Lecture Notes on Safety Analysis

15-316: Software Foundations of Security & Privacy
Matt Fredrikson

Lecture 9
February 10, 2026

1 Introduction

Lab 1 asks you to build a safety checker for a subset of C0. The structure of the tool is simple: you take a program, calculate a weakest liberal precondition, and obtain a verification condition that is a formula in a logic of arithmetic and arrays. The verification condition is the place where all of the work of the safety analysis shows up. If the verification condition is valid, then the original program is safe. If it is not valid, then there is a counterexample, and the safety checker should be able to report a concrete state in which the program can go wrong.

In this lecture we focus on the solver side of the story. Z3 is an SMT solver, meaning it decides satisfiability of formulas over a collection of theories such as machine arithmetic and arrays [Moura and Børner, 2008]. The key point for us is that satisfiability is not quite the same question as validity. A safety checker is usually trying to establish that a verification condition holds for all initial states. Z3, by contrast, is optimized to find one state that makes a formula true. The bridge between these two viewpoints is a short and practical reduction that we can implement directly in Z3Py, Z3's Python library.

2 Introduction to Z3

Z3 answers satisfiability questions. We write down a formula F with free variables, and we ask whether there exists an assignment to those variables that makes F true. If Z3 answers `sat`, it produces a model, which is a concrete assignment that witnesses satisfiability. If it answers `unsat`, then no assignment exists. Sometimes Z3 answers `unknown`, which means it gave up under the strategy and resource limits it chose for that query.

In Z3Py, the basic workflow is to create a solver, add constraints, and call the solver's `check` method. When the result is `sat`, we can ask for the model.

```
import z3

x, y = z3.BitVecs("x y", 64)
s = z3.Solver()
s.add(x + y == 10)
s.add(x > y)
print(s.check())
print(s.model())
```

There are two practical details that matter immediately for program analysis. First, each call to `s.add` conjoins another constraint, so the solver is always working with one big conjunction. Second, this entire interaction is existential. When Z3 says `sat`, it has found one state that satisfies your constraints.

Verification conditions, however, are validity questions. We are trying to show that VC holds in every state under consideration. Z3 does not directly check validity, so we reduce it to satisfiability: VC is valid exactly when $\neg VC$ is unsatisfiable. In practice this means we negate the verification condition, hand it to Z3, and interpret `unsat` as success. When Z3 answers `sat`, the model is a counterexample state that explains why the verification condition fails.

It is worth writing this reduction out because we will use it repeatedly.

```
def check_validity(F):
    s = z3.Solver()
    s.add(z3.Not(F))
    res = s.check()
    if res == z3.unsat:
        return True, None
    if res == z3.sat:
        return False, s.model()
    return None, None
```

The return value distinguishes the two outcomes that matter most for us. If the result is `unsat`, then no counterexample exists and the formula is valid. If the result is `sat`, then the model gives a concrete counterexample. The `unknown` outcome is a conservative failure mode that can arise on harder queries.

Here is a tiny example that behaves the way you want a verification condition to behave. Over mathematical integers, the implication $x > 0 \rightarrow x + 1 > 0$ is valid.

```
x = z3.Int("x")
F = z3.Implies(x > 0, x + 1 > 0)
print(check_validity(F))
```

If we change the postcondition slightly, the verification condition becomes false. Z3 can then produce a counterexample state.

```
x = z3.Int("x")
F = z3.Implies(x > 0, x - 1 > 0)
```

```
print(check_validity(F))
```

When you build a safety checker, this is exactly what you want from the solver. If your weakest liberal precondition calculation is correct, then a satisfying model of the negated verification condition corresponds to an initial state that leads to a safety violation. Z3 is then doing two jobs at once: it is acting as a decision procedure when the answer is `unsat`, and it is acting as a counterexample generator when the answer is `sat`.

One subtlety is that Z3's theories often define operations as total functions, even when your programming language treats them as unsafe. Division by zero is an easy example. An array read outside bounds is another. Z3 will still assign meaning to those expressions, so it is the responsibility of the weakest liberal precondition to generate extra proof obligations that rule out the unsafe cases. This is why, as we have seen in previous lectures, safety verification conditions tend to look like a conjunction of two kinds of claims: one kind expresses the intended postcondition, and the other kind expresses that all of the intermediate operations are used safely.

3 Machine Integers

It is easy to accidentally verify the wrong program by modeling the wrong arithmetic. Many reasoning steps are valid over mathematical integers and fail over machine integers. C0 uses fixed-width machine integers. In Z3, the natural way to model this is with bitvectors. When you use bitvectors, every arithmetic operation wraps around on overflow, and comparisons must be interpreted consistently as signed or unsigned.

The difference shows up immediately in a one-line formula. Over integers, it is impossible that $x + 1 < x$. Over bitvectors, it becomes satisfiable because the addition wraps around at the maximum representable value.

```
x = z3.Int("x")
print(z3.Solver().check(x + 1 < x))

w = 8
x = z3.BitVec("x", w)
s = z3.Solver()
s.add(x + 1 < x)
print(s.check())
print(s.model())
```

This example is more than a curiosity. It tells you exactly what can go wrong if you compute a weakest liberal precondition under one arithmetic and then ask Z3 to prove it under a different arithmetic. The solver will happily prove facts about integers that are not facts about machine integers, and your safety checker will become unsound.

The classic binary search bug gives a concrete illustration. Many programmers compute the midpoint by adding the bounds and dividing by two.

```
mid = (low + high) / 2;
```

Over mathematical integers, the property you want is valid: if $0 \leq low \leq high$, then $low \leq \frac{low+high}{2} \leq high$. Over machine integers, the sum $low + high$ can overflow, and the midpoint can escape the interval even when the bounds are nonnegative.

Z3 can show you the difference directly. The following query asks whether there is a counterexample to the midpoint property under mathematical integers.

```
low, high = z3.Ints("low high")
mid = (low + high) / 2
F = z3.Implies(z3.And(0 <= low, low <= high),
               z3.And(low <= mid, mid <= high))
print(check_validity(F))
```

If we keep the same shape but switch to bitvectors, Z3 can produce a concrete counterexample due to overflow.

```
w = 32
low, high = z3.BitVecs("low high", w)
zero = z3.BitVecVal(0, w)
mid = (low + high) / 2
F = z3.Implies(z3.And(zero <= low, low <= high),
               z3.And(low <= mid, mid <= high))
print(check_validity(F))
```

The standard fix for binary search is to rewrite the midpoint computation so that the subtraction happens before the addition.

```
mid = low + (high - low) / 2;
```

The subtraction reduces the range before the addition, which prevents the overflow in the common case where $low \leq high$. Again, you can use Z3 as a sanity check.

```
w = 32
low, high = z3.BitVecs("low high", w)
zero = z3.BitVecVal(0, w)
mid = low + (high - low) / 2
F = z3.Implies(z3.And(zero <= low, low <= high),
               z3.And(low <= mid, mid <= high))
print(check_validity(F))
```

In your Lab 1 implementation, the important discipline is consistency. Decide which operations are signed, which are unsigned, and which bitwidth you are using, then use that same interpretation in both the weakest liberal precondition rules and the Z3 encoding. Otherwise you will end up proving a formula about a different program than the one you intend to analyze.

4 Modeling arrays in Z3

Our discussion of memory safety in Lecture 6 introduced the theory of arrays as a way to reason about memory updates without inventing a completely new logical language. In that lecture, a heap behaves like a map from indices to values, and we added two terms for interacting with it, namely **read** $H\ i$ and **write** $H\ i\ a$. The intended meaning was that **write** $H\ i\ a$ denotes a new heap that agrees with H everywhere except at index i, where it has value a. This is the same idea as writing $H[i \mapsto a]$ as a functional update term. Reads are then lookups in this map, and safety is enforced by requiring that any address used for a read or write is within the allocated bounds.

Z3's built-in theory of arrays matches this abstraction very closely. A Z3 array is a total function from an index sort to a value sort. The operation `Select` plays the role of **read**, and `Store` plays the role of **write**. A good way to remember the correspondence is to read the Z3 expressions out loud using our lecture notation. In the notation of Lecture 6, we can identify **write** $H\ i\ a$ with the functional update term $H[i \mapsto a]$. Then a Z3 read after a write looks like this.

```
z3.Select(z3.Store(H, i, a), i)
```

This corresponds to the term **read** $H[i \mapsto a]\ i$, and the theory guarantees the defining equations we relied on in Lecture 6, such as **read** $H[i \mapsto a]\ i = a$ and, when $i \neq j$, **read** $H[i \mapsto a]\ j =$ **read** $H\ j$.

You can ask Z3 to validate these core properties directly, and it is useful to write them first in the vocabulary of **read** and **write** and then translate them into `Select` and `Store`.

```
w = 64
Idx, Val = z3.BitVecSort(w), z3.BitVecSort(w)
A = z3.Array("A", Idx, Val)
i = z3.BitVec("i", w)
j = z3.BitVec("j", w)
v = z3.BitVec("v", w)

F1 = z3.Select(z3.Store(A, i, v), i) == v
F2 = z3.Implies(i != j,
            z3.Select(z3.Store(A, i, v), j) == z3.Select(A, j))
print(check_validity(F1))
print(check_validity(F2))
```

This gives you a useful mental model for what Z3 is doing when it reasons about memory. The most important difference is that Z3 does not know anything about allocation or bounds. A Z3 array is total, so `Select` is defined for every index. Safety is therefore something you must encode yourself. When you analyze a program statement like `x = a[i];`, your weakest liberal precondition must generate obligations like $0 \leq i$ and $i < |a|$.

The simplest way to represent a bounded array is to pair the element map with a separate length. This is exactly the style used by the Lab 1 starter: an array value is represented by a Z3 value that contains a length field and a data field for the underlying array map. The length lets you express bounds obligations, and the data lets you use `Select` and `Store` for element access.

```
BIT_WIDTH = 64
IntSort = z3.BitVecSort(BIT_WIDTH)
ArrDataSort = z3.ArraySort(IntSort, IntSort)

Arr = z3.Datatype("Arr")
Arr.declare("mk", ("len", IntSort), ("data", ArrDataSort))
Arr = Arr.create()

def arr_read(a, i):
    return z3.Select(Arr.data(a), i)

def arr_write(a, i, v):
    return Arr.mk(Arr.len(a), z3.Store(Arr.data(a), i, v))
```

This encoding is not the only way to do it. You could represent the pair using two separate Z3 variables rather than a datatype. You could also axiomatize reads and writes with uninterpreted functions if you wanted the solver to treat the array contents more abstractly. The essential ingredients do not change. You need a structure that behaves like functional update for the contents, and you need explicit obligations that express which indices are in bounds.

Real C0 programs also have aliasing. Two variables can refer to the same array object, and then a write through one variable changes what the other variable will read. Lab 1 does not ask you to handle aliasing, but it is useful to know what changes in the encoding if you want to. To model this, we introduce a heap and make array values be pointers into that heap. The heap is itself a Z3 array type that maps each pointer to its current array contents. This is essentially a two-level array: the outer map goes from pointers to inner arrays, and the inner arrays go from indices to values.

Here is a simple C0 example that depends on aliasing. The assignment `b = a` does not copy an array. It makes `b` refer to the same array object as `a`. After that, a write through `b` must be observable through `a`.

```
int[] a = alloc_array(int, 1);
int[] b = a;

b[0] = 7;
assert(a[0] == 7);
```

If you try to model arrays without a heap, it is tempting to treat an array variable as if it were just its contents and its length. In that setting, updating `b[0]` produces a new array value for `b` and leaves `a` unchanged. That is not what the C0 semantics says. This is the point where the heap model becomes necessary.

```
w = 64
Ptr = z3.DeclareSort("Ptr")
Idx = z3.BitVecSort(w)
Val = z3.BitVecSort(w)

HeapData = z3.ArraySort(Ptr, z3.ArraySort(Idx, Val))

def heap_read(heap_data, p, i):
    return z3.Select(z3.Select(heap_data, p), i)

def heap_write(heap_data, p, i, v):
    a = z3.Select(heap_data, p)
    a2 = z3.Store(a, i, v)
    return z3.Store(heap_data, p, a2)
```

With this structure, aliasing is automatic. If two program variables contain the same pointer, then they refer to the same heap entry and therefore see each other's writes. Notice that the pointer sort is not a machine integer sort, as C0 does not treat arrays that way. In the logic, the only primitive operation on pointers is equality, which is exactly what an uninterpreted sort provides.

5 Summary

In Lab 1, Z3 is the component that discharges the arithmetic and array reasoning that remains after we calculate a weakest liberal precondition. The weakest liberal precondition is where the semantics of the language are expressed, and it is also where the safety side conditions are introduced. Z3 is then used to check validity by searching for counterexamples to those conditions. This is why it is so important to keep the semantics aligned. If your WLP is reasoning about machine integers, your Z3 encoding must use bitvectors at the same width, and you must be consistent about signed and unsigned operations. If your WLP is reasoning about bounded arrays, your Z3 encoding must model array contents with `Select` and `Store` while separately carrying length information so that you can generate and discharge bounds obligations. When these pieces match, the solver's counterexamples line up with real program executions and the analysis remains sound.

References

Leonardo De Moura and Nikolaj Børner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340, Budapest, Hungary, mar 2008. Springer LNCS 4963.