# Data Poisoning: Lightweight Soft Fault Injection for Python

Mohammad Amin Alipour, and Alex Groce

Oregon State University

**Abstract.** This paper introduces and explores the idea of data poisoning, a light-weight peer-architecture technique to inject faults into Python programs. This method requires very small modification to the original program, which facilitates evaluation of sensitivity of systems that are prototyped or modeled in Python. We propose different fault scenarios that can be injected to programs using data poisoning. We use Dijkstra's Self Stabilizing Ring Algorithm to illustrate the approach.



## 1 Introduction

Designing dependable software is notoriously hard. In designing critical systems, designers should continuously evaluate the dependability of their design in the presence of potential faults or failures. Fault injection tools emulate potential problems in the environment that a system will operate within [5], by injecting an error state into the running program. We note that given the traditional meanings of fault, error, and failure in software testing, error injection is possibly a better name for this method.

Broadly speaking, there are two primary fault injection techniques: hardware-based fault injection and software-based fault injection. In the first, additional hardware is used to inject faults. This hardware introduces an error via contact with a target circuit by changing voltage or current, or uses radiation or some other physical phenomena to introduce a fault.

Software-based fault injection approaches use code instrumentation or runtime injection. In the code instrumentation approach, additional code is inserted into source code to emulate faults. In runtime injection, events in the computational environment such as interrupts and timeouts are used to introduce the fault. In this paper, we introduce a software-based approach, called *data poisoning*, for fault injection exploiting dynamism in the dispatching mechanism of the Python programming language for fault injection.

Python is a popular programming language for fast prototyping. There are growing number of libraries in Python that help developers to prototype or model system in Python: e.g., MyHDL for hardware design, SimPy for discrete event simulation, Kairos [4] for programming sensor networks, and Pymote for simulation of distributed algorithms.

The Python data model abstracts all data as objects [1]. It also simplifies making changes in the basic behavior of objects. This flexible data model has

**Table 1.** Notation Guide

| Notation | Meaning |
|---|---|
| $\partial(v)$ | `true`, if $v$ is poisoned,<br>`false`, otherwise |
| $uses(S, v)$ | `true`, if statement $S$ uses $v$<br>`false`, otherwise |
| $dev(S, v)$ | `true`, if poisoned $v$ that is used in $S$ has manifested deviant behavior<br>`false`, otherwise |

been utilized to create dynamic symbolic execution engines for Python [3,2], in which concrete objects in the program are replaced with proxy objects that carry out symbolic execution in addition to concrete computation.

*Data poisoning* exploits the flexibility of the Python data model to inject faults with minimal source code changes and a degree of control over how faults and injected and propagated that greatly exceeds that of traditional fault injection tools. The contributions of this short paper are: (1) a definition of data poisoning and (2) an implementation of data poisoning for Python.

## 2 Data Poisoning

In this section, we define data poisoning and provide some examples to clarify the core concepts.

Table 1 shows the notation that we use in the remained of this paper. Suppose a program `Prog` in `SSA` form. That is, all variables in `Prog` are assigned to only once. Predicate $uses(S, v)$ denotes whether statement $S$ uses variable $v$. For example, if $s =$ `velocity = v0 + acceleration*t` and $v =$`acceleration`, predicate $uses(s, v)$ holds, because `acceleration` is used in $S$.

$dev(S, v)$ denotes deviation in the behavior of an operator of a variable $v$ in statement $S$. For example, if the result of multiplication `*` in $s$ is 1% more than the correct value, $dev(s, acceleration)$ holds. Note that $dev(S, v) \implies uses(S, v)$.

**Definition 1 (Data Poisoning)** *Variable $v$ is poisoned, or $\partial(v)$, if and only if there is a statement $s$ in `Prog` such that if $s$ is executed long enough the probability of $dev(s, v)$ approaches 1.*

## 3 Implementation

In this section, we describe the architecture for data poisoning. We also use an example to illustrate data poisoning at work.

Figure 1 illustrates the peer architecture for data poisoning. Target objects are replaced with corresponding poisoned proxy objects. Responsibility of proxy objects is to receive program's operator callbacks and depending on fault injection configurations process them.

### 3.1 Example

Self-stabilizing systems, when perturbed to an error state, are able to converge to a non-error state in a finite number of steps. Such systems are of particular interest in distributed fault-tolerant systems. Designing such systems is not trivial and using fault injection can uncover problems in designs before designers attempt to prove correctness of the design [7].

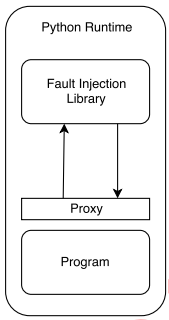


**Fig. 1.** Architecture

Dijkstra assumes a ring network with $N+1$ nodes numbered as $0, .., N$. Each node can be in one of $K$ states, where $K > N$. In a non-error state only one node has the privilege to transition to a new state. The privilege can be seen as a token in the system for performing an action, and there must be only one token at a time in the system; furthermore, all nodes must eventually get the privilege. Making this system self-stabilizing means that if at any time there are 0 or $> 1$ tokens in the system, they system should eventually move to the state with only one token in the system. A fault injection tool and rapid prototyping can help designers develop and refine such algorithms.

Suppose $S$ shows the current state of a node $i$ and $L$ is the current state of its left node, Dijkstra's algorithm for this problem can be described as below:

$$n = 0 \wedge L = S \implies S = (S + 1)\%K$$
$$n \neq 0 \wedge L \neq S \implies S = L$$

The key to understanding this algorithm is to observe the behavior of the algorithm when the state of system is perturbed to an error state.

Figure 2 shows snippets of implementation of Dijkstra's algorithm. The `Node` class represents nodes. `status` shows the current status of an instance class of `Node`. It is initialized to `0`. Function `update` updates the status of the node based on the algorithm. In `update`, whenever a node can make a transition to new state, it obtains the privilege. To observe the state of the system, it calls function `out()` from `Network` class. `Network` class contains a list of nodes in the network. `out` returns a string that represent the state of all nodes in the network. It gets the state of each node, by calling its `MON__hasPrivilege` function. Our tool considers the functions that their names start with `MON__` as the monitoring/reporting functions and disables the data poisoning in them.

Given the definition of class `Node`, one can write a small simulator as in Figure 3. This simulator creates a network of size `5` and `10` times updates the entire network. The result of this simulation is:

```
1,0,0,0,0
0,1,0,0,0
0,0,1,0,0
0,0,0,1,0
0,0,0,0,1
```

```
class Node:
  nodecount = 0
  def __init__(self):
    self.status = 0 # Poison(0, size=1, rate=1, cascade=True)
    self.nid = Node.nodecount
    Node.nodecount += 1
    self.privilege = False

  def MON__hasPrivilege(self):
    L = self.getLeft()
    S = self.status
    if self.nid == 0:
      return L == S
    else:
      return L != S

  def update(self):
    L = self.getLeft()
    S = self.status
    self.privilege = False
    if self.nid == 0 and L == S:
      self.privilege = True
      print(Network.out())
      self.privilege = False
      self.status = (self.status + 1) % K
    if self.nid != 0 and L != self.status:
      self.privilege = True
      print(Network.out())
      self.privilege = False
      self.status = L
```

**Fig. 2.** Dijkstra's Self-Stabilizing Algorithm

```
1,0,0,0,0
0,1,0,0,0
```

Each line in the output shows the overall states of nodes in the network. `1` denotes a node has privilege. In the absence of state perturbation the invariant holds in this run. To inject faults with data poisoning, the designer only needs to import the data poisoning library and replace one statement in Figure 2:

```
self.status =  Poison(0, size=1, rate=1,cascade=True)
```

The `Poison` class implements data poisoning.

## 4 Different types of data poisoning.

The architecture of data poisoning facilitates fine-grained control of fault injection. In this secton, we describe three controllable dimensions of data poisoning. We use Hoare triplets $\{P\}S\{Q\}$ to define the semantics of data poisoning, where $\{P\}$ denotes predicates true before execution of statement $S$ and $\{Q\}$ denotes predicates true after execution of $S$. $S$ denotes basic expressions, i.e. assignment expressions, or conditional expressions that are used in control-flow statements.

**Determinism in Effect of Poisoning.** Poisoned data may either always deviate, or deviate with some probability.

```
NetworkSize = 5
SIMULATIONROUNDS=10
for i in range(NetworkSize):
    n = Node()
    Network.add(n)
for i in range(SIMULATIONROUNDS):
    for nid in range(NetworkSize):
        Network.nodes[nid].update()
```

**Fig. 3.** Sample simulator

**Definition 2 (Deterministic Effect Poisoning)** $\forall S : uses(S, v) \wedge \partial(v) \in P \implies dev(S, v) \in Q$.

**Definition 3 (Intermittent Effect Poisoning)** $\forall S : uses(S, v) \wedge \partial(v) \wedge p \in P \implies dev(S, v) \in Q$, *where* $p$ *is a predicate with a random Boolean value.*

**Lifetime of Poisoned Data.** Poisoned data may either always be poisoned or have a single (generalizable to finite counts) poisoning.

**Definition 4 (Always Poisoned Data)** $\forall S : \partial(v) \in P \implies \partial(v) \in Q$.

**Definition 5 (Transiently Poisoned Data)** $\forall S : \partial(v) \in P \wedge uses(S, v) \implies \neg\, \partial(v) \in Q$.

**Infectiousness of Poisoned data.** Poisoned data can be defined as infectuous, causing poisoning for variables derived from its value, similar to tainting [8,6].

**Definition 6 (Infectious poisoning)** *For an assignment expression,* `x = u binOp v`: $\partial(u) \in P \vee \partial(v) \in P \implies \partial(x) \in Q$, *where* `binOp` *is a binary operation.*

**Definition 7 (Non-infectious poisoning)** *For an assignment expression,* `x = u binOp v`: $\partial(u) \in P \vee \partial(v) \in P \neg \implies \partial(x) \in Q$, *where* `binOp` *is a binary operation.*



```
if self.nid== 0:
    self.status =  Poison(0, infection_size=1, infection_rate=1,cascading=False)
else:
    self.status = 0
```

**Fig. 4.** Non-infectious Transiently

## References


1. Python Data Model. In: https://docs.python.org/2/reference/datamodel.html
2. Ball, T., Daniel, J.: Deconstructing dynamic symbolic execution. Dependable Software Systems Engineering 40, 26 (2015)
3. Bruni, A., Disney, T., Flanagan, C.: A peer architecture for lightweight symbolic execution. In: https://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf
4. Gummadi, R., Kothari, N., Govindan, R., Millstein, T.: Kairos: A macro-programming system for wireless sensor networks. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles. pp. 1–2. SOSP '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1095810.1118600
5. Hsueh, M.C., Tsai, T.K., Iyer, R.K.: Fault injection techniques and tools. Computer 30(4), 75–82 (1997)
6. Newsome, J.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the Network and Distributed System Security Symposium (2005)
7. Perner, M., Sigl, M., Schmid, U., Lenzen, C.: Byzantine self-stabilizing clock distribution with hex: Implementation, simulation, clock multiplication. In: DEPEND 2013, The Sixth Int. Conf. on Dependability. pp. 6–15 (2013)
8. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. pp. 317–331. SP '10, IEEE Computer Society, Washington, DC, USA (2010)