Informações de Debug do contrato `SpyErcBridge`

Esses são os argumentos.

```
Python
```

Código - Observações

- Locais marcados com `# f`:
  Indicam que, durante a execução do contrato analisado, a condição **não foi satisfeita** e, portanto, **não entrou** na condicional associada.

- Locais marcados com `# v`:
  Indicam que a condição **foi satisfeita** e, durante a execução, o contrato **entrou** na condicional associada.

Essas marcações ajudam a identificar o comportamento do contrato em diferentes cenários de execução.

```python
class SymExecWrapper:
    """Wrapper class for the LASER Symbolic virtual machine.

    Symbolically executes the code and does a bit of pre-analysis for
    convenience.
    """

    def __init__(
        self,
        contract,
        address: Union[int, str, BitVec],
        strategy: str,
        dynloader=None,
        max_depth: int = 22,
        execution_timeout: Optional[int] = None,
        loop_bound: int = 3,
        create_timeout: Optional[int] = None,
        transaction_count: int = 2,
        modules: Optional[List[str]] = None,
```

```
        compulsory_statespace: bool = True,
        disable_dependency_pruning: bool = False,
        run_analysis_modules: bool = True,
        custom_modules_directory: str = "",
    ):
        """

        :param contract: Contract to symbolically execute
        :param address: Address of the contract to symbolically execute
        :param strategy: Execution strategy to use (bfs, dfs, etc)
        :param dynloader: Dynamic Loader
        :param max_depth: Max analysis depth
        :param execution_timeout: Timeout for the entire analysis
        :param create_timeout: Timeout for the creation transaction
        :param transaction_count: Number of transactions to symbolically
execute
        :param modules: Analysis modules to run during analysis
        :param compulsory_statespace: Boolean indicating whether or not the
statespace should be saved
        :param iprof: Instruction Profiler
        :param disable_dependency_pruning: Boolean indicating whether
dependency pruning should be disabled
        :param run_analysis_modules: Boolean indicating whether analysis
modules should be executed
        :param enable_coverage_strategy: Boolean indicating whether the
coverage strategy should be enabled
        :param custom_modules_directory: The directory to read custom
analysis modules from
        """
        if isinstance(address, str): # v
            address = symbol_factory.BitVecVal(int(address, 16), 256)
        if isinstance(address, int): # f
            address = symbol_factory.BitVecVal(address, 256)
        beam_width = None
        if strategy == "dfs": # f
            s_strategy: Type[BasicSearchStrategy] = DepthFirstSearchStrategy
        elif strategy == "bfs": # v
            s_strategy = BreadthFirstSearchStrategy
        elif strategy == "naive-random":
            s_strategy = ReturnRandomNaivelyStrategy
        elif strategy == "weighted-random":
            s_strategy = ReturnWeightedRandomStrategy
        elif "beam-search: " in strategy:
            beam_width = int(strategy.split("beam-search: ")[1])
            s_strategy = BeamSearch
        elif "pending" in strategy:
            s_strategy = DelayConstraintStrategy
        else:
```

```
            raise ValueError("Invalid strategy argument supplied")

        if args.incremental_txs is False: # f
            tx_strategy = RfTxPrioritiser(contract)
        else:
            tx_strategy = None # v

        creator_account = Account( # v
            hex(ACTORS.creator.value), "", dynamic_loader=None,
contract_name=None
        )
        attacker_account = Account( # v
            hex(ACTORS.attacker.value), "", dynamic_loader=None,
contract_name=None
        )

        requires_statespace = ( # v
            compulsory_statespace
            or len(ModuleLoader().get_detection_modules(EntryPoint.POST,
modules)) > 0
        )
        if not contract.creation_code: # f
            self.accounts = {hex(ACTORS.attacker.value): attacker_account}
        else:
            self.accounts = { # v
                hex(ACTORS.creator.value): creator_account,
                hex(ACTORS.attacker.value): attacker_account,
            }

        self.laser = svm.LaserEVM( # v
            dynamic_loader=dynloader,
            max_depth=max_depth,
            execution_timeout=execution_timeout,
            strategy=s_strategy,
            create_timeout=create_timeout,
            transaction_count=transaction_count,
            requires_statespace=requires_statespace,
            beam_width=beam_width,
            tx_strategy=tx_strategy,
        )

        if loop_bound is not None: # v
            self.laser.extend_strategy(
                BoundedLoopsStrategy, loop_bound=loop_bound,
beam_width=beam_width
            )

        plugin_loader = LaserPluginLoader()
```

```
        plugin_loader.load(CoverageMetricsPluginBuilder())
        if args.enable_state_merge: # f
            plugin_loader.load(StateMergePluginBuilder())
        if not args.disable_coverage_strategy: # v
            plugin_loader.load(CoveragePluginBuilder())
        if not args.disable_mutation_pruner: # v
            plugin_loader.load(MutationPrunerBuilder())
        if not args.disable_iprof: # v
            plugin_loader.load(InstructionProfilerBuilder())
        if args.enable_summaries: # f
            plugin_loader.load(SymbolicSummaryPluginBuilder())

        plugin_loader.load(CallDepthLimitBuilder())
        plugin_loader.add_args(
            "call-depth-limit", call_depth_limit=args.call_depth_limit
        )

        if not disable_dependency_pruning: # v
            plugin_loader.load(DependencyPrunerBuilder())

        plugin_loader.instrument_virtual_machine(self.laser, None)

        world_state = WorldState()
        for account in self.accounts.values():
            world_state.put_account(account)

        if run_analysis_modules: # v
            analysis_modules = ModuleLoader().get_detection_modules(
                EntryPoint.CALLBACK, modules
            )
            self.laser.register_hooks(
                hook_type="pre",
                hook_dict=get_detection_module_hooks(analysis_modules,
hook_type="pre"),
            )
            self.laser.register_hooks(
                hook_type="post",
                hook_dict=get_detection_module_hooks(
                    analysis_modules, hook_type="post"
                ),
            )

        if isinstance(contract, SolidityContract) and create_timeout != 0: #
v
            self.laser.sym_exec(
                creation_code=contract.creation_code,
                contract_name=contract.name,
                world_state=world_state,
```

```
            )
        elif isinstance(contract, EVMContract) and contract.creation_code:
            self.laser.sym_exec(
                creation_code=contract.creation_code,
                contract_name=contract.name,
                world_state=world_state,
            )
        else:
            account = Account(
                address,
                contract.disassembly,
                dynamic_loader=dynloader,
                contract_name=contract.name,
                balances=world_state.balances,
                concrete_storage=(
                    True if (dynloader is not None and dynloader.active)
else False
                ),
            )  # concrete_storage can get overridden by global args

            if dynloader is not None:
                if isinstance(address, int):
                    try:
                        _balance = dynloader.read_balance(
                            "{0:#0{1}x}".format(address, 42)
                        )
                        account.set_balance(_balance)
                    except:
                        # Initial balance will be a symbolic variable
                        pass
                elif isinstance(address, str):
                    try:
                        _balance = dynloader.read_balance(address)
                        account.set_balance(_balance)
                    except:
                        # Initial balance will be a symbolic variable
                        pass
                elif isinstance(address, BitVec):
                    try:
                        _balance = dynloader.read_balance(
                            "{0:#0{1}x}".format(address.value, 42)
                        )
                        account.set_balance(_balance)
                    except:
                        # Initial balance will be a symbolic variable
                        pass

            world_state.put_account(account)
```

```
            self.laser.sym_exec(world_state=world_state,
target_address=address.value)

        if not requires_statespace: # v
            return

        self.nodes = self.laser.nodes
        self.edges = self.laser.edges

        # Parse calls to make them easily accessible

        self.calls: List[Call] = []

        for key in self.nodes:

            state_index = 0

            for state in self.nodes[key].states:

                instruction = state.get_current_instruction()

                op = instruction["opcode"]

                if op in ("CALL", "CALLCODE", "DELEGATECALL", "STATICCALL"):

                    stack = state.mstate.stack

                    if op in ("CALL", "CALLCODE"):
                        gas, to, value, meminstart, meminsz, _, _ = (
                            get_variable(stack[-1]),
                            get_variable(stack[-2]),
                            get_variable(stack[-3]),
                            get_variable(stack[-4]),
                            get_variable(stack[-5]),
                            get_variable(stack[-6]),
                            get_variable(stack[-7]),
                        )

                        if (
                            to.type == VarType.CONCRETE
                            and 0 < to.val <= PRECOMPILE_COUNT
                        ):
                            # ignore prebuilts
                            continue

                        if (
                            meminstart.type == VarType.CONCRETE
                            and meminsz.type == VarType.CONCRETE
```

```
                        ):
                            self.calls.append(
                                Call(
                                    self.nodes[key],
                                    state,
                                    state_index,
                                    op,
                                    to,
                                    gas,
                                    value,
                                    state.mstate.memory[
                                        meminstart.val : meminsz.val +
meminstart.val
                                    ],
                                )
                            )
                        else:
                            self.calls.append(
                                Call(
                                    self.nodes[key],
                                    state,
                                    state_index,
                                    op,
                                    to,
                                    gas,
                                    value,
                                )
                            )
                    else:
                        gas, to, meminstart, meminsz, _, _ = (
                            get_variable(stack[-1]),
                            get_variable(stack[-2]),
                            get_variable(stack[-3]),
                            get_variable(stack[-4]),
                            get_variable(stack[-5]),
                            get_variable(stack[-6]),
                        )

                        self.calls.append(
                            Call(self.nodes[key], state, state_index, op,
to, gas)
                        )

                state_index += 1

    @property
    def execution_info(self) -> List[ExecutionInfo]:
        return self.laser.execution_info
```