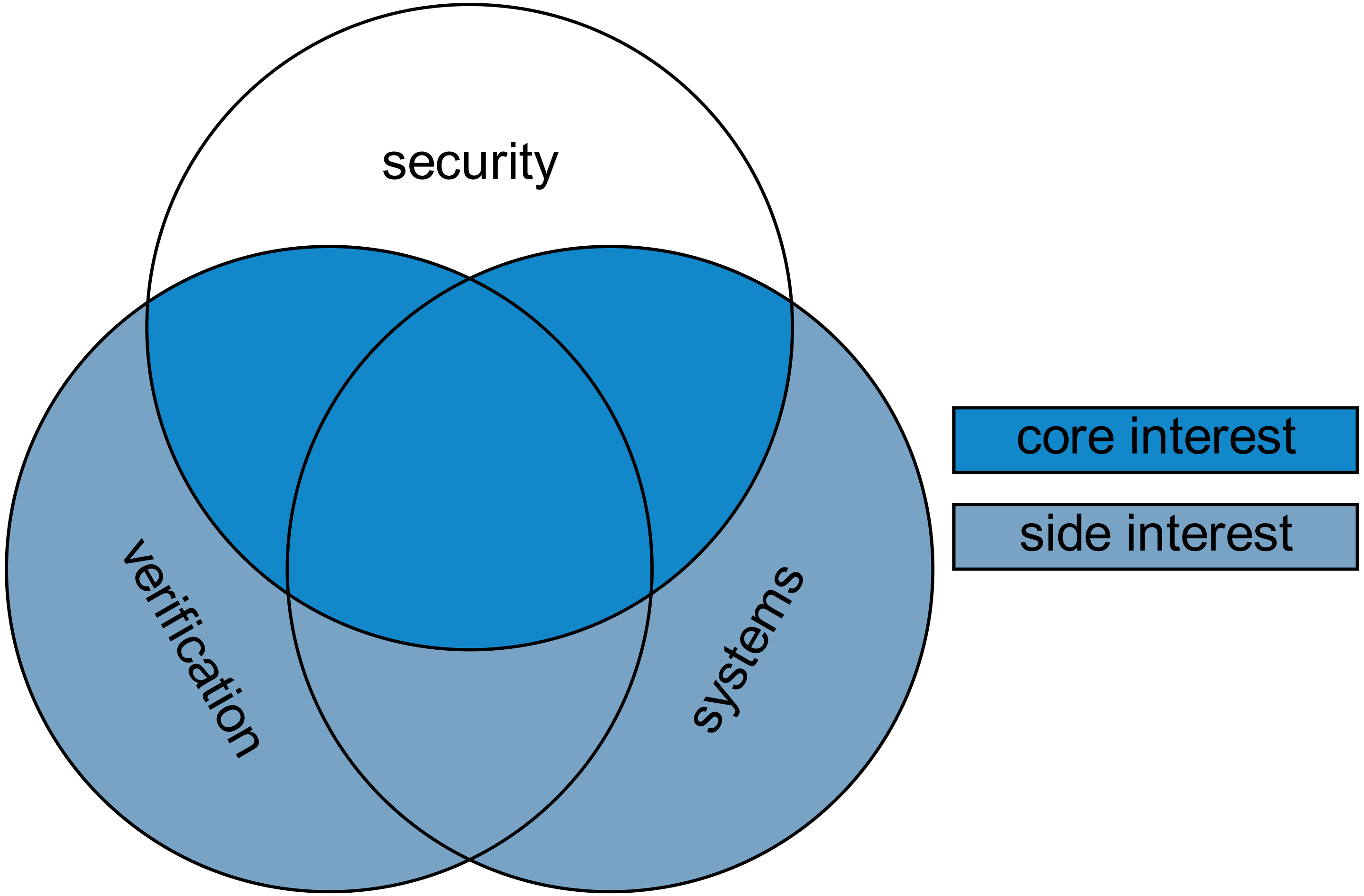
AVR@VUSec

Erik van der Kouwe & Herbert Bos

June 14th, 2023

VUSec Research Areas



What do we do?

- Novel attacks
- Efficient defenses
- Automated vulnerability finding
- Reverse engineering
- Fault tolerance
- Formal verification

What do we do?

- Novel attacks
- Efficient defenses
- **Automated vulnerability finding**
- Reverse engineering
- Fault tolerance
- Formal verification

Fuzzing

"Explore code at runtime to find issues"

Early work on directed fuzzing Dowser [USENIX Sec'13]

Fuzzing

“Explore code at runtime to find issues”

In hardware	Examples
Side channels	Absynthe [NDSS’20]
Rowhammer	TRRespass [S&P’20]
Speculative Execution	Kasper [NDSS’22], BHI [USENIX Sec’22]
Pre-silicon	BugsBunny [SILM’22] + ongoing

Fuzzing

"Explore code at runtime to find issues"

In firmware

Rehosting

Examples

FirmWire [NDSS'22],
FuzzWare [USENIX Sec'22]

→ Less active in this area these days

Fuzzing

"Explore code at runtime to find issues"

In OS kernels	Examples
Linux Type Confusion	Uncontained [USENIX Sec'23]
Speculative execution	Kasper [NDSS'22]

Fuzzing

"Explore code at runtime to find issues"

In applications	Examples
Grammar-based	IFuzzer [ESORICS'16]
Smarter inputs	VUzzer [NDSS'17]
Directed fuzzing	Parmesan [USENIX Sec'20]
Performance / snapshots	SNAPPY [ACSAC'22]
Performance / sanitizers	FloatZone [USENIX Sec'23]
Performance / problems	Don't Look UB [PLDI'23]
Performance / collab	Cupid [ACSAC'20]

Beyond Fuzzing

Ongoing work on other AVR topics

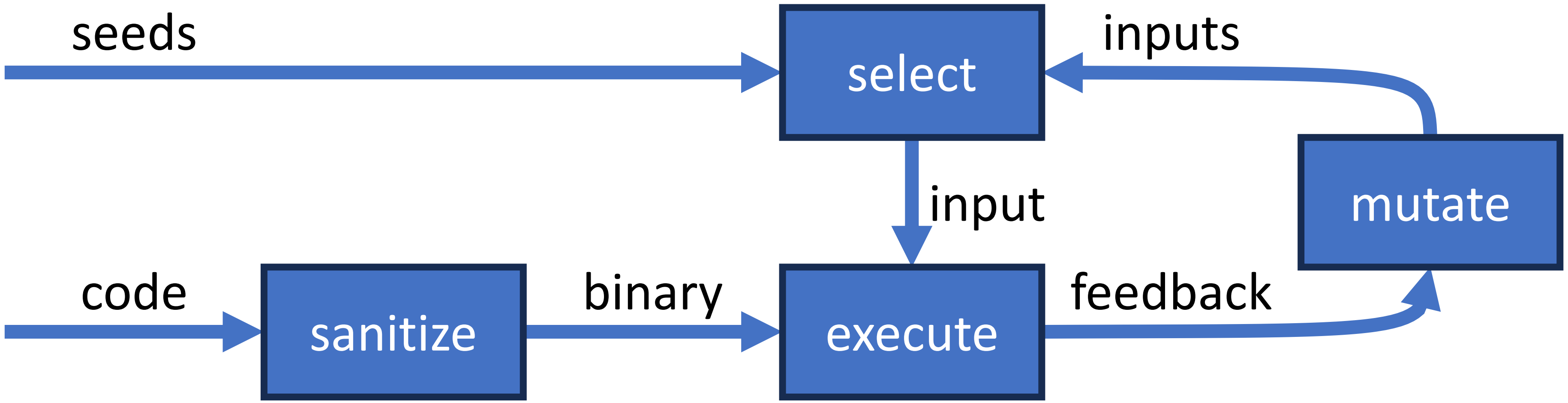
- Vulnerability analysis
- Automated patching

In this presentation…

I will focus on fuzzing applications for crashes.

Fuzzing

- Fuzzing is at the heart of AVR
- Surprisingly effective: finds more bugs than we can fix

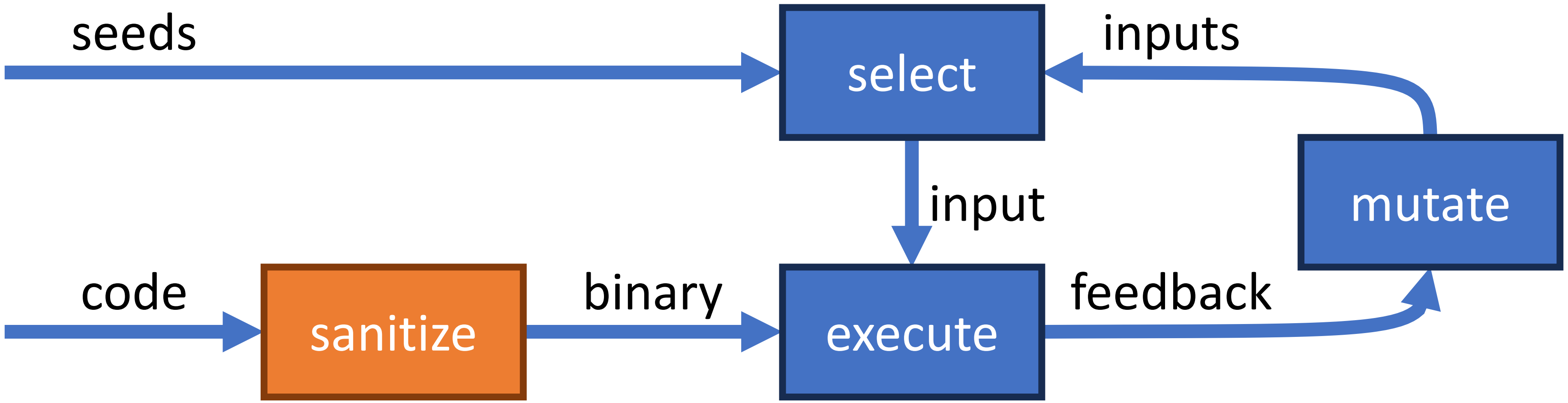


Fuzzing

No! You waste energy and time!

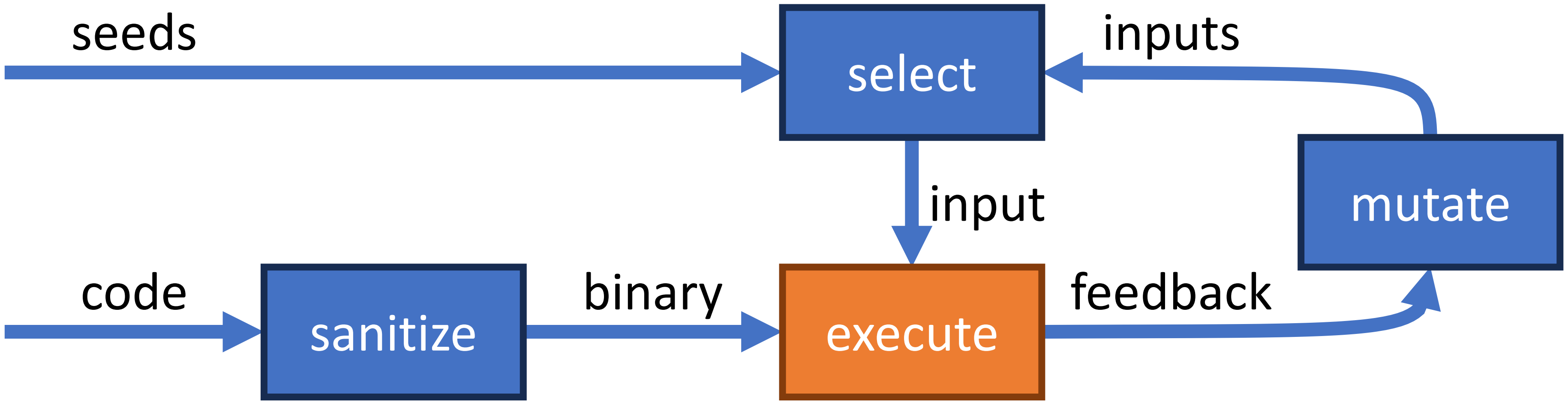
Speeding Up Fuzzing

- Sanitization detects memory errors early, but greatly slows down execution
- **FloatZone: repurpose COTS hardware to make this efficient**



Speeding Up Fuzzing

- We execute the same code over and over again, even before we process changes in input
- **Snappy: take snapshots to reduce redundant execution**



FloatZone: Accelerating Memory Error Detection using the Floating Point Unit

Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. USENIX Security 2023.

FloatZone in a Nutshell

Reasons to accept the paper

- Paper is really just one trick, ... but what a cool trick!

Reasons to not accept the paper

- Paper is really just one trick

Recommended decision

1. Accept

Why FloatZone?

- Detects spatial and temporal memory errors
- Just 37% runtime overhead on SPEC CPU2006 and CPU2017
- 2.88x increase in fuzzing throughput compared to state of the art

Key Insight

- Memory Sanitizers heavily rely on expensive compare and branch instructions to check the validity of memory accesses
- The checks result in high overhead: ASan ~2x slowdown
  - e.g., due to polluting the Branch Predictor and frequent Cache misses
- Checks "always" fine!
- What if we perform sanitizer checks using **floating point additions**?

🤔

- And show you that these branchless checks are **twice as fast**

Solution: Exception-Based Checks

Express comparisons using **floating point underflow exceptions**!

… but when do they happen?

$$1.5 \cdot 2^{-126} - 1.0 \cdot 2^{-126} =$$

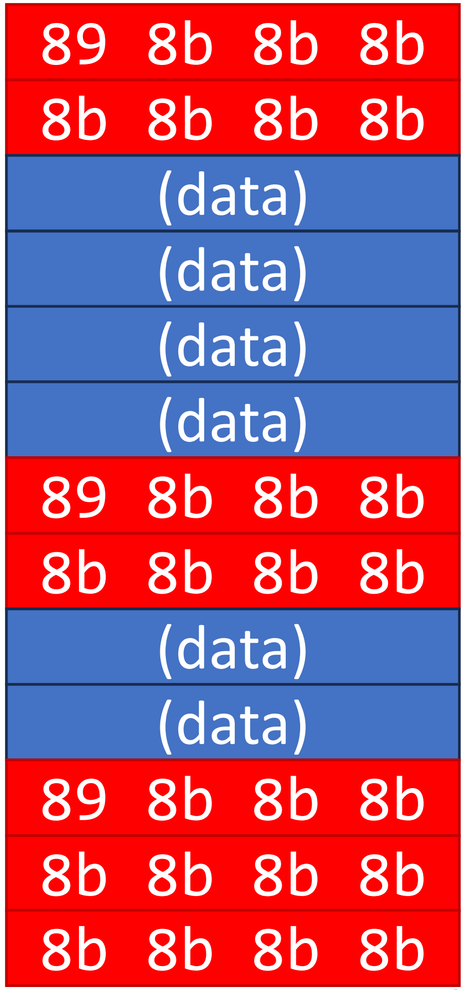
$0.5 \cdot 2^{-126}$ Not in normal form !

$1.0 \cdot 2^{-127}$ Min exponent is -126 !

Underflow SIGFPE

Approach

- Find magic numbers
  - `0x0b8b8b8a` (cast to `float`) causes underflow only when added to `0x8b8b8b8b` or `0x8b8b8b89`
- Maintain redzones in memory
  - In inaccessible regions, write 0x89 byte followed by repeating 0x8b bytes
- Add check before memory access
  - Add `0x0b8b8b8a` to value stored in memory
  - Faults in redzone



Fuzzing Evaluation

- Fuzzing using AFL++ and FloatZone as sanitizer, compared to state of the art
- Geomean increase in total executions across 7 binaries (24h):

Sanitizer	Throughput increase
ASan--	188%
ReZZan	71.4%

Snappy: Efficient Fuzzing with Adaptive and Mutable Snapshots

Elia Geretto, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. ACSAC 2022

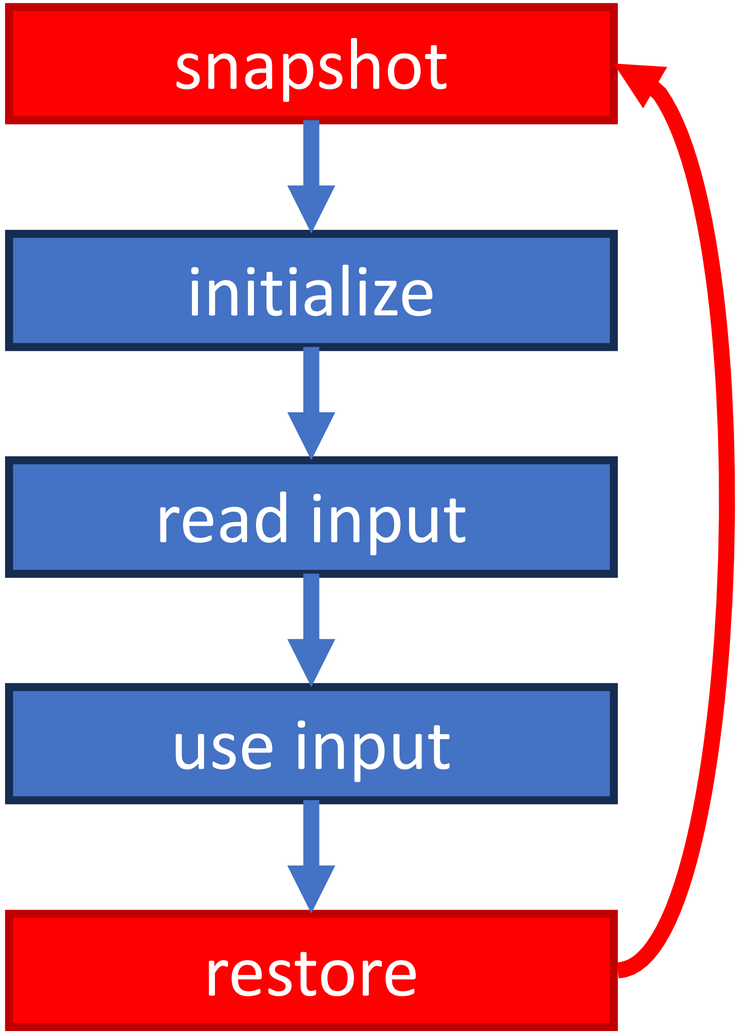
Why Snappy?

- Snappy reduces redundant execution to make fuzzers faster
- It achieves:
  - up to 1.76× speed increase in FuzzBench, with no significant regressions
  - up to 31% coverage increase after 24 hours on real-world programs

Key Insight

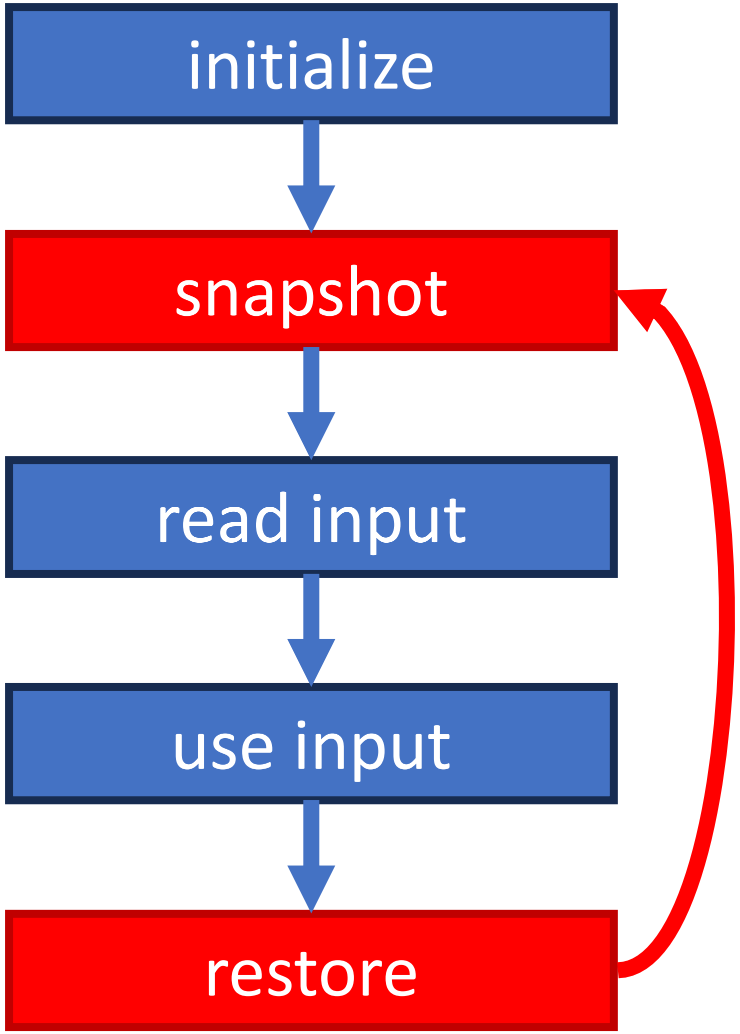
- Fuzzing is trial and error
  - More attempts make success (crashes) more likely
  - Speed (exec/sec) is extremely important
- Operations that do not depend on mutated input are redundant
  - Skip part of program execution that is always the same

Optimization Opportunities



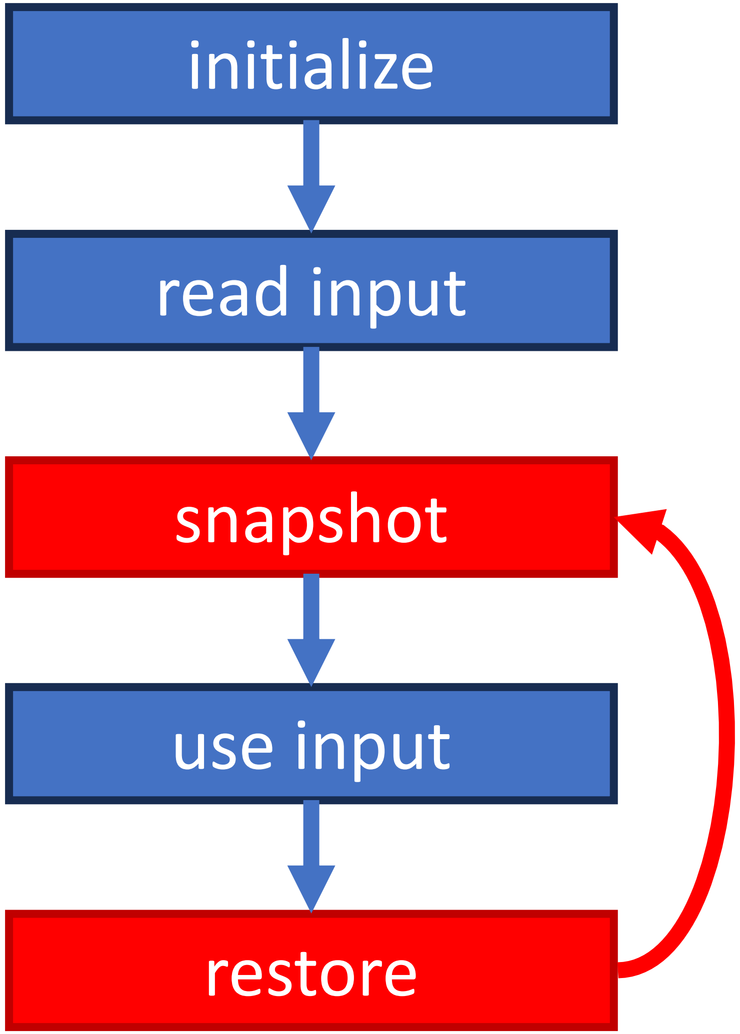
- Program initialization is redundant

Optimization Opportunities



- Program initialization is redundant
- Input data is copied before use, but does not influence the execution
- Several mutation operators leave most of the input unchanged

Optimization Opportunities

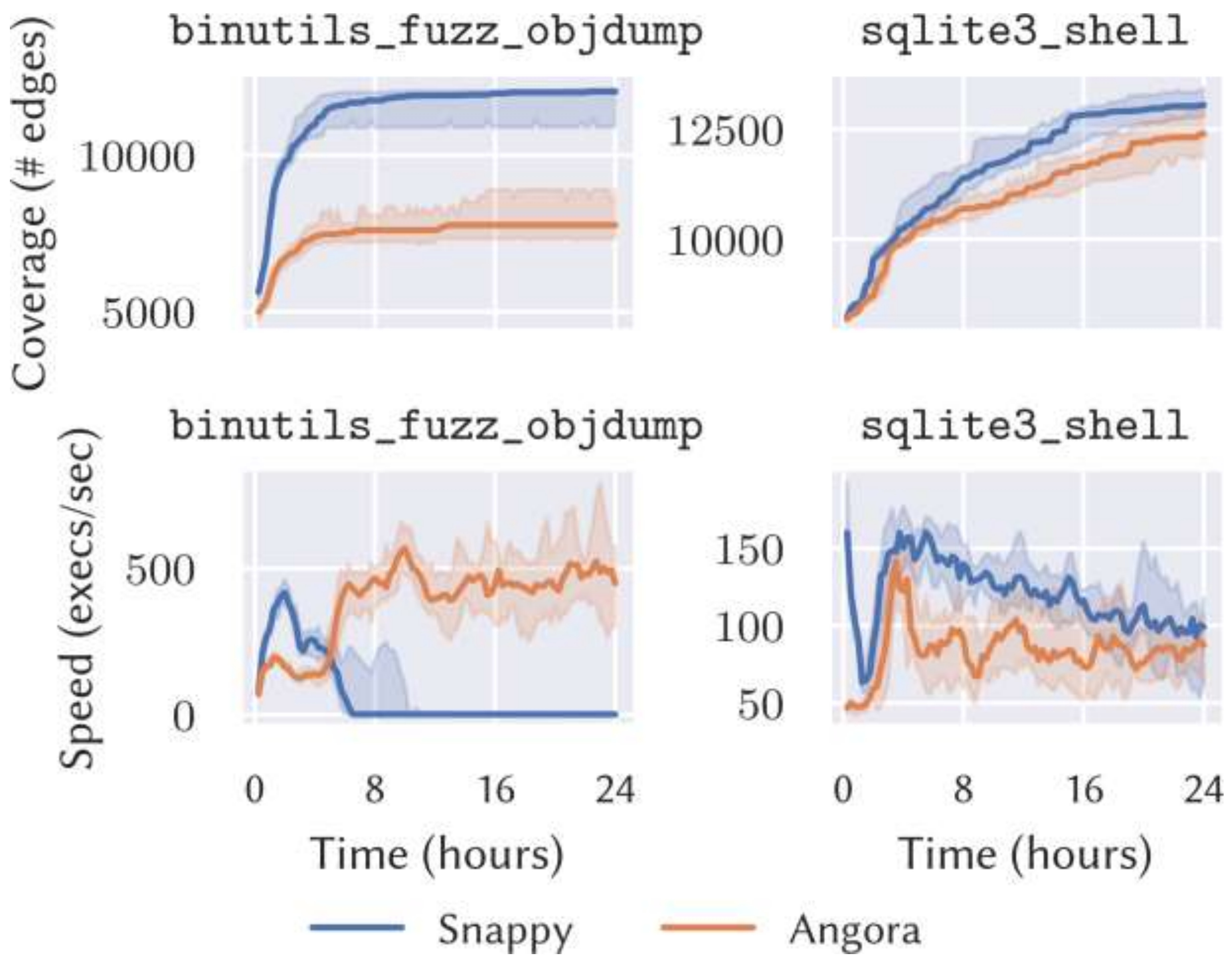


- Program initialization is redundant
- Input data is copied before use, but does not influence the execution
- Several mutation operators leave most of the input unchanged
- Pushing the snapshot into the execution will remove redundant operations

Applying Mutations to Snapshots

- Snapshot creation
  - Dynamic taint analysis to track which input bytes modify which memory bytes
  - Create snapshot when tainted byte controls branch
- Snapshot restore
  - Use taint to update modified input bytes in memory
- Taint tracking is expensive
  - Decide dynamically whether it is worth it, depending on extent of snapshot reuse

Evaluation



binutils_fuzz_objdump
sqlite3_shell
Coverage (# edges)
10000
5000
12500
10000
binutils_fuzz_objdump
sqlite3_shell
Speed (execs/sec)
500
0
150
100
50
0
8
16
24
Time (hours)
0
8
16
24
Time (hours)
Snappy
Angora

Conclusions

Conclusions

- Still plenty of opportunity to improve fuzzing
- Eliminating duplicate work is effective
  - General principle: cache and reuse partial results (memoization)
- Hardware can sometimes do cool tricks we never thought of
  - Any other ideas how to use a primitive that can very quickly compare two 4-byte values for equality where inequality is the common case?