_no))  \_;  else revert();  } | This rule checks that the approximate gas limit to execute this loop is safe or not. For this our tool pre-calculates the gas fee for each basic block. |
| do {  \_;  } while (x > y) | | do {  if (checkGasLimit(y,cur\_block\_no))  \_;  else revert();  } while (x >(comp) y) | This rule is similar to a rule for “for” loop. |
| if (isCheck)  a = x[5] +(Bop) y;  else  break; | | if (isCheck)  {  if (checkIntAdd(x[5],y))  a = IntAdd(x[5],y);  else revert();  }  else  {  Record(cur\_block\_no);  break;  } |  |
| a = x.getBalance()(call) +(Bop) \_; | | tmp = x.call();  if (CheckIntAdd (tmp, \_))  {  a = IntAdd(x, \_);  }  else revert(); | This rule separates the assignment statement to external call() and some operations.  Moreover, in this case, we should apply the rules according to an order of priority. |
| a = x.calc(call)(y.getBal()(call)) +(Bop) z; | | tmp = x.call(y.call());  if (CheckIntAdd (tmp, z))  a = IntAdd(tmp, z);  else revert(); |  |