Singularity

The closer your program gets to a singularity, the slower it runs.
— general theory of relativity

Observations

- **Many complexity vulnerabilities can be triggered by inputs with special patterns**
  - a sorted array
  - a graph with negative-weight loops
  - objects with same hash value
  - . . .

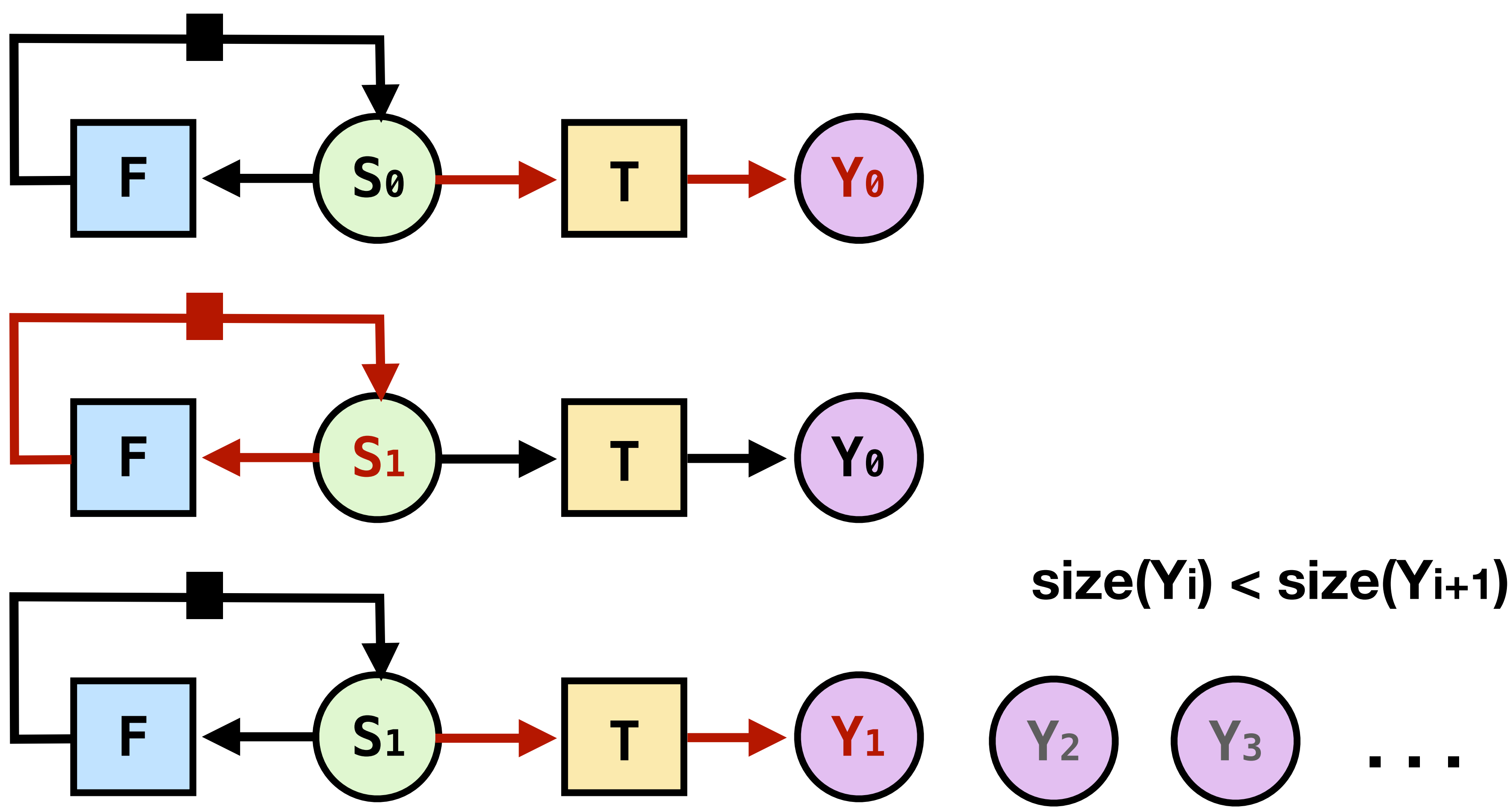
- **Fuzzing large inputs with such properties can be challenging because of the huge search space**
  - the space of possible inputs grows exponentially with size

- **Instead of concrete inputs, we can search for input patterns**

- **We developed *Singularity*, a pattern fuzzer for complexity vulnerabilities**

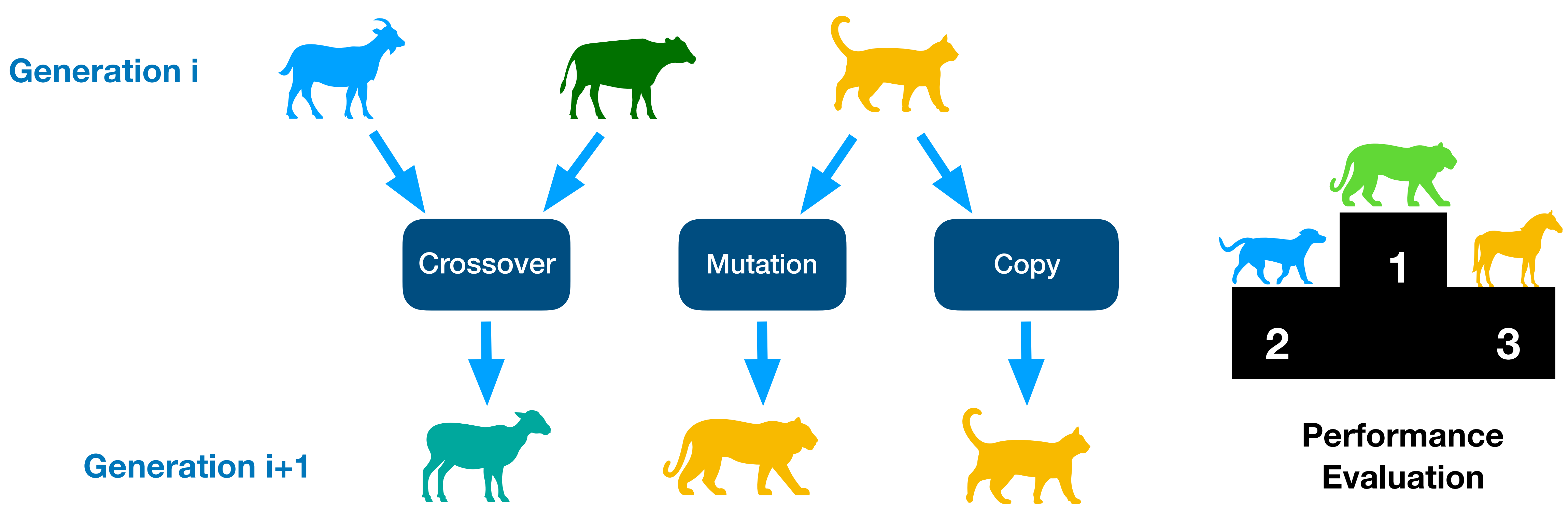
Input Patterns

- **We can represent input patterns as programs**
  - Starting state $\mathbf{S_0}$, Transformer **T**, Updater **F**
  - One pattern can be used to construct inputs of *arbitrarily large size*
  - Represent $\mathbf{S_0}$, **T**, **F** as DSL expressions and search for them instead
  - **S** and **Y** can be tuples containing multiple values



Resource Usage Maximization

- **Search for patterns that can maximize resource usage**
  - To evaluate a pattern, we use it to generate an input of some specified size, and measure the corresponding resource usage (running time, space usage, e.t.c.)
  - A pattern with the highest resource usage will be able to trigger any potential complexity vulnerability
  - To search for patterns with maximal resource usage, we use genetic programming as the optimization algorithm



Solving SearchableBlog

- **In the recent engagement, Singularity helped us solve the *SearchableBlog* problem (Q27)**
  - We need to find a vector V that maximizes the time used by the following code

```
Vector v1 = MatrixRoutines.processVector(V, 0.999);
Matrix A = MatrixRoutines.concatenateColumn(ajMatrix, v1);
estimateStationaryVector(A, bVector, 1.0E-7);
```

  - *estimateStationaryVector* is a numerical iterative algorithm whose running time has complex dependency on the initial vector *V* and the given matrix *ajMatrix*
  - Singularity found the correct pattern within 5min and used it to construct following vector, whose length is 730!

[0,0,0,0,0,0,…,0,1]

  - Initial state: empty vector **[ ]**
  - Updater: **λv. append(v, 0)**
  - Transformer: **λv. append(v, 1)**
  - First few elements:

[1], [0,1], [0,0,1], [0,0,0,1], ..

Solving Braidit

- **Singularity also helped us solve the *Braidit* problem (Q17 Q22)**
  - To trigger the time or space vulnerability, we needed to construct inputs that can cause **Weave.isEquivalent** method run a long time
  - The inputs correspond to two strings (S1, S2) and one integer (n)
  - We let Singularity maximize the running time of the following code

```
val w1 = new Weave(S1, n)
val w2 = new Weave(S2, n)
w1.isEquivalent(w2)
```

  - It found an efficient pattern within 5min, which gives the following inputs for string length less than 50

```
S1 =  “thYPGxofWNEvmdULCtkbSJAriZQHypgXOFwneVMDulcTKBsja"
S2 = "qykYPGxofWNEvmdULCtkbSJAriZQHypgXOFwneVMDulcTKBsja"
n = 27
```

  - Note that the two strings are not equivalent but share a common part. Patterns like this can be challenging for other non-pattern-based fuzzers.

How it works

- Suppose we want to maximize the running time of a QuickSort implementation which always select the middle element in the array as pivot

```
def quick_sort(xs: list):
    if length(xs) <= 1:
        return xs
    pivot = xs[length(xs) // 2]
    left, middle, right = [], [], []
    for x in xs:
        if x==pivot:
            middle.append(x)
        elif x<pivot:
            left.append(x)
        else:
            right.append(x)
    left = quick_sort(left)
    right = quick_sort(right)
    return concat(left, middle, right)
```

How it works

- **Suppose we want to maximize the running time of a QuickSort implementation which always select the middle element in the array as pivot**
  - One way to achieve this is using the following pattern:

```
[ ],[1,0],[3,1,0,2],[5,3,1,0,2,4],...
```

  - Which can be represented as
    - initial state: **[ ]**
    - transformer: **identity**
    - updater: **λx.*append*(*prepend*(*length*(x) + 1, x), *length*(x))**

  - To find the desired pattern, Singularity starts with a population of randomly-generated input patterns
  - For simplicity, assume init states and transformers are already given
  - It is very unlikely that Singularity will discover the correct updater in the this first generation, but it will discover useful sub-optimal updaters. e.g.

```
f1 = λx.append(x,length(x))
f2 = λx.prepend(length(x),x)
..
```

How it works

- Desired updater:

```
λx.append(prepend(length(x) + 1, x), length(x))
```

- Updaters found in first population:

```
f1 = λx.append(x,length(x))
f2 = λx.prepend(length(x),x)
..
```

- For the next population, we randomly pick input patterns with high resource usage from the previous population
- f1 and f2 are likely to be selected because they have higher than average resource usage
- We then use these input patterns to generate a new population by combining them using genetic operators, such as mutation and crossover
- For example, we can obtain the following updater f3 from f1 and f2 by using the crossover operation

```
f3 = λx.append(prepend(length(x),x),length(x))
```

- f3 has higher resource usage than f1 and f2, but still sub-optimal

How it works

- Desired updater:

```
λx.append(prepend(length(x) + 1, x), length(x))
```

- Updaters found:

```
f1 = λx.append(x,length(x))
f2 = λx.prepend(length(x),x)
f3 = λx.append(prepend(length(x),x),length(x))
..
```

- We continue the process of generating new populations and monitor their average performance
- In general, average performance will keep increasing over generations
- At some point, Singularity will generate the desired updater by mutating the sub-expression in f3:

```
f3 = λx.append(prepend(length(x),x),length(x))

f4 = λx.append(prepend(length(x)+1,x),length(x))
```

- Hence, we found the pattern and can use it to construct an attack vector of arbitrarily large size