Escaping the Maze

A Z3 Solver Tutorial for Reverse Engineers

#TDefenceBusiness

Malware Lab

Summary

Executive Summary 05

1. Introduction 06

2. The Challenge 07

    2.1 The Encryption Function 08

3. Tutorial 11

    3.1 How It Works 11

    3.2 Variables and Sorts 13

    3.3 The Solver 13

    3.4 Logical Functions 14

    3.5 BitVec: Bitwise Operations 15

    3.6 Extract and Concat 18

    3.7 ZeroExt and SignExt 21

    3.8 Symbolic If 22

    3.9 Enumerating All Solutions 23

    3.10 Simplify and Equivalence Verification 24

    3.11 Incremental Solving – push and pop 26

4. Challenge Solution 27

    4.1 Why reconstruct 30

    4.2 The Symbolic Index Problem 31

Summary

5. Common Gotchas 33
5.1 Width Mismatch Between BitVecs 33
5.2 Implicit Signed Promotion in C 33
5.3 % and / Are Signed by Default 34
5.4 Python Constants Mixed with BitVec 35
5.5 Duplicate Variable Names 35
5.6 Bool vs BitVec(1) 36
5.7 Unconstrained Variables in the Model 37
5.8 Non-Linear Arithmetic 37
5.9 Large Constraints vs. Many Small Constraints 38
5.10 Symbolic Indices into Concrete Structures 39
6. Conclusion 40
6.1 The Current Ecosystem 40

Our Malware Lab

T-Defence Malware Lab daily performs dissection of malware with the aim of timely understanding the technological evolutions of attacks, consolidating the knowledge of necessary to make more effective and faster the process of incidents responding, contributing to spreading information about emerging threats into the expert's community and among its clients.

Malware Lab analysts are continuously engaged in searching and experimenting new analysis tools, for increasing accuracy and scope of action with regard to the proliferation of new evasion and anti-analysis techniques adopted by malware.

The Malware Lab is also committed to the development of proprietary tools for malware analysis and supporting the management and response of incidents.

Besides malware analysis, Malware Lab ideated and implemented an automatic process of extraction of **Indicators of Compromise (IOC)** that is daily run on dozens of new malwares, intercepted in the wide for populating our Knowledge Base.

Executive Summary

Reverse-engineering a custom cipher is a task where human working memory collapses quickly. When an algorithm combines multiple rounds of permutation, key mixing, and substitution, manually inverting it becomes error-prone and slow. This report demonstrates how an SMT solver, specifically Z3, can turn the problem upside down: instead of inverting the cipher by hand, we re-express the encryption function in a symbolic language and let the solver compute the inverse for us.

The walkthrough uses a Substitution-Permutation Network from KALMAR CTF 2024 as the vehicle, but the methodology is directly applicable to the encryption routines found in loaders, ransomware, and packers analyzed inside the Malware Lab. After covering the Z3 fundamentals that matter in practice, the report presents a complete solve script, dissects the subtle symbolic-index problem that arises in almost every non-trivial cipher, and closes with a catalog of the pitfalls that most frequently derail a first Z3 session.

Readers who finish this document should be able to model a custom cipher in Z3 from scratch, constrain it with partially known plaintext, and recover the original input without writing a single line of inversion logic. Beyond the specific challenge, the report provides a reusable reference for the constraint-solving patterns that underpin modern symbolic execution engines such as angr, Triton, and Binsec.

Autori:

- Edoardo D’Errico: Cybersecurity Researcher

1. Introduction

In 1956, cognitive psychologist George A. Miller demonstrated that **human working memory is limited to approximately 7 items (±2) at a time**. This limit holds regardless of the type of information: whether dealing with digits, words, or musical tones, the brain always collapses around the same threshold.

In the context of reverse engineering, cryptographic function analysis, or scientific work more broadly, this principle has a very direct application: most people have experienced approaching a problem and feeling completely lost, as if navigating a maze. When the number of constraints and variables exceeds that threshold, the human brain simply stops being reliable. **This is not a question of skill, it is physiology**.

This is where **Z3** comes in.

Z3 is an SMT solver created by Microsoft Research, with applications across a surprisingly wide range of fields, from biology and formal network configuration verification to automated test generation and vulnerability research. At its core, an SMT solver determines whether a mathematical formula is satisfiable, generalizing the classical boolean SAT problem to richer domains: integers, reals, bit-vectors, arrays, and strings.

In security, the most relevant application is **symbolic execution**: at each conditional branch, the solver verifies which paths are actually reachable, transforming the question “which input activates this code path?” from an intuition problem into a mathematical one. Actively maintained tools built on SMT solvers include angr, Triton, KLEE, and Binsec.

In this article, we will demonstrate the practical necessity of tools like Z3 through a reversing challenge in which we analyze and invert a custom cipher.

2. The Challenge

The technique is demonstrated on “Symmetry”, a cipher from KALMAR CTF 2024. Rather than working on a sanitized reproduction of a real sample, we deliberately use a competition cipher: it isolates the core analytical technique without the surrounding noise of a full malware analysis, making the methodology easier to follow and reproduce. The same approach applies directly to custom encryption schemes found in loaders, ransomware, and packers.

For this specific challenge, we are provided with a binary and a data.txt file. After loading the binary into a disassembler and cleaning up the decompiled output, this is what we get:

```c
puts("Welcome to the testing service for my new cryptosystem.");
n_blocks = 0;
while (1)
{
    printf("Number of blocks: ");
    __isoc99_scanf("%u", &n_blocks);
    if (n_blocks <= 100)
    {
        blocks = calloc(n_blocks, 8u);
        shifts = calloc(n_blocks, 16u);
        pts    = calloc(n_blocks, 8u);
        cts    = calloc(n_blocks, 8u);
        for (b = 0; b < n_blocks; ++b)
        {
            printf("Please provide a key for block %u: ", b);
            for (i = 0; i <= 7; ++i)
                __isoc99_scanf("%2hhx", &blocks[8 * b + i]);
            for (j = 0; j <= 0xF; ++j)
            {
                printf("Please provide shift %u for block %u: ", j, b);
                __isoc99_scanf("%2hhx", &shifts[16 * b + j]);
            }
            printf("Please provide plaintext for block %u: ", b);
            for (k = 0; k <= 7; ++k)
                __isoc99_scanf("%2hhx", &pts[8 * b + k]);
        }

        start_encrypt(n_blocks, blocks, shifts, pts, cts);
        puts("Ciphertexts:");
        for (m = 0; m < n_blocks; ++m)
```

```
        {
            printf("Block %u: ", m);
            for (n = 0; n <= 7; ++n)
                printf("%02hhx", cts[8 * m + n]);
            putchar(10);
        }
        free(blocks);
        free(shifts);
        free(pts);
        free(cts);
    }
    else
    {
        puts("That is a bit too much...");
    }
}
```

The binary is a custom encryption service that accepts, for each block, a key, 16 shift values, and a plaintext, and returns the corresponding ciphertext.

The `data.txt` file contains the ciphertexts, keys, and shifts. Our task is to invert the encryption function to recover the original input — the flag.

2.1 The Encryption Function

Nothing surprising so far. The interesting part comes when we reverse start_encrypt. After analyzing all the functions it calls and rewriting them from pseudo-C to Python, this is the result:

```
Python
def scramble(shift_elem, idx):
    is_odd = idx & 1
    half_to_shift = shift_elem >> 1
    half_idx = idx >> 1
    if (shift_elem & 1) == 1:
        return ((2 * (half_to_shift - half_idx)) & 0xE) - is_odd + 1
    else:
        return 2 * (half_to_shift + half_idx) + is_odd

def extract_first_high_then_low(block_ptr, i):
    value = block_ptr[i >> 1]
    if (i & 1) != 0:
        return value & 0xF
    else:
        return value >> 4
```

```
def insert_at(new_buf, pos, to_ins):
    real_pos = pos >> 1
    cur_buf_val = new_buf[real_pos]
    if (pos & 1) != 0:
        computed = (cur_buf_val & 0xF0) + to_ins
    else:
        computed = (cur_buf_val & 0xF) + 16 * to_ins
    new_buf[pos >> 1] = computed
    return new_buf

def encrypt(pts):
    new_shifts = [0] * 16
    new_buf = [0] * 8
    cts = []
    for j in range(len(pts)):
        cur_block = pts[j]
        for k in range(16):
            for i in range(16):
                idx = permutation_idx[i]
                new_shifts[idx] = scramble(shifts[j][k], i)
            for m in range(16):  # permutation
                extracted_char = extract_first_high_then_low(cur_block, m)
                insert_at(new_buf, new_shifts[m], extracted_char)
            for n in range(16):  # keymixing
                extracted_block = extract_first_high_then_low(keys[j], n)
                extracted_intern = extract_first_high_then_low(new_buf, n)
                computed_val = scramble(extracted_intern, extracted_block)
                insert_at(new_buf, n, computed_val & 0xF)
            for x in range(16):  # substitution
                extract = extract_first_high_then_low(new_buf, x)
                insert_at(cur_block, x, new_shifts[new_shifts[extract]])
        cts.append(cur_block)
    return cts
```

Looking at the encryption function at a high level, we can recognize the structure of a **Substitution–Permutation Network (SPN)**.

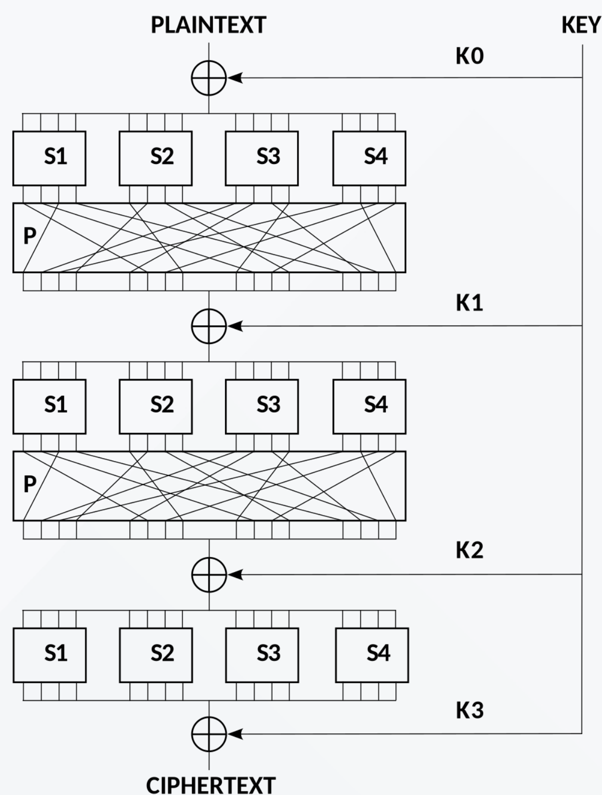

Figure 1 – Substitution-Permutation Network (SPN) general structure

A Substitution-Permutation Network is a family of block ciphers built around two fundamental operations, repeated across multiple rounds:

Substitution: each element of the block is replaced with another via a non-linear function (the S-box), which breaks the direct relationship between plaintext and ciphertext.

Permutation: bits or nibbles are reshuffled in position, providing diffusion — informally, a change at one position propagates across the entire block within a few rounds.

Key mixing (the scramble function in this case) binds the ciphertext to the secret key.

AES is the most well-known example: SubBytes → ShiftRows → MixColumns → AddRoundKey, repeated 10, 12, or 14 times.

In principle, inverting a function composed of multiple steps requires inverting each individual step and applying them in reverse order. With enough patience, this is entirely possible. But what if we could instead provide the necessary data to a computer and have it compute the inverse function, without ever explicitly implementing it? This is exactly where Z3 comes in.

3. Tutorial

At a high level, Z3 can be thought of as a tool similar to MATLAB or NumPy — software used to perform scientific computations on a machine. **The key difference is that Z3 is not limited to classical mathematical operations: it also handles bitwise operations, correctly emulating the behavior of a physical machine**.

3.1 How It Works

Z3 is an **SMT solver** (Satisfiability Modulo Theories). The problem it solves is the following: given a logical formula $\varphi(x_1, x_2, \ldots, x_n)$, does there exist an assignment of values to the variables that makes it true?

$\exists\, x_1, x_2, \ldots, x_n \,.\, \varphi(x_1, \ldots, x_n) = \top$

This problem is called **satisfiability**. The simplest version is **SAT**: given boolean variables and a propositional formula, does a satisfying assignment exist? SAT is NP-complete — theoretically hard, but in practice modern solvers handle enormous instances in seconds using algorithms such as **CDCL** (Conflict-Driven Clause Learning).

SMT is generalized SAT: variables are not limited to booleans but can inhabit richer **theories** such as integers, reals, bit-vectors, arrays, and strings. The solver combines a SAT engine with specialized **theory solvers** for each domain, coordinated by the **DPLL(T)** algorithm. When Z3 reasons about integers it uses linear arithmetic (LA); when reasoning about bit-vectors it uses bit-blasting procedures that reduce the problem to SAT.

In practice, every Z3 session follows the same five steps:

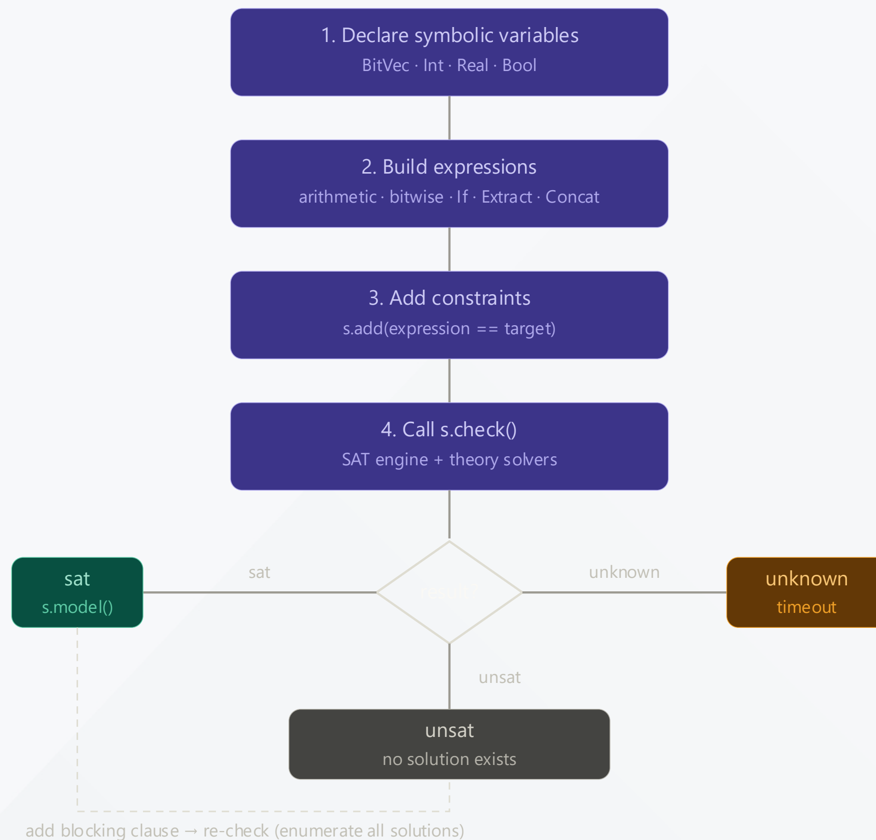

Figure 2 – The canonical Z3 workflow: declare, build, constrain, check, inspect

As a first example, let us solve the equation $6x^2 + 11x - 35 = 0$.

```python
from z3 import *

s = Solver()

x = Real('x')
s.add(6*x**2 + 11*x - 35 == 0)

if s.check() == sat:
    print(s.model())  # [x = 5/3]
```

Note that Z3 returns 5/3 as an exact fraction — it works with precise rational arithmetic, not floating point.

3.2 Variables and Sorts

Z3 distinguishes between different mathematical types, called **sorts**. The choice matters: each sort has different semantics, and the solver uses different decision procedures for each.

```python
Python
from z3 import *

# Int: unbounded mathematical integer, no overflow, no modular arithmetic
x = Int('x')
a, b, c = Ints('a b c')   # Multiple declaration

# Real: rational numbers (exact fractions, not floating point)
r = Real('r')
p, q = Reals('p q')

# Bool: logical value, True or False
flag = Bool('flag')

# BitVec: fixed-width integer, WITH overflow, behaves like C types
byte_var  = BitVec('byte_var', 8)    # uint8_t: values 0-255
word_var  = BitVec('word_var', 32)   # uint32_t
qword_var = BitVec('qword_var', 64)  # uint64_t
```

The critical distinction is between `Int` and `BitVec`. `Int` is the pure mathematical type: no upper bound, no overflow, exact division. `BitVec` models machine types: `BitVec('x', 8)` behaves exactly like a `uint8_t` in C — $255 + 1 \equiv 0 \pmod{256}$, division truncates, and shifts have precise semantics. When modeling cryptographic algorithms, registers, or any code operating on bytes, BitVec is almost always the right choice.

3.3 The Solver

```python
Python
from z3 import *

x, y = Ints('x y')

s = Solver()
s.add(x + y == 90)
s.add(x * 2 == y)
s.add(x > 0)
```

```
result = s.check()

if result == sat:       # At least one solution exists
    m = s.model()
    print(f"x = {m[x]}")
    print(f"y = {m[y]}")
elif result == unsat:   # Constraints are contradictory, no solution
    print("Constraints cannot be satisfied")
else:                   # unknown, timeout or undecidable theory
    print("The solver gave up")
```

`s.check()` returns sat if **at least one** solution exists — it says nothing about uniqueness. The model returns Z3 objects, not Python primitives:

```
Python
m = s.model()

x_val  = m[x].as_long()            # Int -> Python int
r_val  = float(m[r].as_fraction()) # Real -> float (via exact Fraction)
bv_val = m[byte_var].as_long()     # BitVec -> Python int (always unsigned)
```

3.4 Logical Functions

Beyond standard Python operators (+, -, *, ==, !=, <, >, &, |, ^), Z3 exposes several useful functions:

```
Python
from z3 import *

a, b, c = Bools('a b c')

And(a, b, c)    # a AND b AND c
Or(a, b, c)     # a OR b OR c
Not(a)          # NOT a
Xor(a, b)       # a XOR b

x, y, z = Ints('x y z')
Distinct(x, y, z)  # x != y AND x != z AND y != z, all pairwise distinct
```

`Distinct` is useful for combinatorial problems such as Sudoku, where each digit must appear exactly once per row.

3.5 BitVec: Bitwise Operations

When working with ciphers and low-level algorithms, BitVec will account for roughly 90% of your solver code. Most operators behave as expected, but there are a few critical pitfalls related to signed/unsigned semantics.

Right Shift: >> vs LShR

In Z3, the >> operator on a BitVec performs an **arithmetic right shift**: the sign bit is replicated during the shift, matching the behavior of **signed** types in C. For unsigned types, the shift should fill with zeros, which is what LShR does.

Formally, for a value x of n bits shifted by k positions:

$$x \gg k = (x_{n-1} \dots x_{n-1})_k \cdot x_{n-1} \dots x_k \quad \text{(arithmetic)}$$

$$\mathrm{LShR}(x, k) = (0 \dots 0)_k \cdot x_{n-1} \dots x_k \quad \text{(logical)}$$

```python
from z3 import *

x = BitVecVal(0b10000000, 8)  # 128 unsigned, -128 signed

arithmetic = x >> 1       # 0b11000000 = 0xC0 (sign bit replicated)
logical    = LShR(x, 1)   # 0b01000000 = 0x40 (zero-filled)

print(simplify(arithmetic))  # 192
print(simplify(logical))     # 64
```

The same distinction applies to division and comparison:

Operation	Signed	Unsigned
Right shift	x >> n	LShR(x, n)
Division	x / n	UDiv(x, n)
Modulo	x % n	URem(x, n)
Less than	x < y	ULT(x, y)
Greater than	x > y	UGT(x, y)

PRACTICAL RULE

If the original code uses uint8_t, uint32_t, or unsigned types in general, always use LShR, UDiv, and URem.

Rotations

Unlike shifts, rotations are **lossless**: no bits are discarded — whatever exits from one end re-enters from the other.

$$ROL_k(x) = (x \ll k) \mid LShR(x, n - k)$$

$$ROR_k(x) = LShR(x, k) \mid (x \ll (n - k))$$

```
Python
from z3 import *

x = BitVec('x', 32)

rol = RotateLeft(x, 5)
ror = RotateRight(x, 5)

# Property: ROR(ROL(x, k), k) == x for all x
s = Solver()
s.add(RotateRight(RotateLeft(x, 5), 5) != x)
print(s.check())  # unsat, they are always equal, for every x
```

Example 1 – Mini Cipher with Rotations

Suppose you encounter the following encryption scheme during reversing:

```
Python (decompiled model)
def encrypt(plaintext: list[int], key: int) -> list[int]:
    ct = []
    for i, byte in enumerate(plaintext):
        b = byte ^ key
        b = ((b << (i % 8)) | (b >> (8 - i % 8))) & 0xFF
        b = b ^ (i * 0x5A)
        ct.append(b & 0xFF)
    return ct

ciphertext = [0x04, 0x46, 0xB8, 0x26, 0xFB, 0xC1, 0x40, 0xE9]
# The key is unknown, but we know the plaintext starts with b"FLAG"
```

Instead of manually inverting each operation, we model the entire cipher in Z3 using symbolic variables:

```python
from z3 import *

ciphertext = [0x04, 0x46, 0xB8, 0x26, 0xFB, 0xC1, 0x40, 0xE9]

key = BitVec('key', 8)
s = Solver()

for i, ct_byte in enumerate(ciphertext):
    pt = BitVec(f'pt_{i}', 8)
    s.add(pt >= 0x20, pt <= 0x7e)  # Printable ASCII

    b = pt ^ key
    b = RotateLeft(b, i % 8)
    b = b ^ BitVecVal((i * 0x5A) & 0xFF, 8)

    s.add(b == ct_byte)

# Known plaintext: starts with "FLAG"
for i, c in enumerate(b"FLAG"):
    s.add(BitVec(f'pt_{i}', 8) == c)

if s.check() == sat:
    m = s.model()
    print(f"Key: {m[key].as_long()}")
    flag = bytes([m[BitVec(f'pt_{i}', 8)].as_long() for i in range(len(ciphertext))])
    print(f"Plaintext: {flag}")
```

The solver finds the key and decrypts the message without us having manually inverted a single operation. This is the core value proposition of Z3: re-express the cipher with symbolic variables, add the known constraints, and let the solver do the hard work.

BitVecVal — Fixed-Width Constants

When a constant should respect n-bit overflow semantics, wrap it in `BitVecVal`:

```python
from z3 import *

s = Solver()
var   = BitVec('var', 16)
const = BitVecVal(1000, 16)

# 1000 * 2000 = 2,000,000, but in 16 bits: 2,000,000 mod 65536 = 33920
result = const * BitVecVal(2000, 16)
s.add(var == result)
```

```
if s.check() == sat:
    print(s.model()[var].as_long())  # 33920, not 2,000,000
```

3.6 Extract and Concat

These two operations are fundamental for nibble- and byte-level work, and appear constantly when reversing custom ciphers.

Extract

`Extract(high, low, x)` extracts bits from position low to position high (inclusive). The result has width high - low + 1 bits. Bits are numbered from 0 (LSB) to n-1 (MSB).

```
Python
from z3 import *

x = BitVecVal(0xDEADBEEF, 32)

byte3 = Extract(31, 24, x)  # 0xDE, most significant byte
byte2 = Extract(23, 16, x)  # 0xAD
byte1 = Extract(15, 8,  x)  # 0xBE
byte0 = Extract(7,  0,  x)  # 0xEF, least significant byte

high_nibble = Extract(7, 4, x)  # Upper nibble of byte0: 0xE
low_nibble  = Extract(3, 0, x)  # Lower nibble of byte0: 0xF
```

General pattern for iterating over bytes:

```
Python
def get_byte(bv, i):
    """Extract the i-th byte (0 = LSB)."""
    return Extract(i*8 + 7, i*8, bv)

def get_high_nibble(bv, i):
    return Extract(i*8 + 7, i*8 + 4, bv)

def get_low_nibble(bv, i):
    return Extract(i*8 + 3, i*8, bv)

x = BitVec('x', 64)
bytes_list = [get_byte(x, i) for i in range(8)]
```

Concat

`Concat(a, b, ...)` concatenates BitVecs: the **first argument lands in the high bits**, the last in the low bits. The result width equals the sum of all argument widths.

```python
from z3 import *

a = BitVecVal(0xDE, 8)
b = BitVecVal(0xAD, 8)
c = BitVecVal(0xBE, 8)
d = BitVecVal(0xEF, 8)

result = Concat(a, b, c, d)  # 0xDEADBEEF, a is MSB, d is LSB
```

The Tear-Modify-Reassemble Pattern

The most common use of Extract + Concat is modifying a specific portion of a value while leaving the rest intact, a pattern that appears constantly in nibble-based ciphers:

```python
from z3 import *

buf = BitVec('buf', 8)
new_nibble = BitVec('nibble', 4)

# Replace the high nibble of buf
result_high = Concat(new_nibble, Extract(3, 0, buf))

# Replace the low nibble of buf
result_low  = Concat(Extract(7, 4, buf), new_nibble)
```

Extract and Concat also allow verifying non-obvious equivalences:

```python
from z3 import *

x = BitVec('x', 32)

upper   = Extract(31, 16, x)
lower   = Extract(15, 0,  x)
swapped = Concat(lower, upper)
```

```
s = Solver()
s.add(swapped != RotateLeft(x, 16))
print(s.check())  # unsat, identical for every x
```

Example 2 — Nibble-Swap Cipher

Suppose you encounter this during reversing:

```
Python (decompiled model)
def encrypt(plaintext: bytes, key: int) -> list[int]:
    ct = []
    key_hi = (key >> 4) & 0xF
    key_lo = key & 0xF
    for byte in plaintext:
        hi = (byte >> 4) & 0xF
        lo = byte & 0xF
        hi ^= key_hi
        lo ^= key_lo
        ct.append((lo << 4) | hi)  # swap: lo becomes MSB, hi becomes LSB
    return ct

ciphertext = [0x4F, 0x3E, 0x1F, 0xCC, 0xDE, 0x48, 0xAC]
# The key is unknown, but we know the plaintext starts with b"CTF"
```

With Z3, we model the same logic using `Extract` and `Concat`:

```
Python
from z3 import *

ciphertext = [0x4F, 0x3E, 0x1F, 0xCC, 0xDE, 0x48, 0xAC]

key = BitVec('key', 8)
s = Solver()

for i, ct_byte in enumerate(ciphertext):
    pt = BitVec(f'pt_{i}', 8)
    s.add(pt >= 0x20, pt <= 0x7e)  # Printable ASCII

    hi     = Extract(7, 4, pt)
    lo     = Extract(3, 0, pt)
    key_hi = Extract(7, 4, key)
    key_lo = Extract(3, 0, key)

    hi_enc = hi ^ key_hi
    lo_enc = lo ^ key_lo
    result = Concat(lo_enc, hi_enc)  # lo_enc -> bits 7-4, hi_enc -> bits 3-0
```

```
    s.add(result == ct_byte)

for i, c in enumerate(b"CTF"):
    s.add(BitVec(f'pt_{i}', 8) == c)

if s.check() == sat:
    m = s.model()
    print(f"Key: {hex(m[key].as_long())}")
    flag = bytes([m[BitVec(f'pt_{i}', 8)].as_long() for i in range(len(ciphertext))])
    print(f"Plaintext: {flag}")
```

The key point: modeling the swap requires no tricks. `Concat(lo_enc, hi_enc)` places `lo_enc` in the high bits and `hi_enc` in the low bits — exactly what `(lo << 4) | hi` does in Python. Z3 reasons over the full nibble structure without us writing a single line of inversion logic.

3.7 ZeroExt and SignExt

When widening a BitVec, for example from 8 to 32 bits, the choice of how to fill the upper bits must match the signedness of the original C code:

ZeroExt(k, x) = 0...0 (k times) ‖ x   (uint8_t → uint32_t)

SignExt(k, x) = x_{n-1}...x_{n-1} (k times) ‖ x   (int8_t → int32_t)

```
Python
from z3 import *

x = BitVecVal(0xFF, 8)  # 255 unsigned, -1 signed

zero_ext = ZeroExt(24, x)  # 0x000000FF = 255
sign_ext = SignExt(24, x)  # 0xFFFFFFFF = 4294967295 (-1 as int32)

print(simplify(zero_ext))  # 255
print(simplify(sign_ext))  # 4294967295
```

In practice: if the original code casts with `(uint32_t)byte_val`, use `ZeroExt`; if it casts with `(int32_t)(int8_t)byte_val`, use `SignExt`.

3.8 Symbolic If

`If(condition, value_if_true, value_if_false)` is Z3's ternary operator. The difference from a Python if is that the condition here can be **symbolic**: the solver reasons over both branches simultaneously, without concretely executing either.

```python
from z3 import *

x = BitVec('x', 8)

abs_x = If(x >= 0, x, -x)

s = Solver()
s.add(abs_x == 42)
while s.check() == sat:
    print(s.model()[x].as_long())
    s.add(x != s.model()[x])
```

If Pattern – Lookup Tables as If-Chains

When modeling an S-box or any lookup table, If is the right tool. The idea is to construct a conditional expression that returns the correct value for every possible input:

```python
from z3 import *

SBOX = [0xC, 0x5, 0x6, 0xB, 0x9, 0x0, 0xA, 0xD,
        0x3, 0xE, 0xF, 0x8, 0x4, 0x7, 0x1, 0x2]

def sbox_z3(x, table):
    """Model a lookup table as a Z3 expression."""
    result = BitVecVal(table[0], 4)
    for i in range(1, len(table)):
        result = If(x == i, BitVecVal(table[i], 4), result)
    return result

inp = BitVec('inp', 4)
s = Solver()
s.add(sbox_z3(inp, SBOX) == 0xE)

if s.check() == sat:
    print(hex(s.model()[inp].as_long()))  # 0x9
```

This works even on non-invertible S-boxes and many-to-one functions — Z3 automatically finds all preimages through enumeration.

3.9 Enumerating All Solutions

By default Z3 finds **one** solution. To retrieve all solutions, use the **blocking clause** pattern: after each solution is found, add a constraint that explicitly excludes it.

```python
from z3 import *

x = BitVec('x', 8)
s = Solver()
s.add(x & 0xF0 == 0xB0)  # High nibble fixed at 0xB, low nibble free

solutions = []
while s.check() == sat:
    val = s.model()[x].as_long()
    solutions.append(val)
    s.add(x != val)

print([hex(v) for v in solutions])
# [0xb0, 0xb1, ..., 0xbf], 16 solutions
```

With multiple variables, the blocking clause must exclude the **complete combination**, not individual variables:

```python
# Wrong: separately blocks x=x_val AND y=y_val (too restrictive)
s.add(x != x_val)
s.add(y != y_val)

# Correct: excludes only the specific pair (x_val, y_val)
s.add(Or(x != x_val, y != y_val))
```

The full pattern for recovering a flag as a byte array:

```python
from z3 import *

flag = [BitVec(f"flag_{i}", 8) for i in range(8)]
s = Solver()

for c in flag:
    s.add(c >= 0x20, c <= 0x7e)  # Printable ASCII

# [...other constraints on the flag...]

while s.check() == sat:
    m = s.model()
    result = bytes([m[flag[i]].as_long() for i in range(len(flag))])
    print(result)
    s.add(Or([flag[i] != m[flag[i]] for i in range(len(flag))]))
```

3.10 Simplify and Equivalence Verification

MBA — Mixed Boolean Arithmetic

Before discussing simplify, it is worth understanding the context in which these tools become indispensable in reversing: **MBA**, or Mixed Boolean Arithmetic.

MBA is an **obfuscation technique** that replaces simple arithmetic operations with equivalent expressions that mix arithmetic (+, -, *) and boolean operations (AND, OR, XOR, NOT). The objective is to make the code unrecognizable at a glance — decompilers present it as-is, and even experienced reverse engineers struggle to determine what a function is actually computing.

A concrete example: the same add function, before and after one layer of MBA:

```c
// Before
int add(int x, int y) { return x + y; }

// After
int add(int x, int y) {
    return (~(~x & ~y)) + (~(~x | ~y));
}
```

Both versions are **mathematically identical** for every possible input. To verify, apply De Morgan's laws:

$\sim(\sim x \,\&\, \sim y) \;=\; x \mid y \qquad (\text{De Morgan})$

$\sim(\sim x \mid \sim y) \;=\; x \,\&\, y \qquad (\text{De Morgan})$

$(x \mid y) + (x \,\&\, y) \;=\; x + y$

In a real sample, however, an obfuscator applies dozens of these layers in cascade, making manual reasoning completely impractical. This is where simplify and equivalence verification come in.

simplify

simplify() reduces a Z3 expression by applying a fixed set of algebraic rewrite rules. It is useful for debugging and collapsing redundant expressions, but **it is not a general MBA de-obfuscator**: on multi-layer transformations it will not necessarily produce the simplest form.

```python
from z3 import *

x = BitVec('x', 32)

print(simplify(x ^ x))          # 0
print(simplify(x + x - x * 2))  # 0
print(simplify(x & 0xFFFFFFFF)) # x (AND with all ones is identity)
```

Equivalence Verification

The correct pattern for reasoning about MBA is to assert that the two expressions differ and check for `unsat`. If the two forms are identical for every possible input, the solver will never find a counterexample, regardless of how many obfuscation layers were applied:

$$A \equiv B \quad \Leftrightarrow \quad \nexists\, x \,.\, A(x) \neq B(x) \quad \Leftrightarrow \quad UNSAT(\exists\, x \,.\, A(x) \neq B(x))$$

```python
Python
from z3 import *

x, y = BitVecs('x y', 32)

lhs = x + y
rhs = ~(~x & ~y) + ~(~x | ~y)

s = Solver()
s.add(lhs != rhs)
print(s.check())  # unsat -> equivalent for all x, y
```

3.11 Incremental Solving – push and pop

The solver supports nested scopes via `push()` and `pop()`. Constraints added inside a scope are removed when `pop()` is called. This is useful for exploring multiple scenarios without rebuilding the solver from scratch:

```python
Python
from z3 import *

x = Int('x')
s = Solver()
s.add(x > 10)  # Permanent constraint

# Scenario 1: 10 < x < 20
s.push()
s.add(x < 20)
if s.check() == sat:
    print(f"Scenario 1: x = {s.model()[x]}")
s.pop()  # Removes x < 20

# Scenario 2: x > 200 (previous constraint is gone)
s.push()
s.add(x > 200)
if s.check() == sat:
    print(f"Scenario 2: x = {s.model()[x]}")
s.pop()
```

`push/pop` is particularly useful in symbolic execution: at each branch, push the branch condition, explore, then pop to return to the previous state — exactly what engines like angr do internally.

4. Challenge Solution

Armed with the Z3 fundamentals covered above, we can now decrypt the custom cipher. Below is the full solve script with comments on the most significant sections.

Python – full solve script

```
from z3 import BitVec, BitVecVal, Concat, Extract, If, Solver, sat


def reconstruct(splitted):
    return [(splitted[i + 1] | (splitted[i] << 4)) for i in range(0, len(splitted), 2)]


permutation_idx = [9, 10, 8, 1, 14, 3, 7, 15, 11, 12, 2, 0, 4, 5, 6, 13]

keys = [
    reconstruct([2, 3, 3, 5, 3, 3, 1, 4, 1, 1, 3, 3, 1, 2, 2, 0]),
    reconstruct([5, 1, 5, 4, 7, 3, 2, 0, 0, 1, 7, 1, 0, 6, 2, 7]),
    reconstruct([2, 6, 6, 0, 5, 6, 5, 1, 6, 4, 7, 1, 1, 7, 3, 1]),
    reconstruct([4, 3, 0, 5, 2, 0, 4, 2, 7, 7, 7, 1, 1, 1, 7, 5]),
    reconstruct([1, 0, 0, 0, 4, 5, 3, 6, 3, 4, 7, 6, 4, 0, 1, 2]),
    reconstruct([5, 5, 7, 1, 3, 1, 7, 6, 6, 3, 1, 1, 2, 4, 6, 7]),
    reconstruct([2, 6, 2, 4, 1, 6, 0, 3, 7, 0, 6, 3, 0, 6, 7, 3]),
]

shifts = [
    [0, 1, 3, 5, 0, 1, 1, 0, 0, 1, 3, 5, 0, 1, 1, 0],
    [0, 1, 3, 5, 0, 1, 1, 0, 0, 1, 3, 5, 0, 1, 1, 0],
    [0, 1, 3, 5, 0, 1, 1, 0, 0, 1, 3, 5, 0, 1, 1, 0],
    [0, 1, 3, 5, 0, 1, 1, 0, 0, 1, 3, 5, 0, 1, 1, 0],
    [0, 1, 3, 5, 0, 1, 1, 0, 0, 1, 3, 5, 0, 1, 1, 0],
    [0, 1, 3, 5, 0, 1, 1, 0, 0, 1, 3, 5, 0, 1, 1, 0],
    [0, 1, 3, 5, 0, 1, 1, 0, 0, 1, 3, 5, 0, 1, 1, 0],
]

ciphertexts = [
    reconstruct([8, 12, 10, 7, 2, 6, 3, 2, 14, 1, 8, 4, 2, 12, 9, 15]),
    reconstruct([10, 13, 5, 2, 13, 12, 11, 5, 14, 5, 3, 12, 4, 11, 0, 9]),
    reconstruct([10, 4, 0, 3, 9, 13, 13, 2, 2, 1, 0, 4, 3, 15, 11, 12]),
    reconstruct([7, 13, 1, 13, 9, 9, 9, 10, 9, 12, 3, 0, 1, 10, 7, 12]),
    reconstruct([13, 3, 10, 6, 9, 9, 2, 13, 1, 10, 13, 0, 4, 2, 1, 0]),
    reconstruct([6, 2, 2, 2, 15, 9, 12, 4, 7, 6, 2, 15, 1, 10, 14, 7]),
    reconstruct([10, 12, 6, 14, 14, 2, 14, 12, 15, 0, 15, 0, 8, 9, 4, 2]),
]
```

```
n_blocks = len(ciphertexts)
known_start = b"kalmar{"


def extract_first_high_then_low(block_ptr: list, i: int) -> int:
    value = block_ptr[i >> 1]
    return (value & 0xF) if (i & 1) else (value >> 4)


def extract_first_high_then_low_z3(block_ptr: list, i: int):
    value = block_ptr[i >> 1]
    return Extract(3, 0, value) if (i & 1) else Extract(7, 4, value)


def insert_at_z3(new_buf: list, pos: int, to_ins):
    cur_buf_val = new_buf[pos >> 1]
    if pos & 1:
        new_buf[pos >> 1] = Concat(Extract(7, 4, cur_buf_val), to_ins)
    else:
        new_buf[pos >> 1] = Concat(to_ins, Extract(3, 0, cur_buf_val))


def scramble(shift_elem: int, idx: int) -> int:
    is_odd = idx & 1
    half_to_shift = shift_elem >> 1
    half_idx = idx >> 1
    if (shift_elem & 1) == 1:
        return ((2 * (half_to_shift - half_idx)) & 0xE) - is_odd + 1
    else:
        return 2 * (half_to_shift + half_idx) + is_odd


def scramble_z3(shift_elem, idx: int):
    lut = [scramble(v, idx) for v in range(16)]
    result = BitVecVal(lut[0], 4)
    for v in range(1, 16):
        result = If(shift_elem == v, BitVecVal(lut[v], 4), result)
    return result


if __name__ == "__main__":
    pts = [
        [BitVec(f"x_{i}_{j}", 8) for i in range(8)] for j in range(n_blocks)
    ]
    s = Solver()

    for i, c in enumerate(known_start):
        s.add(pts[0][i] == c)

    for j in range(n_blocks):
        new_buf = [BitVecVal(0, 8) for _ in range(8)]
        cur_block = list(pts[j])
```

```
        for k in range(16):
            new_shifts = [0] * 16
            for i in range(16):
                idx = permutation_idx[i]
                new_shifts[idx] = scramble(shifts[j][k], i)

            for m in range(16):  # permutation
                extracted_char = extract_first_high_then_low_z3(cur_block, m)
                insert_at_z3(new_buf, new_shifts[m], extracted_char)

            for n in range(16):  # keymixing
                extracted_block = extract_first_high_then_low(keys[j], n)
                extracted_intern = extract_first_high_then_low_z3(new_buf, n)
                computed_val = scramble_z3(extracted_intern, extracted_block)
                insert_at_z3(new_buf, n, computed_val)

            for x in range(16):  # substitution
                extract = extract_first_high_then_low_z3(new_buf, x)
                ns_ns = [new_shifts[new_shifts[v]] for v in range(16)]
                mapped = BitVecVal(ns_ns[0], 4)
                for v in range(1, 16):
                    mapped = If(extract == v, BitVecVal(ns_ns[v], 4), mapped)
                insert_at_z3(cur_block, x, mapped)

        for i in range(8):
            s.add(cur_block[i] == ciphertexts[j][i])

    if s.check() == sat:
        model = s.model()
        flag = ""
        for j in range(n_blocks):
            for i in range(8):
                flag += chr(model.evaluate(pts[j][i]).as_long())
        print(flag)
```

The core idea is this: instead of computing $Enc^{-1}(C, K)$, we **rewrite the encryption function using the Z3 API**, replacing concrete plaintext values with symbolic variables, and let the solver find the values that satisfy:

Enc(P_sym, K) = C

The cipher becomes a system of constraints. Z3 finds the assignment of P_sym that satisfies it — that is, the original plaintext.

Two categories of constraints are added:

4. Known plaintext: the flag format tells us the first block starts with `kalmar{`. Seven known bytes that significantly constrain the search space.

5. Ciphertext match: for each block j, after symbolically executing all 16 rounds, the result must match the known ciphertext.

```python
Python
pts = [
    [BitVec(f"x_{i}_{j}", 8) for i in range(8)] for j in range(n_blocks)
]

s = Solver()

for i, c in enumerate(known_start):
    s.add(pts[0][i] == c)

for j in range(n_blocks):
    # ... symbolic encryption ...
    for i in range(8):
        s.add(cur_block[i] == ciphertexts[j][i])
```

We are not asking the solver to “guess” anything. We describe exactly the same transformation the cipher performs, but with variables in place of concrete values. The solver is not doing magic, it is solving a very complex system of equations that we constructed.

4.1 Why reconstruct

The raw data in the file, keys and ciphertexts, is stored as sequences of nibbles, 4-bit values each:

```
[2, 3, 3, 5, 3, 3, 1, 4, ...]
```

The cipher operates on 8-bit bytes, where each byte holds two nibbles: the high nibble (bits 7–4) and the low nibble (bits 3–0). Before doing anything, these nibbles must be reassembled into bytes according to:

$$byte_i = nibble_{2i} \ll 4 \mid nibble_{2i+1}$$

```python
Python
def reconstruct(splitted):
    return [(splitted[i + 1] | (splitted[i] << 4)) for i in range(0, len(splitted), 2)]
```

So `[2, 3]` becomes `0x23`, `[3, 5]` becomes `0x35`, and so on. The data in the file is stored in the same internal nibble representation that the algorithm uses during encryption, and must be reconstructed before being passed to the solver.

This also raises a design decision: should the symbolic variables represent full 8-bit bytes, or 4-bit nibbles? Working at byte level, after calling `reconstruct` first, keeps the code cleaner, the model output is directly readable as a string.

4.2 The Symbolic Index Problem

This is the most subtle part of the entire solution. The problem is general: it appears in any challenge where an input-dependent value is used as an array index.

The third step of each round, **substitution**, works as follows in concrete code: extract a nibble from `new_buf`, use it as an index into `new_shifts` for a double lookup, and write the result to `cur_block`.

```
Python
for x in range(16):
    nibble = new_buf[x]
    result = new_shifts[new_shifts[nibble]]  # double lookup
    cur_block[x] = result
```

When translating this to Z3, we hit a wall. `extract_first_high_then_low_z3(new_buf, x)` does not return a Python integer — it returns a **Z3 expression**, an algebraic symbol without a concrete value. Python cannot use it as a list index:

```
Python
extract = extract_first_high_then_low_z3(new_buf, x)  # Z3 expression, not an int
result = new_shifts[extract]  # TypeError, Python stops here
```

The issue is fundamental: Python performs list indexing at runtime, during Z3 model construction, when symbolic values have no concrete evaluation yet. There is no way to circumvent this directly.

The Solution — Transform the Lookup into an If-Chain

The key observation is that `new_shifts` is **entirely concrete**: it depends only on `shifts[j][k]` and `permutation_idx`, both of which are fixed and known. The only unknown is the value extracted from `new_buf` — the index.

But that index is a 4-bit nibble: it can only take 16 possible values, v ∈ {0, 1, …, 15}. For each of those values, the result of the double lookup is perfectly computable in advance:

```python
ns_ns = [new_shifts[new_shifts[v]] for v in range(16)]
```

`ns_ns` is now a concrete list of 16 integers. The next step is to build a Z3 expression that selects the correct value based on whatever the solver assigns to the symbolic index — which is exactly what an If-chain does:

```python
extract = extract_first_high_then_low_z3(new_buf, x)

ns_ns = [new_shifts[new_shifts[v]] for v in range(16)]

mapped = BitVecVal(ns_ns[0], 4)
for v in range(1, 16):
    mapped = If(extract == v, BitVecVal(ns_ns[v], 4), mapped)
```

The resulting `mapped` expression is equivalent to `ns_ns[extract]`, but expressed entirely in Z3's language:

```
If(extract == 1,  ns_ns[1],
If(extract == 2,  ns_ns[2],
...
If(extract == 15, ns_ns[15],
   ns_ns[0])))
```

The solver reasons over this structure like any other expression: it knows that `mapped` equals `ns_ns[v]` if and only if `extract == v`, and uses this to propagate constraints toward the solution.

Running the script, we successfully recover the flag.

5. Common Gotchas

Translating C code into Z3 is conceptually straightforward, but in practice full of pitfalls. C is rich in implicit behaviors, type promotions, signed/unsigned semantics, silent casts, that Z3 does not replicate automatically. Python's dynamic typing makes silent errors even harder to detect. The following is a collection of the most common issues encountered when working with Z3.

5.1 Width Mismatch Between BitVecs

Z3 does not perform implicit conversions between BitVecs of different widths. Attempting to add a `BitVec(8)` to a `BitVec(32)` raises an exception. In C this would be handled by the compiler's automatic promotion; in Z3 it must be done manually with `ZeroExt` or `SignExt`.

```python
from z3 import *

x = BitVec('x', 8)
y = BitVec('y', 32)

# z = x + y  ->  Z3Exception: sort mismatch

# Correct: widen x to 32 bits before the operation
z = ZeroExt(24, x) + y  # unsigned promotion
z = SignExt(24, x) + y  # signed promotion
```

5.2 Implicit Signed Promotion in C

C promotes `char`, `short`, and any type smaller than `int` to `int` in almost every arithmetic expression, and this promotion is **always signed**, regardless of the original type. If the C code operates on `int8_t` or `char` in mixed expressions, Z3 must replicate that promotion with `SignExt`, otherwise results diverge silently.

```c
// C
int8_t x = -1;       // 0xFF
int result = x * 2;  // Implicit promotion: (int)x * 2 = -2, not 510
```

```python
# Wrong: Z3 works on 8 bits, does not replicate the promotion
x = BitVecVal(0xFF, 8)
result = x * BitVecVal(2, 8)  # 0xFE = 254, not -2

# Correct: replicate the signed promotion to 32 bits
x_promoted = SignExt(24, BitVecVal(0xFF, 8))  # 0xFFFFFFFF = -1 as int32
result = x_promoted * BitVecVal(2, 32)        # 0xFFFFFFFE = -2
```

This error is particularly insidious because the code runs without exceptions but produces results that diverge from the original binary, and the solver finds solutions that do not correspond to reality.

5.3 % and / Are Signed by Default

Like >>, division and modulo on BitVec follow **signed** semantics by default in Z3. If the C code uses unsigned types, results will diverge silently for operands with the high bit set.

```python
from z3 import *

x = BitVecVal(0xFF, 8)  # 255 unsigned, -1 signed

print(simplify(x % 10))        # 9 , bvsmod (floor, Python-style): -1 mod 10 = 9
print(simplify(SRem(x, 10)))  # -1, bvsrem (C-style truncation): result has sign of
dividend
print(simplify(URem(x, 10)))  # 5 , unsigned: 255 % 10 = 5
```

An important distinction: in Z3, % on BitVec corresponds to `bvsmod` — **floor** semantics (Python-style), where the result always has the sign of the divisor. This is not `bvsrem`, which is C's **truncation** semantics where the result has the sign of the dividend. To replicate C's % on signed values exactly, use SRem.

The complete operator reference:

C	Z3 (wrong)	Z3 (correct)
`>>` unsigned	x >> n	LShR(x, n)
`/` unsigned	x / n	UDiv(x, n)
`%` unsigned	x % n	URem(x, n)
`%` signed (C)	x % n	SRem(x, n)
`<` unsigned	x < y	ULT(x, y)
`>` unsigned	x > y	UGT(x, y)
`<=` unsigned	x <= y	ULE(x, y)
`>=` unsigned	x >= y	UGE(x, y)

5.4 Python Constants Mixed with BitVec

```
Python
from z3 import *

x = BitVec('x', 8)

# Appears to work, but internal conversions are not always predictable
y = x ^ 0xFF

# Explicit and safe form
y = x ^ BitVecVal(0xFF, 8)
```

The rule is simple: when working with `BitVec`, **always wrap constants** with `BitVecVal(value, width)`. It adds verbosity but eliminates an entire class of silent bugs.

5.5 Duplicate Variable Names

In Z3, the name passed to `BitVec('x', 8)` is not just a label — it is the **identifier** of the variable in the constraint system. Two declarations with the same name are the **same variable**, not two independent ones.

```python
from z3 import *

# These are NOT two independent bytes, they are the same variable
a = BitVec('byte', 8)
b = BitVec('byte', 8)

s = Solver()
s.add(a == 0x41)
s.add(b == 0x42)
print(s.check())  # unsat, the same variable cannot be both 0x41 and 0x42

# Correct: distinct names
a = BitVec('byte_0', 8)
b = BitVec('byte_1', 8)
```

This error commonly appears when generating variables in a loop using static names instead of indexed ones.

5.6 Bool vs BitVec(1)

In Z3, Bool and `BitVec(1)` are **distinct types** and are not interchangeable. A `BitVec(1)` cannot be used directly as a condition in `If`, And, `Or`, or `s.add()` — Z3 expects a `Bool`.

```python
from z3 import *

x = BitVec('x', 8)
flag = BitVec('flag', 1)  # Not a Bool!

# Wrong
result = If(flag, x, BitVecVal(0, 8))  # Z3Exception

# Correct: explicit comparison
result = If(flag == 1, x, BitVecVal(0, 8))
```

This error often appears when modeling C code that uses single bits or `int` as boolean flags. In C, any nonzero value is truthy; in Z3, the comparison must be made explicit.

5.7 Unconstrained Variables in the Model

If a symbolic variable is not sufficiently constrained, Z3 returns `None` from the model, with no exception and no warning. The code either crashes silently or produces empty output.

```python
Python
from z3 import *

x = BitVec('x', 8)
y = BitVec('y', 8)
s = Solver()
s.add(x == 42)
# y has no constraints

if s.check() == sat:
    m = s.model()
    print(m[x].as_long())  # 42
    print(m[y])            # None, y is unconstrained
    print(m[y].as_long())  # AttributeError: 'NoneType'
```

The correct form is to use `m.evaluate(y, model_completion=True)`, which assigns an arbitrary concrete value to free variables instead of returning the symbolic expression itself:

```python
Python
print(m.evaluate(y, model_completion=True).as_long())  # Some value, e.g. 0
```

5.8 Non-Linear Arithmetic

Z3 distinguishes sharply between **linear arithmetic**, where every variable appears with a constant coefficient, and **non-linear arithmetic**, where products of symbolic variables appear, such as $x \cdot y$.

Linear arithmetic is decidable and fast. Non-linear arithmetic over integers is **undecidable** in general — no procedure is guaranteed to always terminate with a correct answer. Z3 handles it with heuristics but may return unknown or fail to terminate.

```python
from z3 import *

x = Int('x')
y = Int('y')
s = Solver()

# Linear: fast and decidable
s.add(x + y == 10)
s.add(2*x - y == 4)  # 2 is a constant, not a variable

# Non-linear: slow, potentially non-terminating
s.add(x * y == 42)   # product of two symbolic variables, use with caution
```

On `BitVec`, unlike mathematical integers, non-linear arithmetic is **decidable** because the value space is finite and the solver can reduce everything to SAT via bit-blasting. The concern is complexity: each additional bit can double the number of generated clauses. A symbolic 64-bit multiplication between two unknown variables can make solving times prohibitive.

The strategy is to make as many operands **concrete** as possible: if one factor is known, the multiplication becomes linear and returns to being fast.

5.9 Large Constraints vs. Many Small Constraints

This is one of the less obvious performance pitfalls, documented in real CTF writeups. When modeling a complex algorithm, it is tempting to build a single large expression representing the entire computation and add it to the solver in one shot. In practice, this often does not terminate.

The reason is that Z3 performs significantly better with **many small, simple constraints** than with a few large, complex ones, even when the two approaches are mathematically equivalent. Smaller constraints allow the solver to propagate information, prune the search space, and detect conflicts earlier.

```python
# Approach 1: a single constraint representing the entire computation
s.add(encrypt_symbolic(message, unknown_constants) == known_ciphertext)
# -> The solver may not terminate

# Approach 2: one constraint per round, with intermediate assertions where possible
for round_idx in range(16):
    state = round_step_symbolic(state, round_keys[round_idx])
    if round_idx == 7:
        s.add(state[0] == known_intermediate_value)  # Guides the solver
# -> Much faster
```

It is always worth asking whether constraints can be broken into smaller pieces, or whether intermediate values at specific rounds can be constrained when known.

5.10 Symbolic Indices into Concrete Structures

Whenever the original code contains `list[value]` and that `value` becomes symbolic in Z3, Python will raise a `TypeError` at runtime. The solution is to precompute all possible outputs for every concrete value of the index and build an If-chain, as detailed in the solution section above.

```
Python
# Concrete list, symbolic index
lut = [concrete_list[v] for v in range(16)]  # Precompute all 16 outputs

result = BitVecVal(lut[0], 4)
for v in range(1, 16):
    result = If(symbolic_index == v, BitVecVal(lut[v], 4), result)
```

6. Conclusion

Z3 performs well on linear problems and on bit-vectors of reasonable size, but degrades rapidly outside those cases. Non-linear arithmetic over integers is undecidable, loops require manual unrolling, and complexity explodes with the number of symbolic variables. Its real potential emerges when used as the backend of larger systems — such as **angr** for symbolic execution over full binaries and **Triton** for concolic execution — which are built precisely to mitigate these limitations.

6.1 The Current Ecosystem

In production, raw Z3 is rarely used in isolation. The ecosystem has consolidated around frameworks such as angr, Triton, **and** Binsec**, which manage operational complexity while leaving Z3 in the role of constraint solver. In malware analysis, the most documented applications are automatic MBA deobfuscation — where tools such as Quarkslab's** SSPAM use Z3 to verify equivalence between an obfuscated expression and its simplified form — and the recovery of custom encryption keys used by ransomware and loaders to protect C2 configurations and internal strings.

Next | Donexit | Foramil | Innodesi

Via Giacomo Peroni, 452 – 00131 Roma
tel. 06.45752720 – info@defencetech.it
www.tinextadefence.it

#TDefenceBusiness