**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(); |  |

1. **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.
* ***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 filed is updated with; field-assignments evaluate to the object that e3 evaluates to.
* ***branches***of the form if e1 e*2* e3: a branch expression evaluates to the trivial null object.
* ***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; e the loop’s body.