Techniques for Fuzzing Embedded and Distributed Systems

18 November 2020

Michael D. Brown

Software Assurance Branch Head (interim)

Brian Schultz

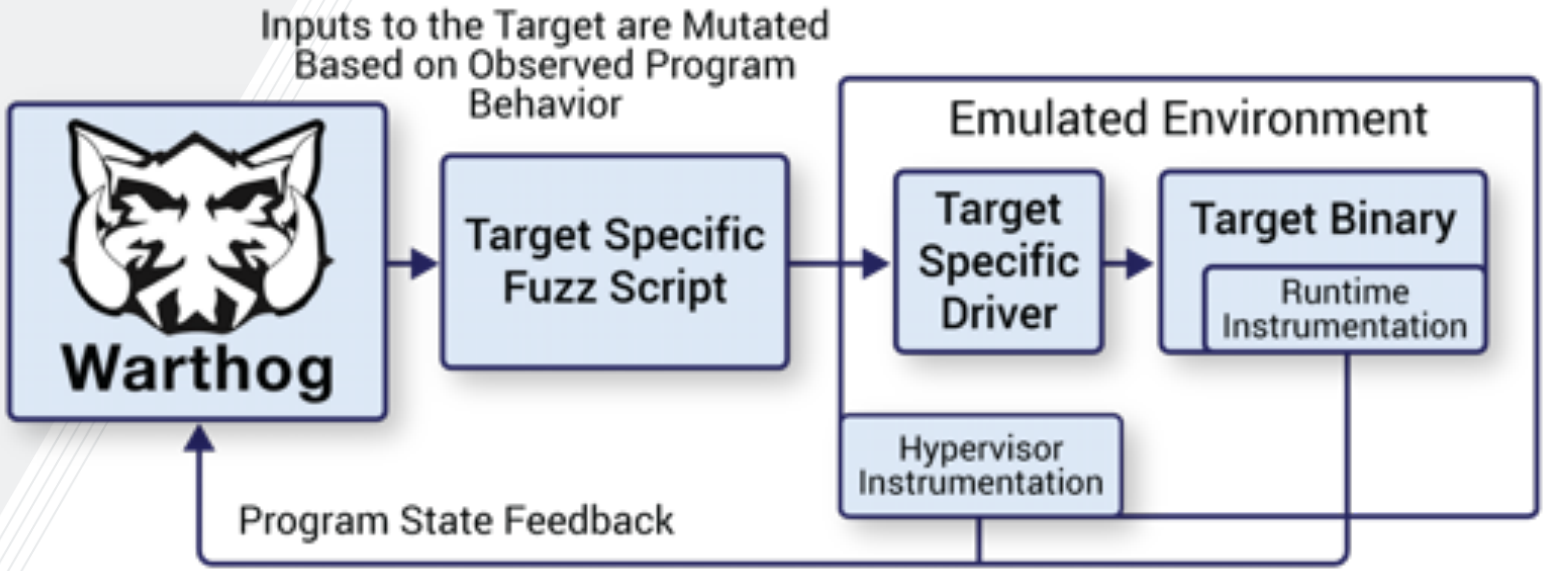
Software Assurance Branch

What is Fuzzing?

- Fuzzing (or Fuzz testing) is an automated technique for vulnerability discovery in programs or systems.
- In its most basic form, fuzzing is a brute force method; repeatedly run a program with randomly generated inputs to see if any of them cause a crash.
- A program crash likely indicates some kind of program bug, which may be exploitable.
- Fuzzing is useful for both DCO (fixing bugs in blue systems) and OCO (finding holes in red systems)
- Basic fuzzing is very inefficient; most of the inputs generated will be invalid and probably won't find interesting bugs.

What is Fuzzing?

- Modern fuzzers incorporate different techniques that increase the likelihood of discovering crashes.
- For example, GTRI's in-house (GOTS) fuzzer, Warthog, uses virtualization to parallelize fuzzing and program feedback to guide the input generator.

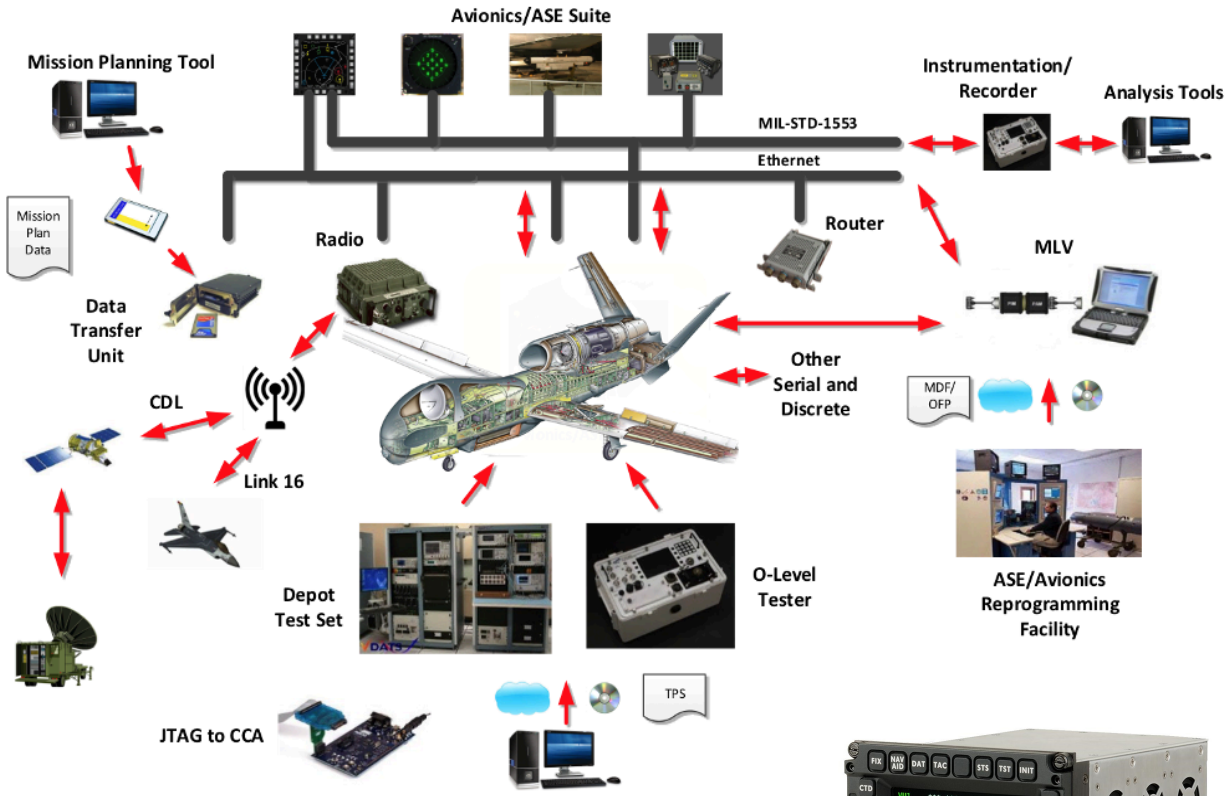


Fuzzing Tools and Their Limitations

- Modern fuzzers that employ these advanced techniques have been shown to be very effective at discovering exploitable bugs.
- However, these tools are still far from universal; they largely target enterprise software running on general purpose computing systems.
- This is very different from embedded systems which exhibit tight coupling between software, the OS, and the hardware (which is often non-standard).
- Modern fuzzers are also difficult to use on distributed systems due to their reliance on a 1:1 fuzzing paradigm. This model is a poor fit for distributed systems that are made up of several interconnected components working in concert.

Fuzzing Weapons System Software

- Most DoD weapons systems we have analyzed are implemented as either embedded systems, distributed systems, or both (cyber-physical systems)
- This means modern fuzzing tools cannot be brought to bear on these systems without significant effort.
- Unfortunately, this affects both small scale systems like aircraft avionics and large scale platforms like helicopters and ships.



Techniques for Fuzzing Weapons System Software

- To overcome these limitations, GTRI has researched and developed several candidate techniques for applying modern fuzzing capabilities to DoD weapons systems.
- In the first part of this presentation, we discuss prototype extensions to Warthog that allow us to virtualize embedded systems and fuzz them.
- In the second part of this presentation, we discuss a prototype fuzzing tool we have developed called NetFuzz. NetFuzz is purpose built for fuzzing distributed systems and can also identify non-crashing bugs such as resource exhaustion vulnerabilities.

Embedded Systems

Warthog Embedded Systems Extensions

Fuzzing Embedded Software

Problem:

- Fuzzing embedded systems software running on real-time operating systems (RTOS) is not possible with existing fuzzers.
- Embedded systems software is tightly coupled to RTOS and hardware, both of which are abstracted in general purpose computing systems.

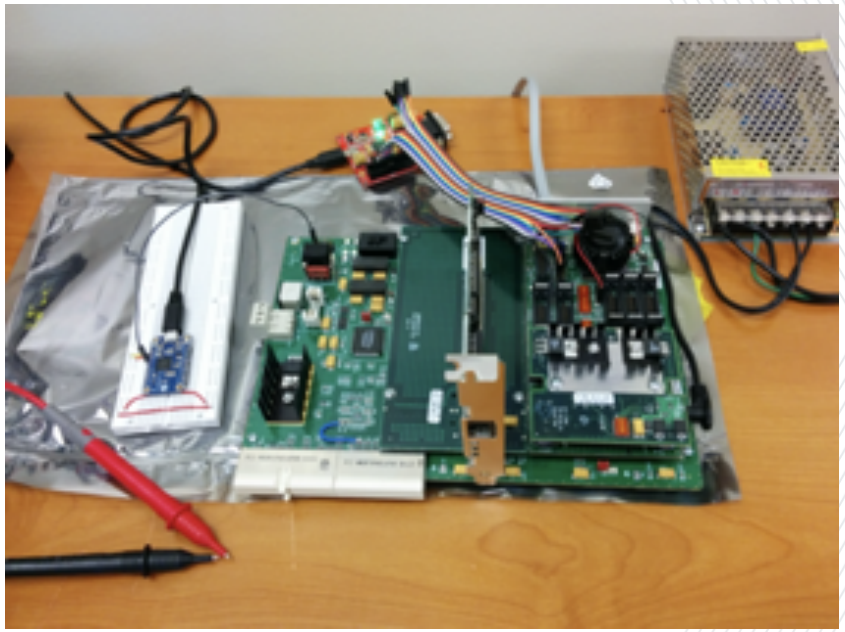
Approach:

- Emulate an embedded system's hardware
- Create a virtual environment that can run instrumented embedded software on emulated hardware
- Fuzz this environment with Warthog to discover vulnerabilities
- Verify these vulnerabilities actually exist on physical hardware

Fuzzing Embedded Software

Implementation:

- Built support for PPMC7400 test bed device in to QEMU.
- Required building custom emulation support for the board's memory controller and network hardware.



Fuzzing Embedded Software

Implementation:

- For our proof of concept, we targeted the Unix pipe library in VxWorks 5.4.
- We created a simple wrapper program for the OS library and placed it (along with the VxWorks 5.4 RTOS) on this virtual hardware.
- Provide seed inputs and necessary scripting to enable warthog to fuzz this environment.
  - Fuzz script (pictured right) is written in Python and is less than 100 lines of code.

```python
class TestMonitor(vxworks.Monitor):
    def fuzz(self, input_):
        input1 = input_[0:(len(input_)) // 2] + b'\x00'
        input2 = input_[(len(input_) // 2):] + b'\x00'
        spec = {
            'params': {
                '1': [0x01, 0x64, 0x02, 0x0],
                '2': [],
                '4': [len(input2) - 1 if len(input2) > 0 else 1],
                'ptr': [input1, input2, b"x" * (len(input_))],
            },
            'funcs': [
                {
                    'name': 'pipeDrv',
                    'ret_index': 100,
                    'args': [],
                },
                {
                    'name': 'pipeDevCreate',
                    'ret_index': 101,
                    'args': [
                        (vxworks.ARG_PTR, 0),
                        (vxworks.ARG_1_BYTE, 0),
                        (vxworks.ARG_1_BYTE, 1),
                    ],
                },
                {
                    'name': 'open',
                    'ret_index': 0,
                    'args': [
                        (vxworks.ARG_PTR, 0),
                        (vxworks.ARG_1_BYTE, 2),
                    ],
                },
```

Fuzzing Embedded Software

Results:

- Warthog discovered a vulnerability in the `ioFullFileNameGet` library function, which we verified on the actual hardware.
- Using the information collected by the fuzzer, we were able to craft an exploit that compromises a file open operation on the embedded system.
- When delivered via the board's network interface, the attacker can open a system shell on the device with full system privileges (a complete compromise).

Fuzzing Embedded Software

Takeaways:

- It is possible to fuzz embedded systems with modern tools, but it requires significant effort to put devices and RTOS under emulation.
- Biggest payoffs from employing this technique will be on systems / software packages that are common to multiple platforms.

Future Efforts:

- Add support for different types of hardware and RTOS.
- GTRI is currently exploring advanced software analysis and transformation techniques to fuzz embedded software without requiring emulation.

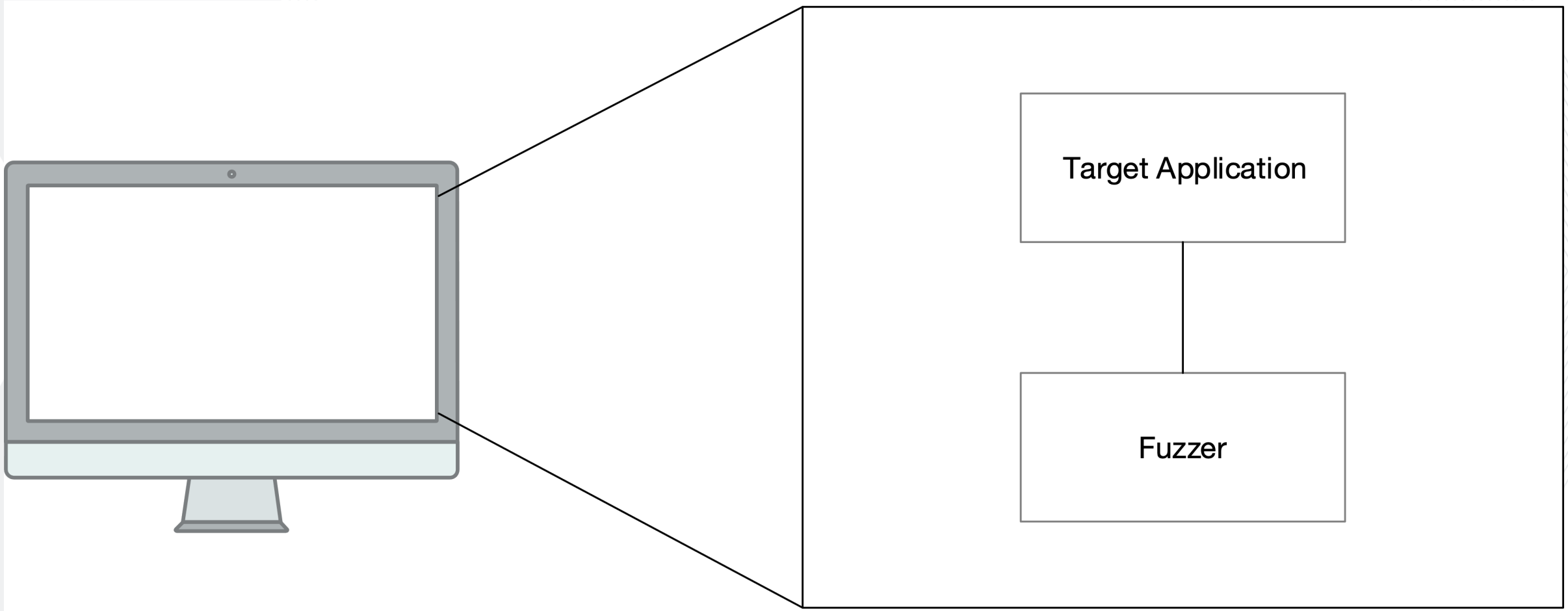
Distributed Systems

NetFuzz Distributed Systems Fuzzer

Georgia Tech Research Institute

Traditional vs Distributed System Fuzzing

Traditional Fuzzing Setup



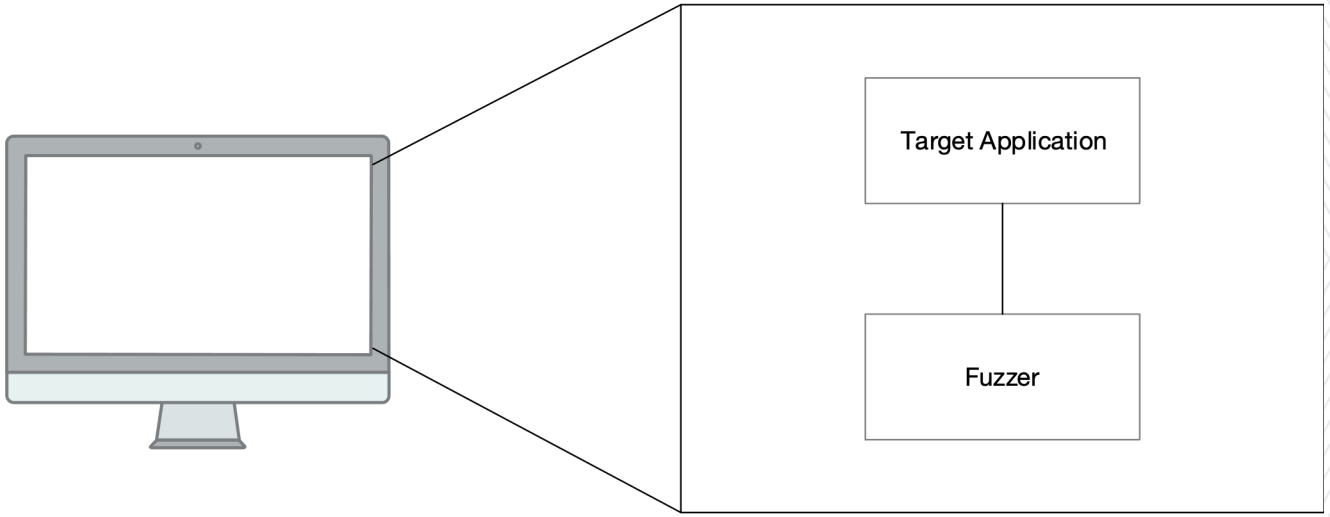
Traditional Fuzzing

Benefits

- Usually deterministic output
- Many variants and programs
- Well studied
- “Small” Input Space

Limitations

- Simple Applications
- Limited Interactions
- Instrumentation and stub services
- Misses soft errors



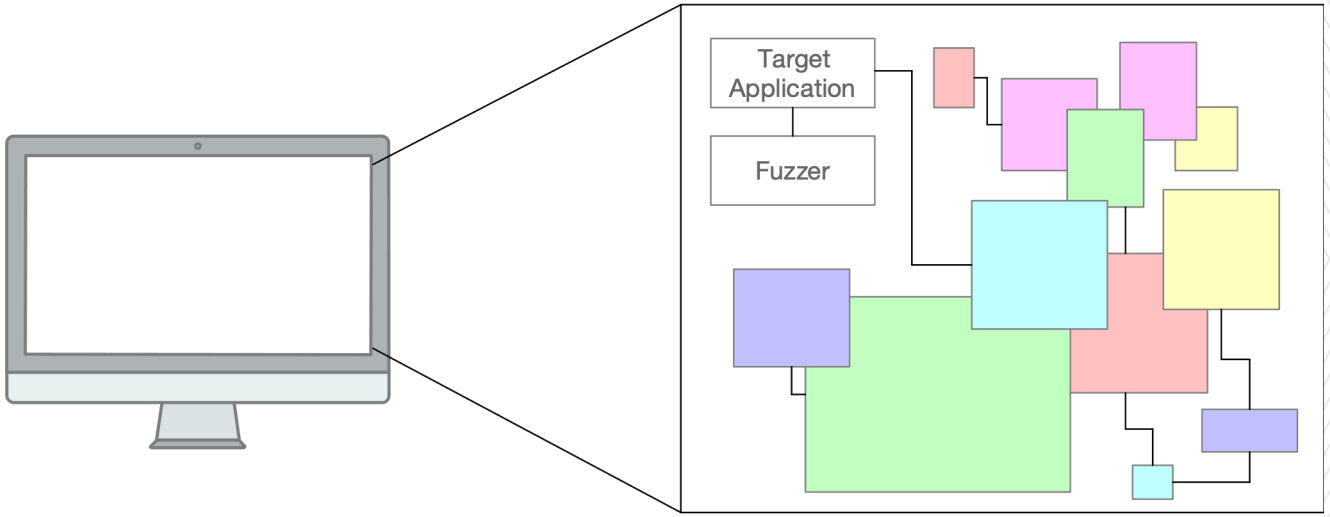
Traditional Fuzzing

Benefits

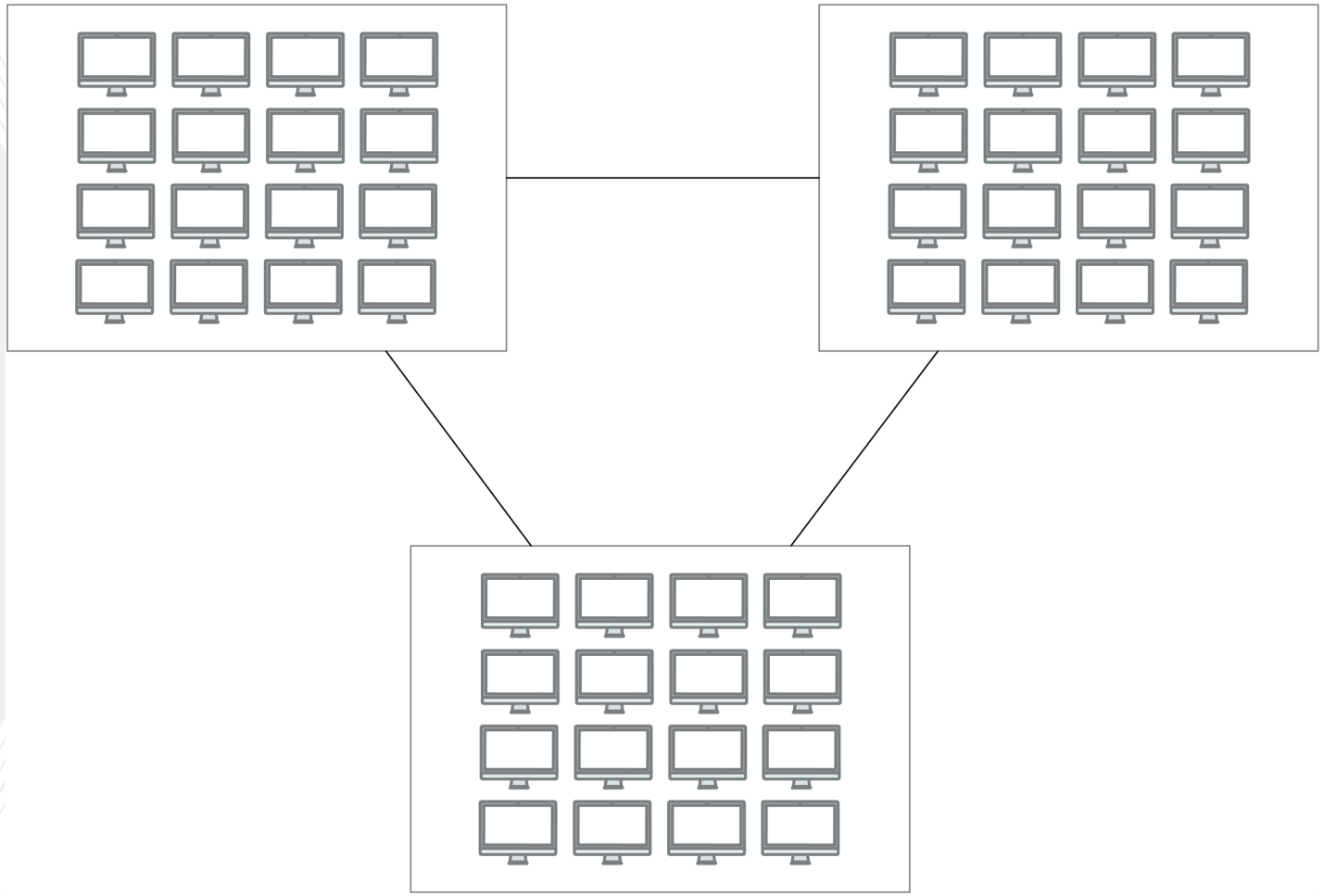
- Usually deterministic output
- Many variants and programs
- Well studied
- “Small” Input Space

Limitations

- Simple Applications
- Limited Interactions
- Instrumentation and stub services
- Misses soft errors



Distributed System Fuzzing



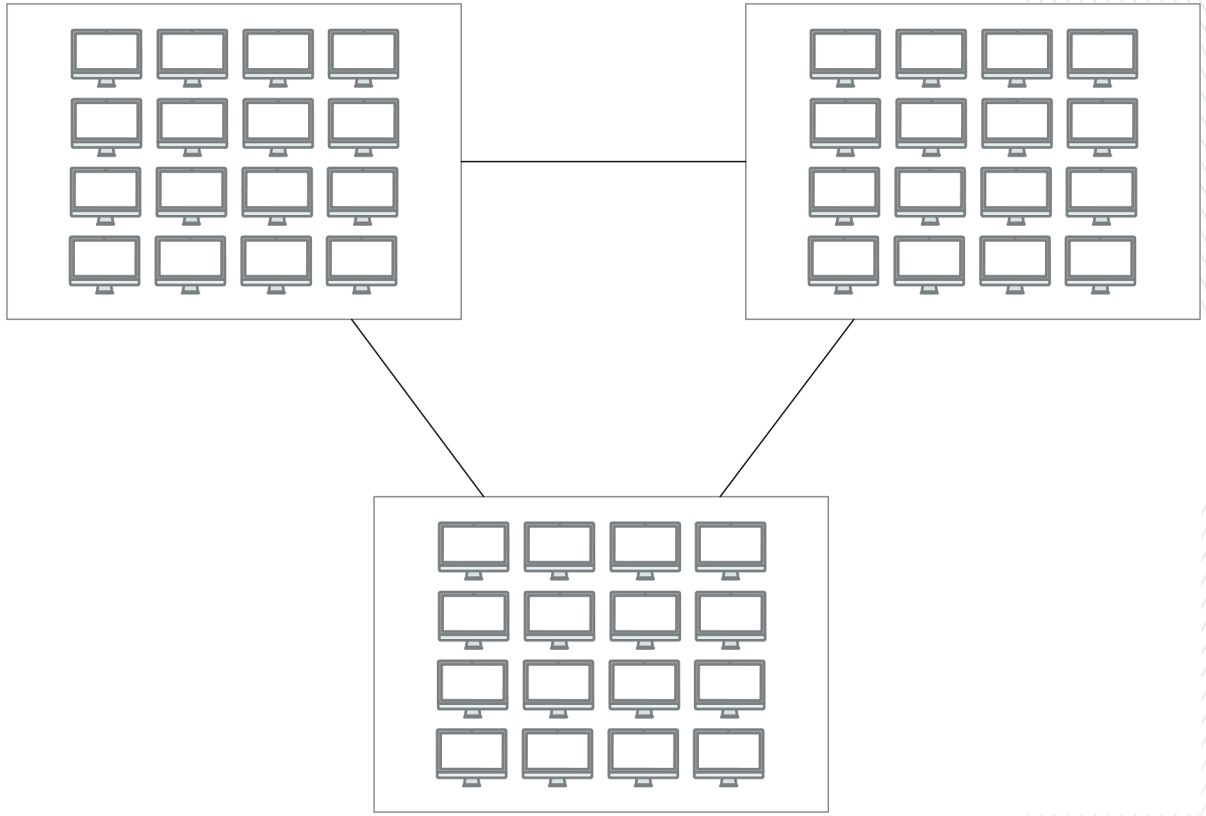
Distributed System Fuzzing

Benefits

- Side Effects
- Aggregate Effects
- Resource Exhaustion
- Correlational Inputs

Difficulties

- Large Input Space
- State Management
- Initialization
- Input Tracing



Georgia Tech Research Institute

NetFuzz

NetFuzz

What it does

- Fuzzes one or more processes in a distributed system
- Monitors one or more processes for crashes
- Ends fuzzing when a crash is detected
- Saves window of inputs leading up to crash

Design

- Distributed system for fuzzing distributed systems
- Python 3.7+
- Async support via *asyncio*
- Inter-process communication via *zmq*

NetFuzz

What it does

- Fuzzes one or more processes in a distributed system
- Monitors one or more processes for crashes
- Ends fuzzing when a crash is detected
- Saves window of inputs leading up to crash

Design

- Distributed system for fuzzing distributed systems
- Python 3.7+
- Async support via *asyncio*
- Inter-process communication via *zmq*

NetFuzz Components

Fuzz Controller

- Single central unit
- Status messages
- Start/stop fuzzing

Fuzz Node

- One or more
- Performs fuzzing
- Captures sequence of fuzzing inputs
- Fuzzing replay

System Monitor

- Zero or more
- Provides feedback

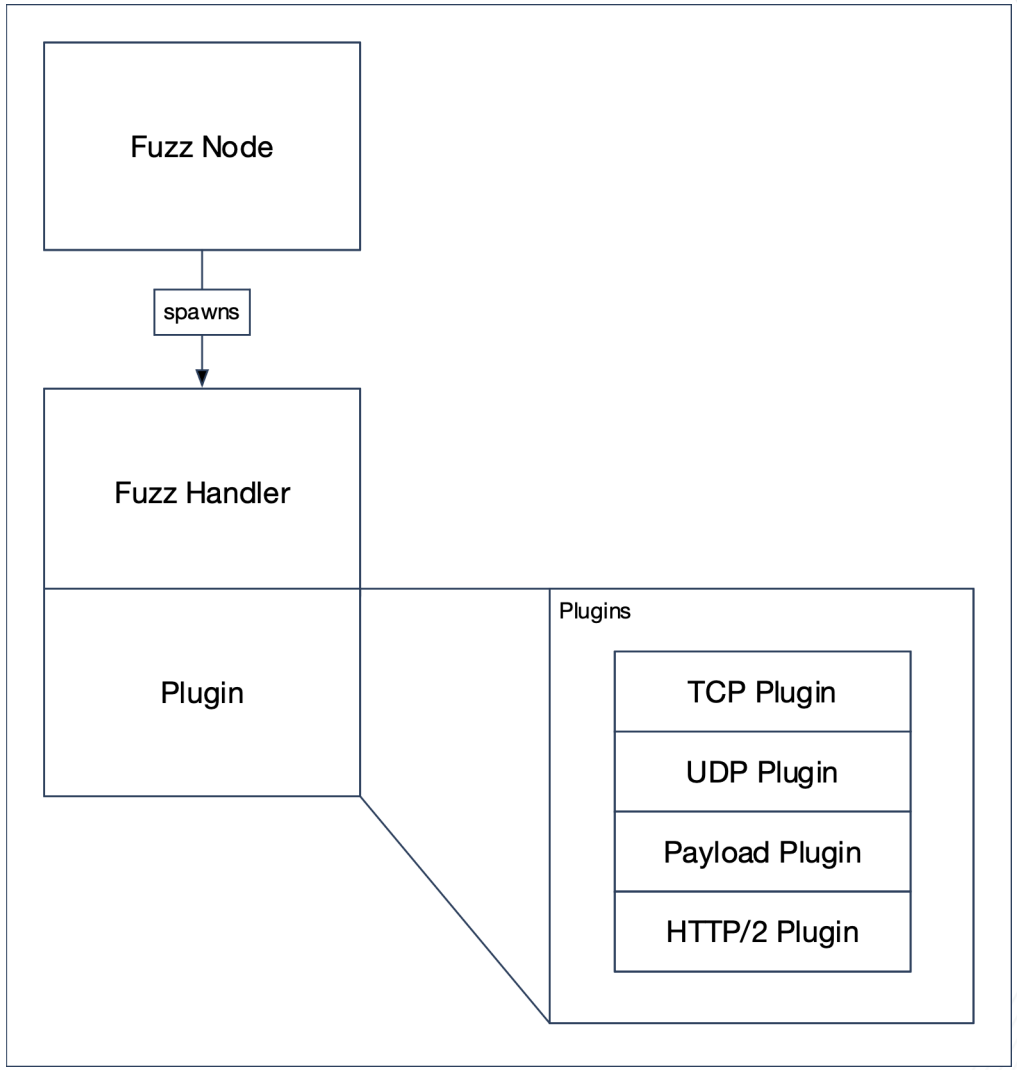
NetFuzz Features

Fuzzing paradigms

- One-to-one
- Many-to-one
- Many one-to-one
- Many-to-many

Plugin Architecture

- Fast protocol integration
- Config file or Python class



Georgia Tech Research Institute

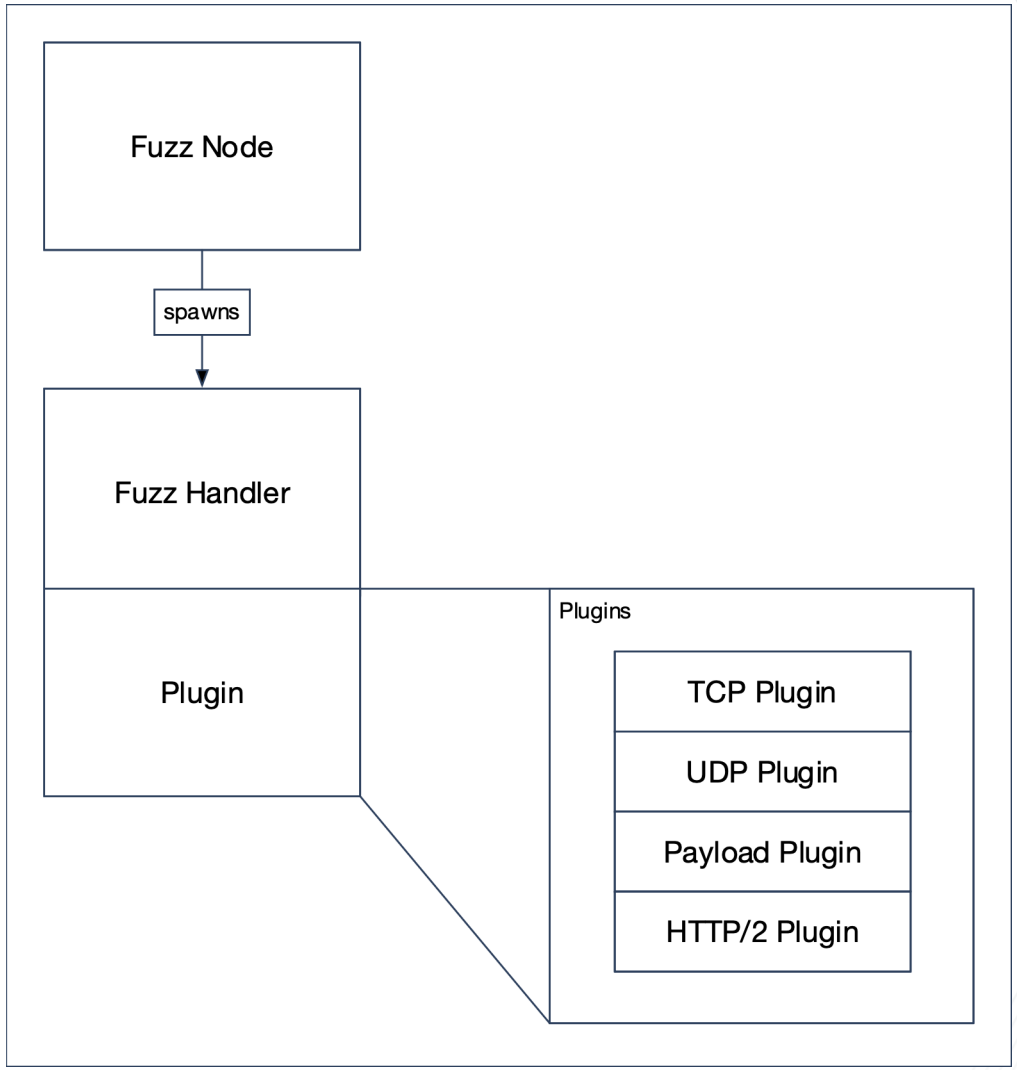
NetFuzz Features

Fuzzing paradigms

- One-to-one
- Many-to-one
- Many one-to-one
- Many-to-many

Plugin Architecture

- Fast protocol integration
- Config file or Python class



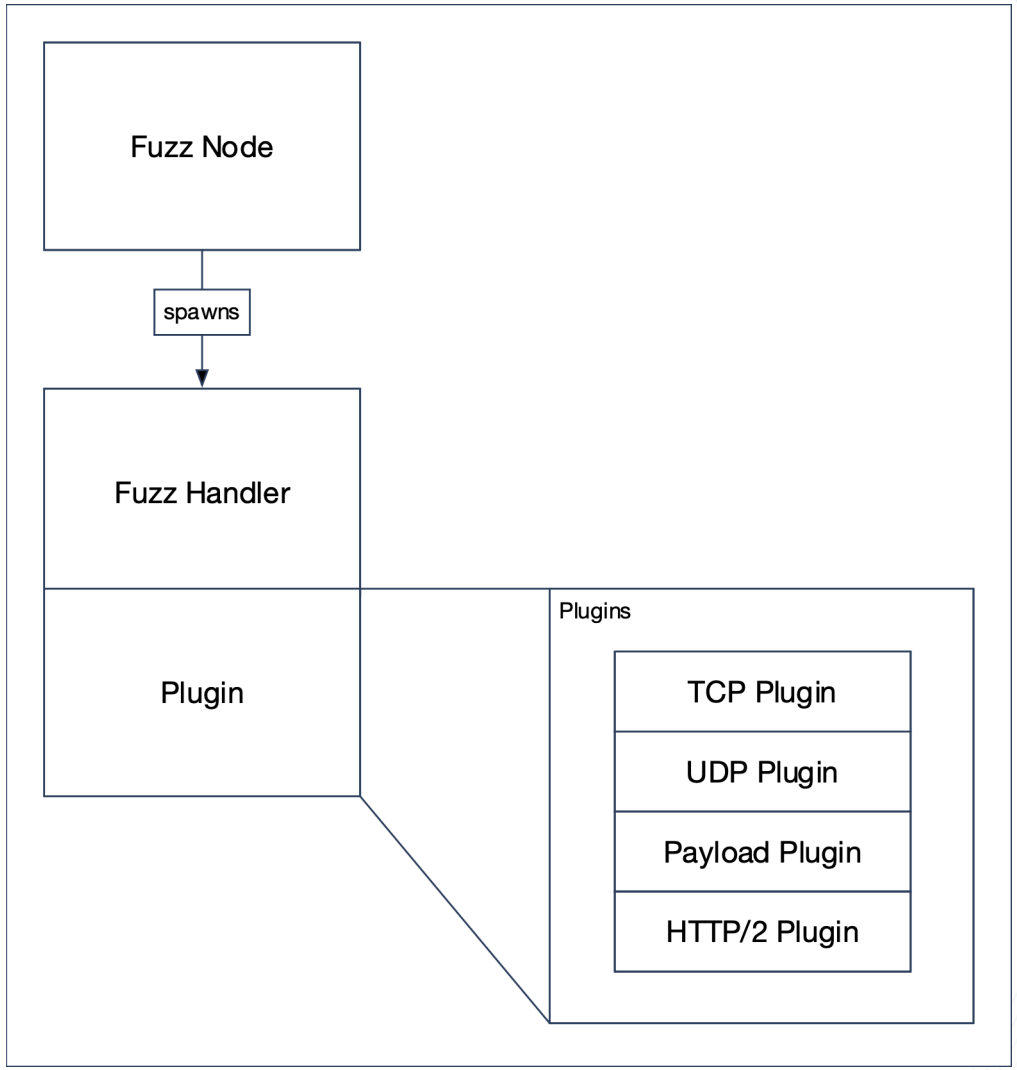
NetFuzz Features

Mutational fuzzing

- Read PCAP files
- Filter and mutate packets

Template fuzzing

- User-defined templates
- Stochastic fields



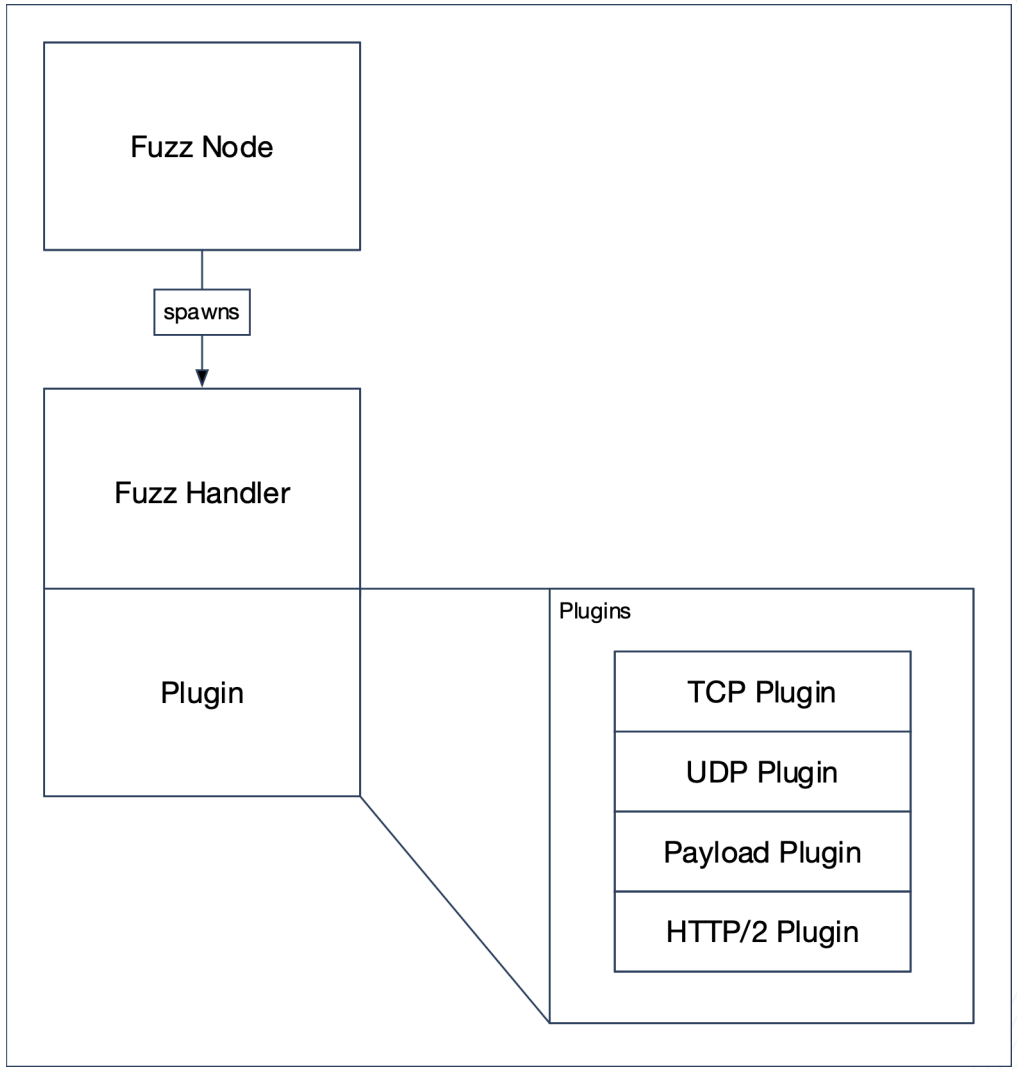
NetFuzz Features

Mutational fuzzing

- Read PCAP files
- Filter and mutate packets

Template fuzzing

- User-defined templates
- Stochastic fields



NetFuzz in Action

Vulnerability Assessment of a Large Scale CPS

- Received multiple VMs with customer software
- Installed NetFuzz on additional VM
- Created plugins to target different process

Results

- NetFuzz produced recurring crashes
- System Monitor showed increasing memory consumption
- Identified a resource exhaustion vulnerability caused by specific inputs

NetFuzz in Action

Vulnerability Assessment of a Large Scale CPS

- Received multiple VMs with customer software
- Installed NetFuzz on additional VM
- Created plugins to target different process

Results

- NetFuzz produced recurring crashes
- System Monitor showed increasing memory consumption
- Identified a resource exhaustion vulnerability caused by specific inputs

Takeaways and Future Work

Takeaways

- Generalization is difficult
  - Fast protocol iteration
  - Exploration patterns
- Massive input space
  - Fuzzing takes a long time to produce results
  - Human guided inputs help reduce search time

Future Work

- Process state estimation
- Granular feedback mechanism

Questions?

Demonstrations excluded due to virtual workshop format.

michael.brown@gtri.gatech.edu