Format

- Welcome
- Discuss Confidentiality
- Issue Clearing (if required)
- Meaningful Update & Follow-Up
- Paper Pipeline
- Quick Talk (if desired)
- Presentation
- Feedback

REVelry Nyx

Fuzzing Hypervisors w/ Snapshotting & Affine Types

About Me | Jay Warne

Currently

- DARPA research
  - Side-channels
  - Processors
  - Hypervisors
  - Rowhammer Style Attacks
  - Program Analysis Methods
- Project Work
  - PO for Videographic Data Analysis
  - Routine Software Engineering
- Advisory
  - Product Development
  - Product Direction
  - Expert Reviewer

Previously

- Ran a Security Ops Team
- Occasional RE & Red Team
- Field Forensic Analysis & Tool Deployment

Things I Like

- Alpine Ski Racing
- Ski/ Alpine Mountaineering
- Ice/ Rock Climbing
- Surfing
- Backpacking

Nyx – tl;dr

- Code coverage
- Customized snapshotting tracking dirty pages & devise state
- Affine types to prevent inefficient/ ineffectual inputs

That is build into kAFL

If you just want to use it,
grab kAFL from IntelLabs

Some Terms

Terms

- L0 - Hypervisor
- L1 - Guest Hypervisor
- L2 - Guests of Guest Hypervisor
- Hypercall - Software trap from guest to hypervisor; “hypervisor syscall”
- DAG - Directed Acyclic Graph

What We Will Cover

- Hypervisor Fuzzing Roadblocks
- Some Additional Background
    - x86 Hypervisors
    - Trap-VM-Exit
    - Affine Types
- Nyx's Solution
    - What it solves
- Results
- Implementation Details Q&A

Hypervisor Fuzzing Roadblocks

Hypervisor Fuzzing Roadblocks

Hypervisor Fuzzing Roadblocks

1) Handling Crashes
2) Nested Virtualization
3) Stateful Applications
4) Interactive Interfaces

Hypervisor Fuzzing Roadblocks

1) Handling Crashes
2) Nested Virtualization
3) Stateful Applications
4) Interactive Interfaces

Hypervisor Fuzzing Roadblocks

1) Handling Crashes
2) Nested Virtualization
3) Stateful Applications
4) Interactive Interfaces

Example: Write access to Nested Guest (L2)

- L2 Writes to Port I/O Address
- L0 (Host Hypervisor) handles trap
- L0 Passes Exit Reason to L1 (guest Hypervisor)
- Trap VM re-entry at L1
- Emulate it and continue execution in L2

Hypervisor Fuzzing Roadblocks

1) Handling Crashes
2) Nested Virtualization
3) Stateful Applications
4) Interactive Interfaces

Examples –

- Write file to disk
- Time to derive hash table key

Hypervisors have many stateful components

Reproduction of test cases requires full state

Hypervisor Fuzzing Roadblocks

1) Handling Crashes
2) Nested Virtualization
3) Stateful Applications
4) Interactive Interfaces

Fuzzers using single unstructured byte arrays are bad at this

- Constantly generating failure cases that aren't relevant → invalid pointers

Grammar Based Fuzzers

- Describe input well but don't address the temporal issue either
- Produce Directed Acyclic Graphs (DAGs) not binary data

Additional Background

x86 Hypervisors

Share resources with Virtual Machines (guests)

Implemented with the help of CPU Features

- Instructions
- Access Schemas

Hypervisors try to emulate as much of hardware as they can

When they can’t, they “pass through”

Trapping and Paravirtualization

Privileged Operations in VMs are “Trapped” by the Hypervisor

Allows the hypervisor to emulate components and do security things

Emulated Drivers

- Memory-Mapped I/O (MMIO)
- Port I/O (PIO)

Trapping and Paravirtualization

Privileged Operations in VMs are “Trapped” by the Hypervisor

Allows the hypervisor to emulate components and do security things

Emulated Drivers

- Memory-Mapped I/O (MMIO)
- Port I/O (PIO)

Hypervisor Exit

Trapping and Paravirtualization

Privileged Operations in VMs are “Trapped” by the Hypervisor

Allows the hypervisor to emulate components and do security things

Emulated Drivers

- Memory-Mapped I/O (MMIO)
- Port I/O (PIO)

CPU Traps on in/ out

in/ out instructions

HV Captures these VM-Exists, looks at the reason, and returns the device emulator

Trapping and Paravirtualization

Privileged Operations in VMs are “Trapped” by the Hypervisor

Allows the hypervisor to emulate components and do security things

Paravirtualization

- Everything stays in guest mem
- Contains instructions for whole sequences
- **Context switch avoided**

Affine Types

Fancy academic words

Affine Types

Class of type system where values/ resources can only be used once

No reuse → No spurious use after closed

Nyx: Fast Snapshotting & Affine Types

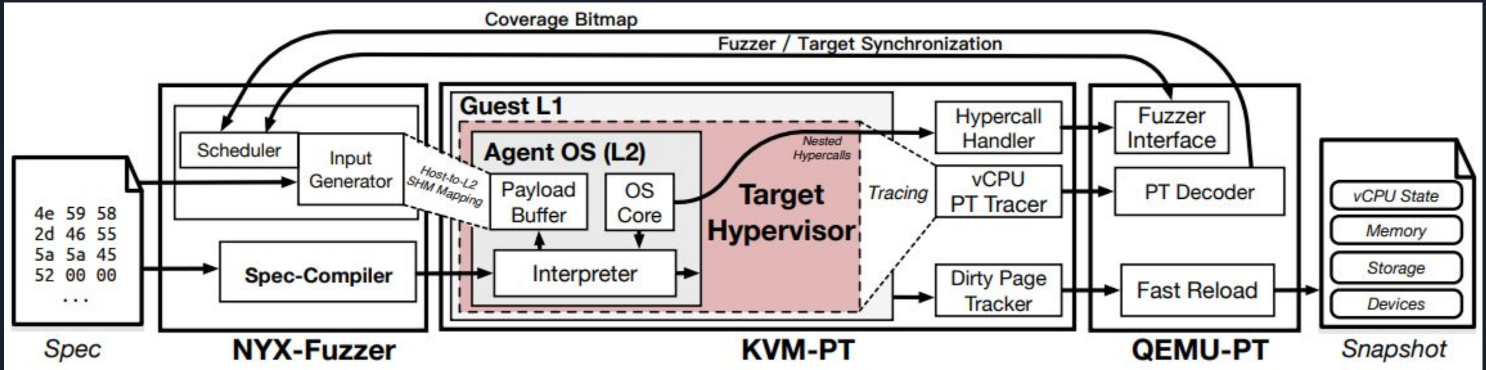
Stability & Determinism – Crashing & State

Stability & Determinism – Crashing & State

Fuzzer sits outside

Target is run inside of a KVM-PT

On Crash VM can restore to prior state



Stability & Determinism – Crashing & State

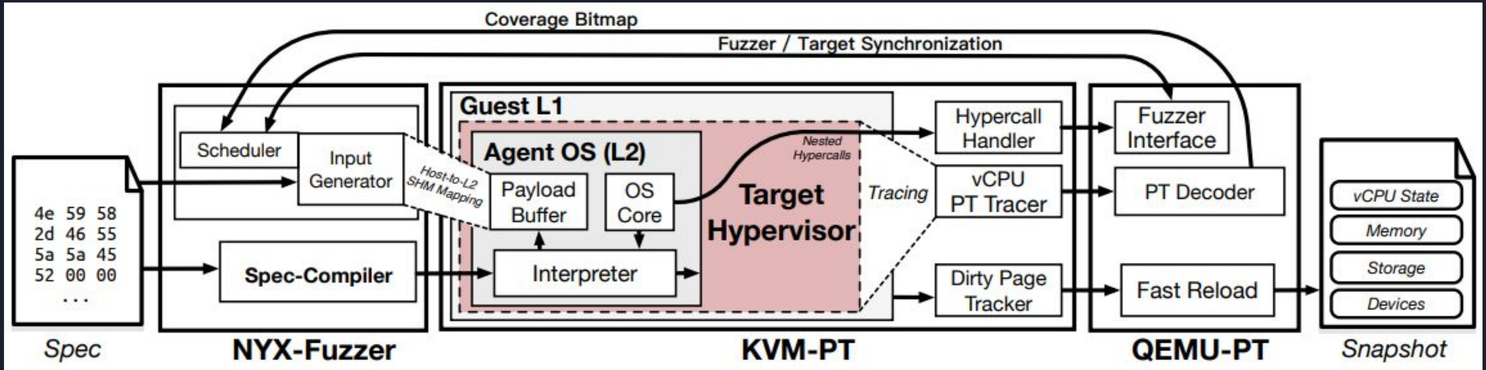
Fuzzer sits outside

Target is run inside of a KVM-PT

On Crash VM can restore to prior state

The Agent OS & Tgt Hypervisor have tons of state data → noisy coverage trace results

Extended KVM-PT & QEMU-PT to perform "Fast Reload" operations



Nested Virtualization

Nested Virtualization

Terms

- L0 - Hypervisor
- L1 - Guest Hypervisor
- L2 - Guests of Guest Hypervisor

Nested Virtualization

Terms

- L0 - Hypervisor
- L1 - Guest Hypervisor
- L2 - Guests of Guest Hypervisor

“Normal” Nested Virtualization Land

- L2 Hypercall → L0
- L0 Forwards Hypercall → L1

Nested Virtualization

Terms

- L0 - Hypervisor
- L1 - Guest Hypervisor
- L2 - Guests of Guest Hypervisor

"Normal" Nested Virtualization Land

- L2 Hypercall → L0
- L0 Forwards Hypercall → L1

Why wouldn't we want that?

Nested Virtualization

Terms

- L0 - Hypervisor
- L1 - Guest Hypervisor
- L2 - Guests of Guest Hypervisor

“Normal” Nested Virtualization Land

- L2 Hypercall → L0
- L0 Forwards Hypercall → L1

Why wouldn’t we want that?

- Speed
- Keeps our target hypervisor clean

Nested Virtualization

Terms

- L0 - Hypervisor
- L1 - Guest Hypervisor
- L2 - Guests of Guest Hypervisor

"Normal" Nested Virtualization Land

- L2 Hypercall → L0
- L0 Forwards Hypercall → L1

Nyx Version

- Extends Hyper-Cube OS [2020]
- Is "L2" inside tgt hypervisor "L1" communicates w/ "L0" with hypercalls
- Additional hypercalls and handlers
- Shared Memory between Fuzzer & L2

Interactive Interfaces – Affine Types Engine

Interactive Interfaces – Affine Types Engine

Custom Specifications From User

- Specify actions on the target

Nyx uses specs to generate and mutate “bytecodes” for fuzzing

These specifications are similar to the “context-free grammars” used by Syzkaller

Interactive Interfaces – Affine Types Engine

Custom Specifications From User

- Specify actions on the target

Nyx uses specs to generate and mutate "bytecodes" for fuzzing

These specifications are similar to the "context-free grammars" used by Syzkaller

Interactive Interfaces – Affine Types Engine

Custom Specifications From User

- Specify actions on the target

Nyx uses specs to generate and mutate "bytecodes" for fuzzing

These specifications are similar to the "context-free grammars" used by Syzkaller

Specifications are used to generate C code

Inputs are represented by a DAG

Interactive Interfaces – Affine Types Engine

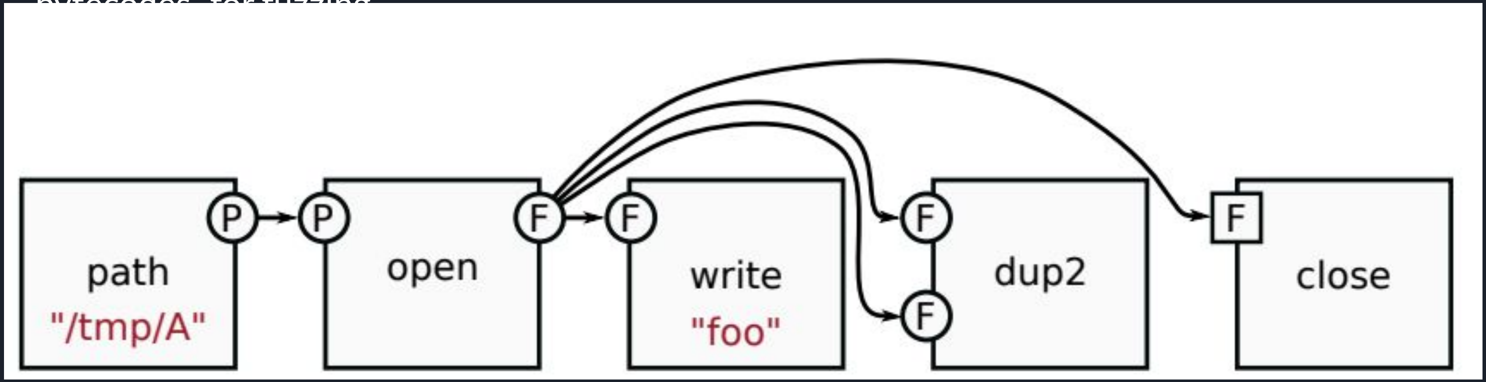
Custom Specifications From User

- Specify actions on the target

Nyx uses specs to generate and mutate “bytecodes” for fuzzing

Inputs are represented by a DAG

[n_new_path_id, p, n_open_id, p, f, n_write_id, f, n_dup2_id, f, f, n_close_id, f] – node, edge(s)



Interactive Interfaces – Affine Types Engine

Custom Specifications From User

- Specify actions on the target

Nyx uses specs to generate and mutate "bytecodes" for fuzzing

These specifications are similar to the "context-free grammars" used by Syzkaller

Specifications are used to generate C code

- Implements the interpreter
- User must provide behavior of nodes

Inputs are represented by a DAG

[n_new_path_id, p, n_open_id, p, f, n_write_id, f, n_dup2_id, f, f, n_close_id, f] – node, edge(s)

Results

Results

Nyx was compared with HyperCube, the tool it extends and is seeking to outperform

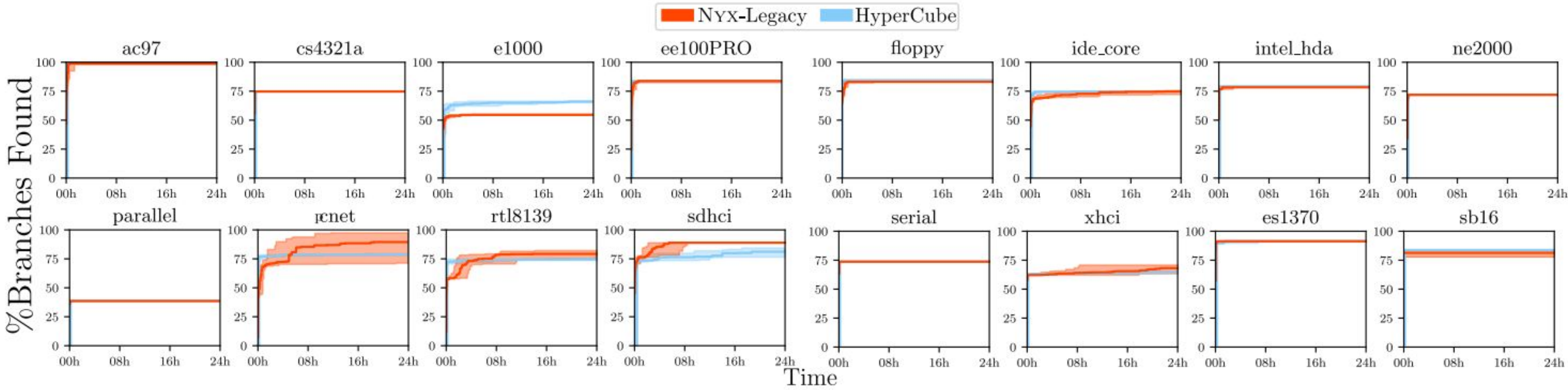
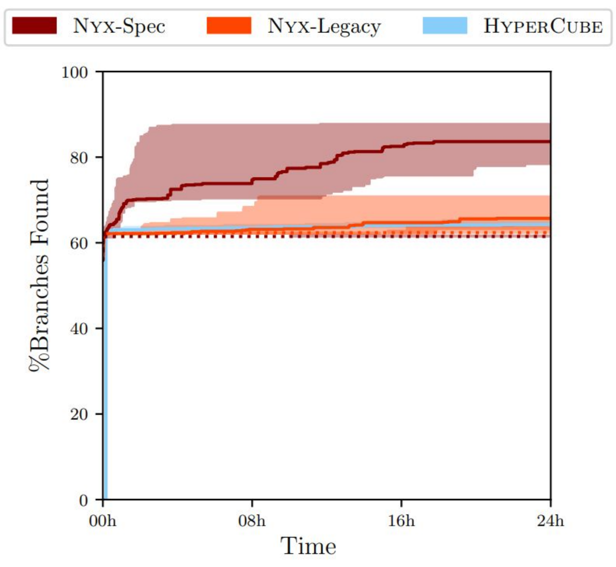
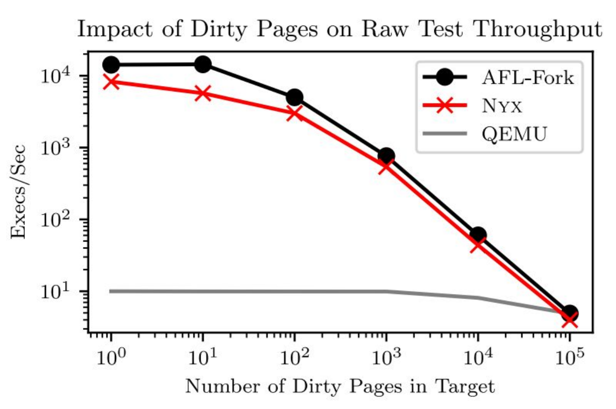


Figure 7: The median, best, and worst branch coverage of 10 runs (24h each).

Results

Device	VDF Cov	HYPER-CUBE Cov	NYX Cov	NYX Δ
AC97	53.0%	**100.00%**	98.92%	-1.62
CS4231a	56.0%	74.76%	74.76%	-
ES1370	72.7%	91.38%	91.38%	-
Intel-HDA	58.6%	**79.17%**	78.33%	-0.84
SoundBlaster	81.0%	**83.80%**	81.34%	-2.46
Floppy	70.5%	**84.51%**	83.10%	-1.41
Parallel	42.9%	38.61%	38.61%	-
Serial	44.6%	73.76%	73.76%	-
IDE Core	27.5%	74.87%	74.69%	-0.18
EEPro100	75.4%	83.82%	83.82%	-
E1000	81.6%	**66.08%**	54.55%	-11.53
NE2000 (PCI)	71.7%	71.89%	71.89%	-
PCNET (PCI)	36.1%	78.71%	**89.49%**	+10.78
RTL8139	63.0%	74.68%	**79.28%**	+4.60
SDHCI	90.5%	81.15%	**88.93%**	+7.78
XHCI	-	64.70%	**69.93%**	+5.23





Implementation Details

Hypercall Interaction