AliyunCTF2025 mba Writeup

server.py

```
#!/usr/bin/env python3

from lark import Lark, Transformer
from typing import Any, List, Tuple, Self
import z3, os

BITSET = {
  'x': [0, 0, 1, 1],
  'y': [0, 1, 0, 1],
}

Rule = r"""
?start: expr -> expression

?expr: coterm -> coefterm
    | expr "+" coterm -> add
    | expr "-" coterm -> sub

?coterm: term -> term
    | integer "*" term -> mul
    | integer -> const

?term: "(" term ")"
    | "~" "(" term ")" -> bnot_term
    | factor "&" factor -> band
    | factor "|" factor -> bor
    | factor "^" factor -> bxor
    | factor -> single

?factor: "x" -> x
    | "y" -> y
    | "~" factor -> bnot

?integer: /\d{1,8}/

%import common.WS
%ignore WS
"""

P = Lark(Rule, parser='lalr', start='start')
PT = Lark(Rule, parser='lalr', start='term')

class MBATransformer(Transformer):
  def expression(self, args):
    return MBAExpr(args[0])

  def coefterm(self, args):
    return [args[0]]

  def add(self, args):
```

```
    return args[0] + [args[1]]

  def sub(self, args):
    new_arg = (-args[1][0], args[1][1])
    return args[0] + [new_arg]

  def term(self, args):
    return (1, args[0])

  def mul(self, args):
    num = int(args[0])
    return (num, args[1])

  def const(self, args):
    num = int(args[0])
    return (-num, BoolFunction('&', ['x', '~x'], True))

  def bnot_term(self, args):
    return args[0].invert()

  def band(self, args):
    return BoolFunction('&', args)

  def bor(self, args):
    return BoolFunction('|', args)

  def bxor(self, args):
    return BoolFunction('^', args)

  def single(self, args):
    return BoolFunction(None, args)

  def x(self, args):
    return 'x'

  def y(self, args):
    return 'y'

  def bnot(self, args):
    if args[0] == 'x' or args[0] == 'y':
      return '~' + args[0]
    assert args[0][0] == '~', "Invalid expression"
    return args[0][1]   # double negation

  def integer(self, args):
    return args[0]

boolean = lambda x: 1 if x else 0
def _handle_uop(op: str, a: List[int] | int) -> List[int] | int:
  if op != '~':
    raise ValueError("Invalid unary operator")
  if isinstance(a, int):
    return boolean(1 if a == 0 else 0)
  return [boolean(1 if x == 0 else 0) for x in a]

def _get_bitvec(s: str, nbits: int) -> z3.BitVec:
  return z3.BitVec(s, nbits)
```

```
def _get_bitvecval(v: int, nbits: int) -> z3.BitVecVal:
  return z3.BitVecVal(v, nbits)

class BoolFunction(object):
  def __init__(self, op: str | None, args: List[str], inverted: bool = False):
    if op is not None:
      assert len(args) == 2, "Binary operator must have two arguments"
    else:
      assert len(args) == 1, "A bool function must have at least one argument"
      if inverted and args[0] == '~':   # double neg
        inverted = False
        args = [args[0][1]]
    self.op = op
    self.args = args
    self.inverted = inverted

  def __str__(self):
    if self.op is not None:
      s = "(" + self.op.join(self.args) + ")"
    else:
      s = self.args[0]

    if self.inverted:
      return f"~{s}"
    return s

  def __repr__(self):
    return str(self)

  def _get_arg_symbol(self, s: str) -> str:
    if s[0] != '~':
      return s
    return s[1]

  def _get_bitset(self, s: str) -> List[int]:
    if s[0] != '~':
      return BITSET[s]
    return _handle_uop('~', self._get_bitset(s[1]))

  def _eval_arg(self, s: str, x: int, y: int) -> int:
    bitset = {'x': x, 'y': y}
    if s[0] != '~':
      return bitset[s]
    return _handle_uop('~', bitset[s[1]])

  def invert(self) -> Self:
    return BoolFunction(self.op, self.args, not self.inverted)

  def _get_arg_z3expr(self, s: str, nbits: int) -> z3.BitVecRef:
    if s[0] != '~':
      return _get_bitvec(s, nbits)
    return ~(_get_bitvec(s[1], nbits))

  def to_z3expr(self, nbits: int) -> Any:
    if not self.op:
      arg_expr = self._get_arg_z3expr(self.args[0], nbits)
      if self.inverted:
        return ~arg_expr
```

```
      return arg_expr

    a = self._get_arg_z3expr(self.args[0], nbits)
    b = self._get_arg_z3expr(self.args[1], nbits)
    if self.op == '&':
      expr = a & b
    elif self.op == '|':
      expr = a | b
    elif self.op == '^':
      expr = a ^ b
    else:
      raise ValueError("Invalid operator")

    if self.inverted:
      return ~expr
    return expr

class MBAExpr(object):
  def __init__(self, coterms: List[Tuple[int, BoolFunction]]):
    self._coterms = coterms

  def __len__(self):
    return len(self._coterms)

  def __getitem__(self, i: int) -> Tuple[int, BoolFunction]:
    return self._coterms[i]

  def __setitem__(self, i: int, v: Tuple[int, BoolFunction] | BoolFunction):
    coef = self._coterms[i][0]
    if isinstance(v, BoolFunction):
      self._coterms[i] = (coef, v)
    else:
      self._coterms[i] = v

  def __str__(self):
    r = ""
    for c, t in self._coterms:
      if c < 0:
        r += f"-{abs(c)}*{t}"
      else:
        r += f"+{c}*{t}"
    return r

  def __repr__(self):
    return str(self)

  def to_z3expr(self, nbits: int) -> z3.BitVecRef:
    expr = 0
    for c, t in self._coterms:
      expr += _get_bitvecval(c, nbits) * t.to_z3expr(nbits)
    return expr

  @property
  def coterms(self) -> List[Tuple[int, BoolFunction]]:
    return self._coterms

T = MBATransformer()
def parse(expr: str) -> MBAExpr:
```

```
  return T.transform(P.parse(expr))

def parse_term(term: str) -> BoolFunction:
  return T.transform(PT.parse(term))

def check_expression(t: z3.Tactic, e: MBAExpr) -> bool:
  expr = e.to_z3expr(64)
  s = t.solver()
  s.add(expr != expr)

  s.set('timeout', 30000)   # 30 seconds
  r = s.check()
  if r == z3.unknown:
    print("Solver timed out")
    exit(1)
  return r == z3.unsat

def serve_challenge():
  FLAG = os.environ.get('FLAG', 'aliyunctf{this_is_a_test_flag}')

  expr = input("Please enter the expression: ")
  if len(expr) > 200:
    print("Expression is too long")
    exit(1)

  try:
    mba = parse(expr)
  except Exception as e:
    print("Could not parse the expression")
    exit(1)

  if len(mba.coterms) > 15:
    print("Too many terms")
    exit(1)

  t = z3.Then(
    z3.Tactic('mba'),
    z3.Tactic('simplify'),
    z3.Tactic('smt')
  )

  if check_expression(t, mba):
    print("It works!")
  else:
    print(f"Flag: {FLAG}")
  return

if __name__ == '__main__':
  serve_challenge()
```

new-tactic.patch

```
diff --git a/src/tactic/bv/CMakeLists.txt b/src/tactic/bv/CMakeLists.txt
index 9009e6fa5..72bd2cfa1 100644
--- a/src/tactic/bv/CMakeLists.txt
+++ b/src/tactic/bv/CMakeLists.txt
@@ -10,6 +10,7 @@ z3_add_component(bv_tactics
```

```
     bv_size_reduction_tactic.cpp
     dt2bv_tactic.cpp
     elim_small_bv_tactic.cpp
+    mba_tactic.cpp
   COMPONENT_DEPENDENCIES
     bit_blaster
     core_tactics
@@ -25,4 +26,5 @@ z3_add_component(bv_tactics
     dt2bv_tactic.h
     elim_small_bv_tactic.h
     max_bv_sharing_tactic.h
+    mba_tactic.h
 )
diff --git a/src/tactic/bv/mba_tactic.cpp b/src/tactic/bv/mba_tactic.cpp
new file mode 100644
index 000000000..f3796c1e7
--- /dev/null
+++ b/src/tactic/bv/mba_tactic.cpp
@@ -0,0 +1,381 @@
+#include "tactic/tactic.h"
+#include "tactic/tactical.h"
+#include "tactic/bv/mba_tactic.h"
+#include "ast/bv_decl_plugin.h"
+
+#include <tuple>
+#include <vector>
+
+
+namespace {
+
+const size_t kBVSize = 64;
+
+int basis[][4] = {
+  {0, 0, 0, 0},
+  {-1, -1, 1, 1},
+  {0, 1, -1, 0},
+  {-1, 0, 0, 1},
+  {1, 0, -1, 0},
+  {0, -1, 0, 1},
+  {1, 1, -2, 0},
+  {0, 0, -1, 1},
+  {0, 0, 1, 0},
+  {-1, -1, 2, 1},
+  {0, 1, 0, 0},
+  {-1, 0, 1, 1},
+  {1, 0, 0, 0},
+  {0, -1, 1, 1},
+  {1, 1, -1, 0},
+  {0, 0, 0, 1}
+};
+
+struct bool_function {
+  using boolvar = std::tuple<bool, char>;
+  expr_ref e;
+  char op;
+  std::vector<boolvar> vars;
+  bool negated;
+
```

```
+  bool_function(ast_manager & m, expr * e) : e(e, m), op(0), negated(false) { }
+
+  bool evaluate(bool x, bool y) {
+    auto eval_var = [&](const boolvar & v) {
+      bool neg; char name;
+      std::tie(neg, name) = v;
+      return neg ? !((name == 'x' ? x : y)) : (name == 'x' ? x : y);
+    };
+
+    bool result;
+    switch (op) {
+    case '&': result = eval_var(vars[0]) && eval_var(vars[1]); break;
+    case '|': result = eval_var(vars[0]) || eval_var(vars[1]); break;
+    case '^': result = eval_var(vars[0]) ^ eval_var(vars[1]); break;
+    default: result = eval_var(vars[0]); break;
+    }
+    return negated ? !result : result;
+  }
+
+  int truth_value(void) {
+    int result = 0;
+    for (size_t i = 0; i < 4; i++) {
+      bool x = i & 2;
+      bool y = i & 1;
+      if (evaluate(x, y))
+        result |= 1 << i;
+    }
+    return result;
+  }
+};
+
+
+using coeff_type = long long;
+using mba_term = std::tuple<coeff_type, bool_function>;
+
+
+struct mba_expr {
+  std::vector<mba_term> terms;
+  ast_manager & m;
+
+  mba_expr(ast_manager & m) : m(m) { }
+};
+
+class mba_tactic : public tactic {
+  ast_manager & m_manager;
+  bv_util m_bv_util;
+  params_ref m_params;
+
+  ast_manager & m() const { return m_manager; }
+
+  bv_util & bv() { return m_bv_util; }
+
+  coeff_type get_coeff(expr * e) {
+    rational r;
+    if (!bv().is_numeral(e, r))
+      throw tactic_exception("expected numeral");
+
+    if (r.is_int64())
```

```
+      return r.get_int64();
+    else if (r.is_int32())
+      return r.get_int32();
+    else if (r.is_uint64()) {
+      return r.get_uint64();
+    }
+    throw tactic_exception("expected int64");
+  }
+
+  bool is_indeterminate(expr * e) {
+    if (!is_app(e))
+      return false;
+
+    app * a = to_app(e);
+    if (a->get_num_args() != 0)
+      return false;
+
+    sort * s = a->get_decl()->get_range();
+    if (!bv().is_bv_sort(s))
+      return false;
+
+    unsigned bv_size = s->get_parameter(0).get_int();
+    if (bv_size != kBVSize)
+      return false;
+
+    func_decl * f = a->get_decl();
+    if (f->get_name() == "x" || f->get_name() == "y")
+      return true;
+    return false;
+  }
+
+  expr * mk_indeterminate(const char* name) {
+    return m().mk_const(name, bv().mk_sort(kBVSize));
+  }
+
+  expr * mk_numeral(int64_t u) {
+    return bv().mk_numeral(u, kBVSize);
+  }
+
+  bool build_bool_function_terms(app * a, bool_function & bf) {
+    unsigned num_args = a->get_num_args();
+    if (num_args > 2) {
+      return false;
+    }
+
+    for (unsigned i = 0; i < num_args; i++) {
+      expr * arg = a->get_arg(i);
+      if (!is_app(arg)) {
+        return false;
+      }
+      app * arg_app = to_app(arg);
+
+      if (bv().is_bv_not(arg_app)) {
+        expr * indet = arg_app->get_arg(0);
+        if (!is_indeterminate(indet)) {
+          return false;
+        }
+        char name = to_app(indet)->get_decl()->get_name().str()[0];
```

```
+        bf.vars.push_back(std::make_tuple(true, name));
+      } else if (is_indeterminate(arg_app)) {
+        char name = arg_app->get_decl()->get_name().str()[0];
+        bf.vars.push_back(std::make_tuple(false, name));
+      } else {
+        TRACE("mba", tout << "not an indeterminate\n";);
+        return false;
+      }
+    }
+    return true;
+  }
+
+  bool build_bool_function(expr * e, bool_function & bf) {
+    if (!is_app(e))
+      return false;
+
+    app * a = to_app(e);
+    if (bv().is_bv_not(a)) {
+      bf.negated = !bf.negated;
+      return build_bool_function(a->get_arg(0), bf);
+    } else if (bv().is_bv_and(a)) {
+      bf.op = '&';
+      return build_bool_function_terms(a, bf);
+    } else if (bv().is_bv_or(a)) {
+      bf.op = '|';
+      return build_bool_function_terms(a, bf);
+    } else if (bv().is_bv_xor(a)) {
+      bf.op = '^';
+      return build_bool_function_terms(a, bf);
+    }
+
+    if (!is_indeterminate(a))
+      return false;
+
+    char name = a->get_decl()->get_name().str()[0];
+    bf.vars.push_back(std::make_tuple(false, name));
+    return true;
+  }
+
+  bool build_mba_expr(expr * e, mba_expr & mba, bool negative) {
+    if (!is_app(e))
+      return false;
+
+    app * a = to_app(e);
+    if (bv().is_bv_add(a)) {
+      unsigned num_args = a->get_num_args();
+
+      if (num_args != 2)
+        return false;
+
+      expr * arg1 = a->get_arg(0);
+      expr * arg2 = a->get_arg(1);
+
+      if (!build_mba_expr(arg1, mba, negative))
+        return false;
+      if (!build_mba_expr(arg2, mba, negative))
+        return false;
+      return true;
```

```
+    } else if (bv().is_bv_sub(a)) {
+      unsigned num_args = a->get_num_args();
+      if (num_args != 2)
+        return false;
+
+      expr * arg1 = a->get_arg(0);
+      expr * arg2 = a->get_arg(1);
+
+      if (!build_mba_expr(arg1, mba, negative))
+        return false;
+      if (!build_mba_expr(arg2, mba, !negative))
+        return false;
+      return true;
+    } else if (bv().is_bv_mul(a)) {
+      if (a->get_num_args() != 2)
+        return false;
+
+      expr * coef = a->get_arg(0);
+      expr * term = a->get_arg(1);
+      if (!bv().is_numeral(coef))
+        return false;
+
+      bool_function bf(m(), term);
+      if (!build_bool_function(term, bf))
+        return false;
+
+      coeff_type c = get_coeff(coef);
+      if (negative)
+        c = -c;
+      mba.terms.push_back(std::make_tuple(c, bf));
+      return true;
+    } else if (bv().is_numeral(a)) {
+      expr * indet = mk_indeterminate("x");
+      expr * term =
bv().mk_bv_not(bv().mk_bv_and(indet,bv().mk_bv_not(indet)));
+
+      bool_function bf(m(), term);
+      if (!build_bool_function(term, bf))
+        return false;
+
+      coeff_type c = get_coeff(a);
+      if (negative)
+        c = -c;
+      mba.terms.push_back(std::make_tuple(-c, bf));
+      return true;
+    }
+
+    // probably a bool function
+    bool_function bf(m(), e);
+    if (!build_bool_function(e, bf))
+      return false;
+
+    coeff_type c = negative ? -1 : 1;
+    mba.terms.push_back(std::make_tuple(c, bf));
+    return true;
+  }
+
+  expr * mk_expressiion(int * basis) {
```

```
+    expr * x = mk_indeterminate("x");
+    expr * y = mk_indeterminate("y");
+    expr * x_and_y = bv().mk_bv_and(x, y);
+    expr * one = mk_numeral(-1ull);
+    expr * basis_expr[] = { x, y, x_and_y, one };
+
+    expr * result = nullptr;
+    for (size_t i = 0; i < 4; i++) {
+      if (basis[i] == 0)
+        continue;
+
+      expr * coterm = bv().mk_bv_mul(
+        mk_numeral(basis[i]),
+        basis_expr[i]
+      );
+      if (!result)
+        result = coterm;
+      else
+        result = bv().mk_bv_add(result, coterm);
+    }
+    return result;
+  }
+
+  expr * construct_simplified_mba(expr * e) {
+    mba_expr mba(m());
+
+    if (!build_mba_expr(e, mba, false))
+      return nullptr;
+
+    int basis_comb[4] = {0, 0, 0, 0};
+    for (size_t i = 0; i < mba.terms.size(); i++) {
+      int truth_value = std::get<1>(mba.terms[i]).truth_value();
+      coeff_type coeff = std::get<0>(mba.terms[i]);
+      for (size_t j = 0; j < 4; j++) {
+        basis_comb[j] += basis[truth_value][j] * coeff;
+      }
+    }
+    return mk_expressiion(basis_comb);
+  }
+
+  bool simplify_form(expr * e, expr_ref & result) {
+    if (!is_app(e))
+      return false;
+
+    app * a = to_app(e);
+
+    if (m().is_eq(a) || m().is_distinct(a)) {
+      SASSERT(a->get_num_args() == 2);
+      expr * lhs = a->get_arg(0);
+      expr * rhs = a->get_arg(1);
+      expr * simplified = construct_simplified_mba(lhs);
+
+      if (simplified) {
+        if (m().is_eq(a))
+          result = m().mk_eq(simplified, rhs);
+        else {
+          expr * args[] = { simplified, rhs };
+          result = m().mk_distinct(2, args);
```

```
+        }
+        return true;
+      }
+    }
+    return false;
+  }
+
+  void simplify_goal(goal & g) {
+    if (g.inconsistent())
+      return;
+    if (g.proofs_enabled()) {
+      return; // not supported
+    }
+
+    expr_ref new_curr(m());
+    proof_ref new_pr(m());
+    unsigned size = g.size();
+    for(unsigned idx = 0; idx < size; idx++) {
+      if (g.inconsistent()) {
+        break;
+      }
+      expr * curr = g.form(idx);
+      if (simplify_form(curr, new_curr)) {
+        g.update(idx, new_curr, new_pr, g.dep(idx));
+      }
+    }
+  }
+
+public:
+  mba_tactic(ast_manager & m, params_ref const & p) : m_manager(m),
m_bv_util(m), m_params(p) { }
+
+  void collect_statistics(statistics & st) const override { }
+
+  void operator()(goal_ref const & in, goal_ref_buffer & result) override {
+    TRACE("mba", tout << "mba tactic\n";);
+    simplify_goal(*in.get());
+    in->inc_depth();
+    result.push_back(in.get());
+  }
+
+  void cleanup() override { }
+
+  tactic * translate(ast_manager & m) override { return alloc(mba_tactic, m,
m_params); }
+
+  const char* name() const override { return "mba"; }
+};
+
+} // namespace
+
+tactic * mk_mba_tactic(ast_manager & m, params_ref const & p) {
+  return clean(alloc(mba_tactic, m, p));
+}
+
diff --git a/src/tactic/bv/mba_tactic.h b/src/tactic/bv/mba_tactic.h
new file mode 100644
index 000000000..b779cdc2b
```

```
--- /dev/null
+++ b/src/tactic/bv/mba_tactic.h
@@ -0,0 +1,12 @@
+#pragma once
+
+#include "util/params.h"
+
+class ast_manager;
+class tactic;
+
+tactic * mk_mba_tactic(ast_manager & m, params_ref const & p = params_ref());
+
+/*
+    ADD_TACTIC("mba", "Toy MBA simplifier", "mk_mba_tactic(m, p)")
+*/
\ No newline at end of file
```

先看server.py，大概意思就是要求输入的mba表达式满足以下条件：

1. 数字不超过8位
2. 表达式长度不超过200个字符
3. 表达式项数不超过15
4. 这个表达式不能等于自己

再来看patch，直接丢给ds它就会告诉你这里面有个数据溢出的漏洞，用int类型存储了long long类型数据的计算结果

```
94  +
95  +
96  +using coeff_type = long long;
97  +using mba_term = std::tuple<coeff_type, bool_function>;
98  +
99  +
```

```
317 +
318 +  expr * construct_simplified_mba(expr * e) {
319 +    mba_expr mba(m());
320 +
321 +    if (!build_mba_expr(e, mba, false))
322 +      return nullptr;
323 +
324 +    int basis_comb[4] = {0, 0, 0, 0};
325 +    for (size_t i = 0; i < mba.terms.size(); i++) {
326 +      int truth_value = std::get<1>(mba.terms[i]).truth_value();
327 +      coeff_type coeff = std::get<0>(mba.terms[i]);
328 +      for (size_t j = 0; j < 4; j++) {
329 +        basis_comb[j] += basis[truth_value][j] * coeff;
330 +      }
331 +    }
332 +    return mk_expressiion(basis_comb);
333 +  }
334 +
```

因此我们构造一个足够大且满足条件的mba表达式使其溢出，即可导致expr!=expr：

99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+9999
9999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)

```
C:\Users\jyzho>nc 121.41.238.106 21012
Please enter the expression: 99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+9
9999999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)+99999999*(x^y)
Flag: aliyunctf{9049ca97-827d-4db6-a05b-a895f5f5e32c}
```