Static Analysis as a Fuzzing Aid

Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, Anja Feldmann

Context

- We had good success against an SDN switch called OvS
- **6 CVEs** in about a month*

But, **poor test coverage** (< 5%)

* https://bshastry.github.io/2017/07/24/Fuzzing-OpenvSwitch.html

Summary

In a nutshell, I will tell you

- Why handwritten parsers still exist?
- Why thorough testing of handwritten parsers is challenging?
- Static analysis can improve test effectiveness
- Present evidence in favor

Handwritten Parsing Code Considered Dangerous

- Heartbleed old wine, new bottle
- **No memory safety** guarantees in C/C++

"... for complex [protocols] the recognition that matches the programmer's expectations can be equivalent to the 'halting problem'"

\- **Sassaman** et al. 2011

Why Handwritten Network Parsers in 2017?

Some educated guesses…

- **Legacy** code
- **Informal** specification
  - IETF RFCs are human readable
- **Multi-protocol** handling
- **Complex** protocol grammar
  - Hard to express as context-free specification

Network Analysis Tools

- Handwritten parsers **backbone** of network analysis tools
- Packet analyzers, NIDS etc.
  - Parse a few **hundred** network **protocols**

How do we test them **thoroughly**?

Limitations of Existing Techniques

Optimal seed selection problem

- How diverse should seeds be?
- How to obtain seeds that are sufficiently diverse?

Analysis precision vs run time

- How to scale up analysis while reducing false positives?

Our Proposal

Static analysis guided fuzzing

- Exploits complementary nature of SA and fuzzing
    - SA to find *what good seeds look like*
    - Fuzzing to find bugs
    - No false positives and potentially high coverage!

Challenges

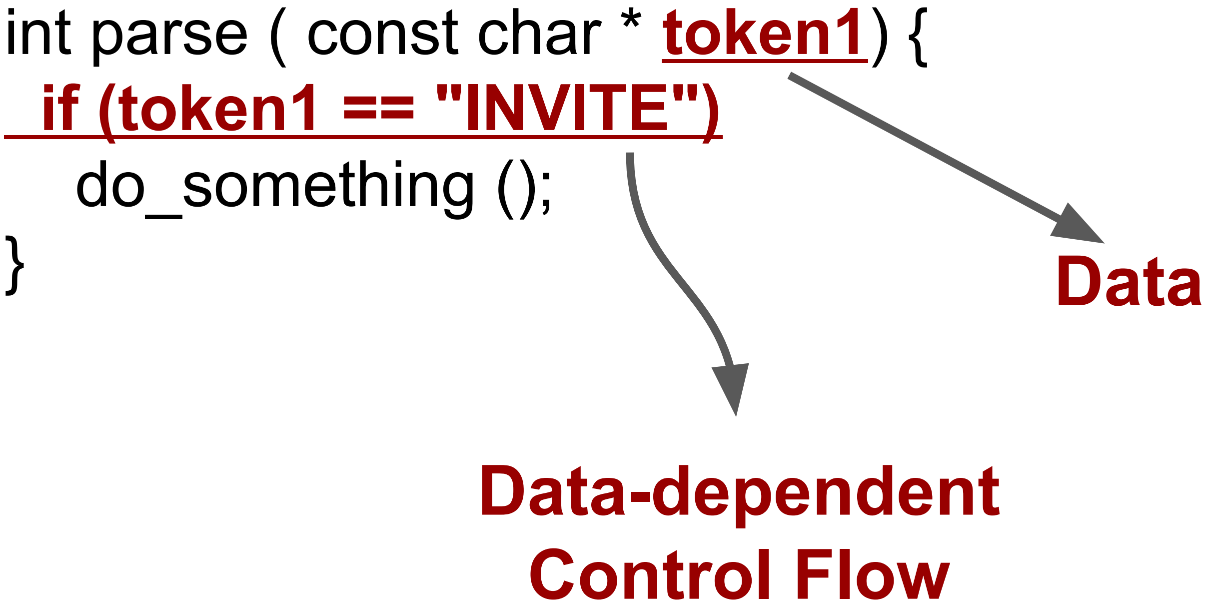
How do I look for protocol message fragments?

- Identify **tainted data-dependent** program control flow

What do seeds look like?

- From this, find
    - (Constant) **Tokens**
    - **Relation** between tokens
    - Partial **ordering** (if any) between tokens

Data-dependent Control Flow



```
int parse ( const char * token1) {
  if (token1 == "INVITE")
    do_something ();
}
```

Data

Data-dependent Control Flow

Tainted Control Flow

```
int parse ( const char * token1) {
  if (token1 == "INVITE")
    do_something ();
}

int main (int argc, char *argv[]) {
   parse(argv[1]);
   parse(“TEST”);
}
```

Tainted Input

Identifying Tainted APIs

- Requires forward slicing
  - Intractable for large programs
- Our proposal
  - Apriori database of known taint sinks
  - Based on SANS/CERT secure coding guidelines
  - May also be developer supplied

Now we know how to look for message fragments in source code…

How to build a dictionary?

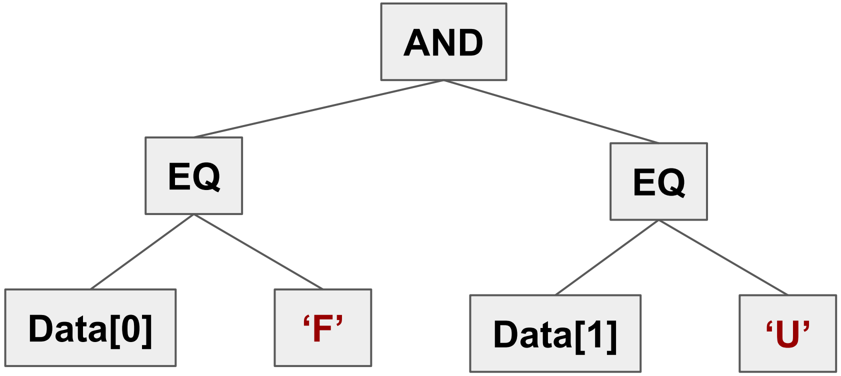
Dictionary Generation

- Constant tokens
    - **Syntactic** code analysis sufficient
    - **Fast** wrt compilation time
- Token relationship and ordering
    - Requires **semantic** code analysis
    - **Slow** wrt compilation time

Extracting Constant Tokens

```
bool FuzzMe(const uint8_t
*Data, size_t Size) {
    return Size >=1 &&
          Data[0] == 'F' &&
          Data[1] == 'U' &&
}
```

Source Code



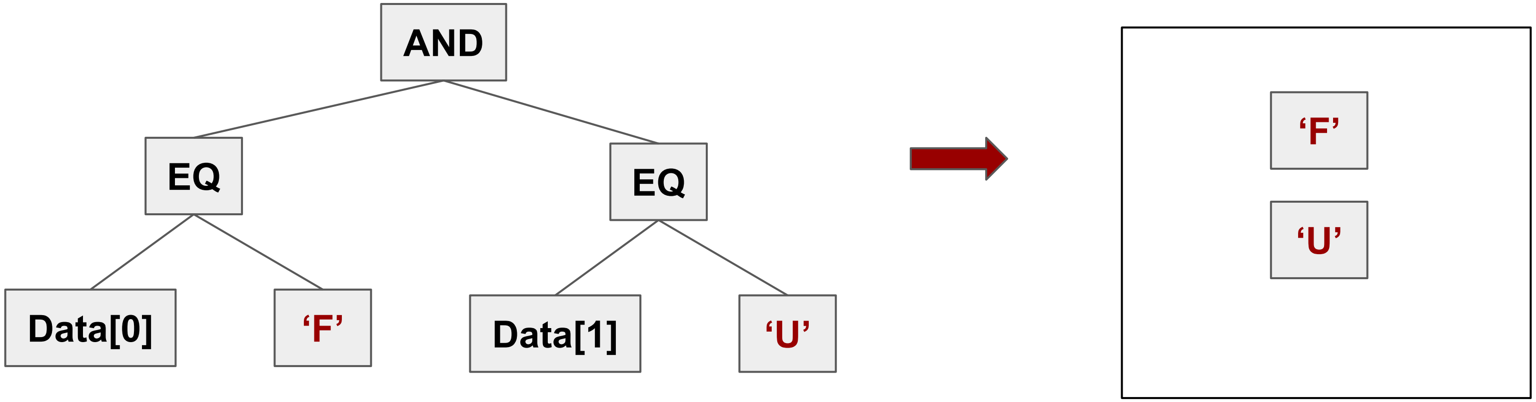
AST

Extracting Constant Tokens

Algorithm

- Make a pass over source code
- Obtain abstract syntax tree
- From AST, obtain constant tokens in “hot path”

Extracting Constant Tokens



AST

Dictionary

Inferring Token Relationship and Ordering

- In what **context** is a given constant token used?
    - E.g., “INVITE” **follows** “SIP 2.0”
- What do productions in the protocol grammar look like?
    - E.g., “SIP 2.0 INVITE”

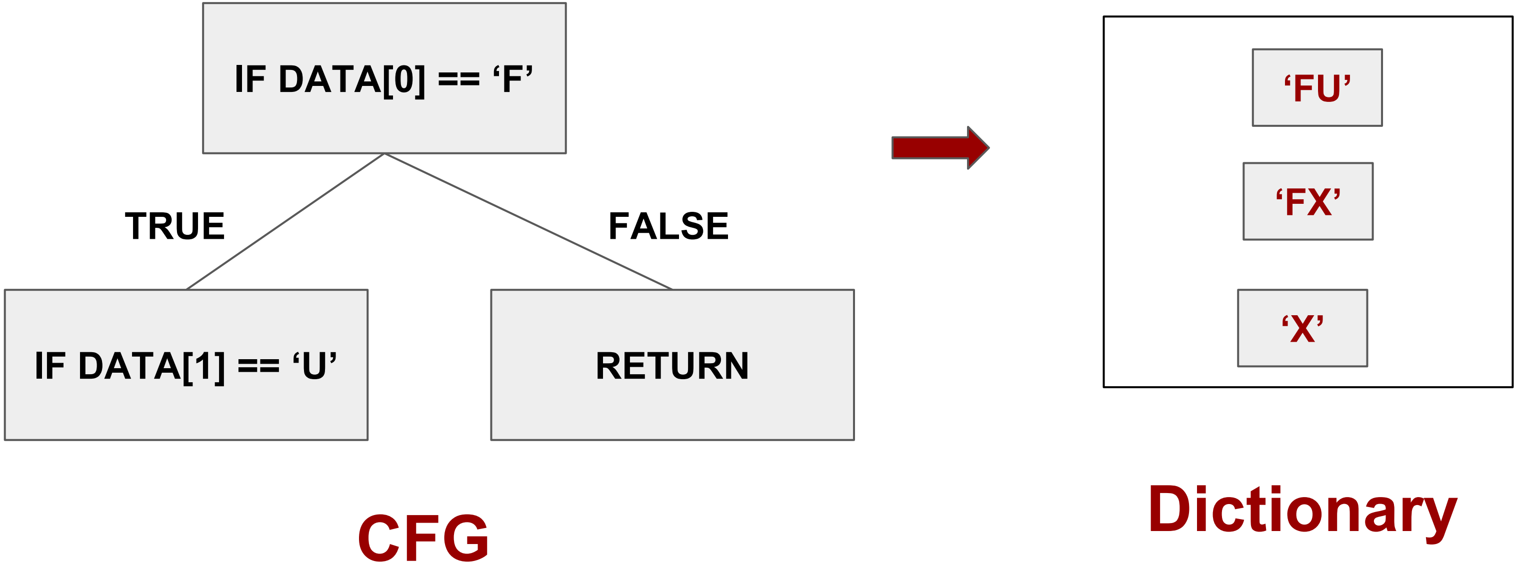
This requires **semantic** code analysis

Extracting Token Productions

Algorithm

- Make a pass over source code
- Obtain control flow graph
- From CFG, identify dependencies between tokens

Extracting Token Productions

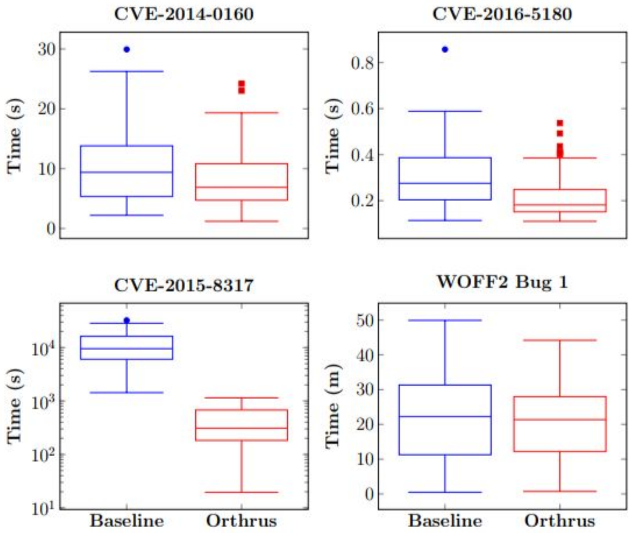


Evaluation

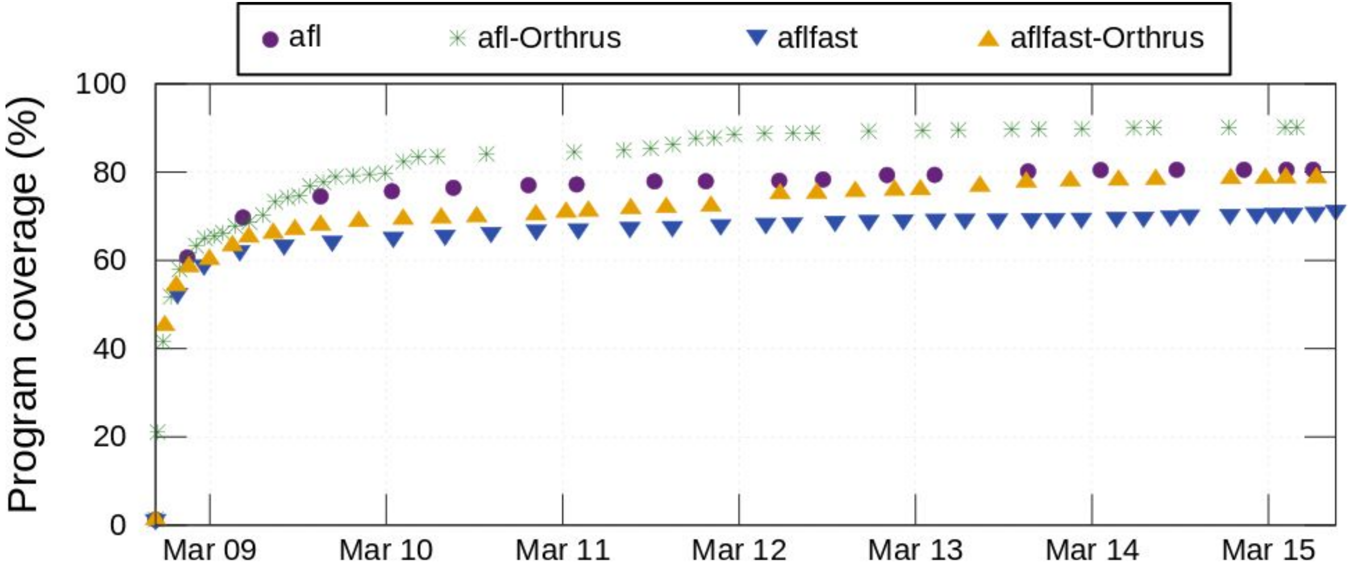
- Chose three state-of-the-art fuzzers
  - libFuzzer, afl-fuzz, afl-fuzz-fast [CCS'16]
- Methodology: Measure fuzzer findings with and without dictionary
- Both controlled and uncontrolled tests
  - Controlled: Time to find known vulnerabilities
  - Uncontrolled: Vulnerabilities and test coverage for production code

Results: Controlled Set Up

- openssl, c-ares, libxml2, woff2
- Orthrus consistently **reduce** time-to-vuln-exposure
- **High opportunity cost** when bug is in **parsing path**!



Results: Uncontrolled Set Up



tcpdump

Results: Number of Discovered Vulnerabilities

Software	afl	afl-Orthrus	aflfast	aflfast-Orthrus
tcpdump	15	26 (+10)	1	5 (+4)
nDPI	26	27 (+4)	24	17 (+1)

Found a new zero-day in **snort++** post submission!

Impact: tcpdump 4.9.2

- Fuzzed by eight independent teams
- 92 CVEs discovered in total
- We discovered 43 CVEs using Orthrus

We found **just under 50%** of them!

Conclusions

- Exhaustive testing of network parsers important
- Our heuristics capture protocol message fragments, feeding it to a fuzzer
- Static analysis can augment fuzzing effectively
  - Test coverage increased 10-15%
  - Tens of new zero-day vulnerabilities
  - Fast analysis, one-time cost

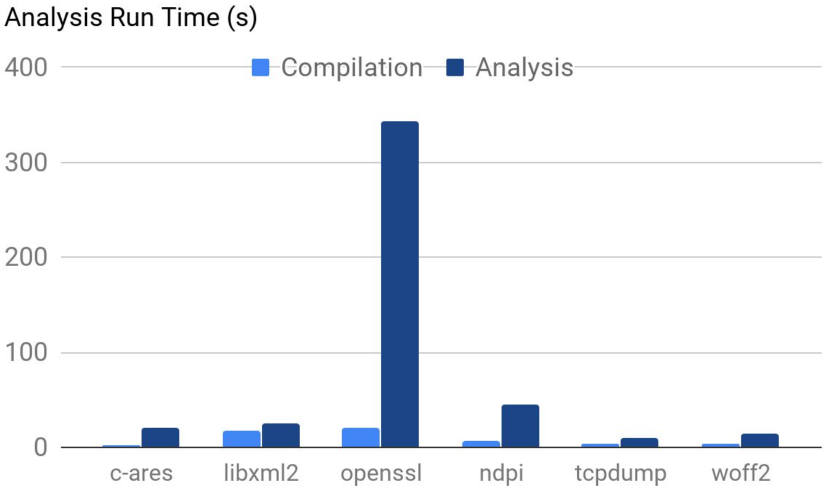
Future Work

- Scale up evaluation (parsers on GitHub!)
- Evaluate yacc generated parsers
- Port to Java-based parsers
- Automated parser test-case generation

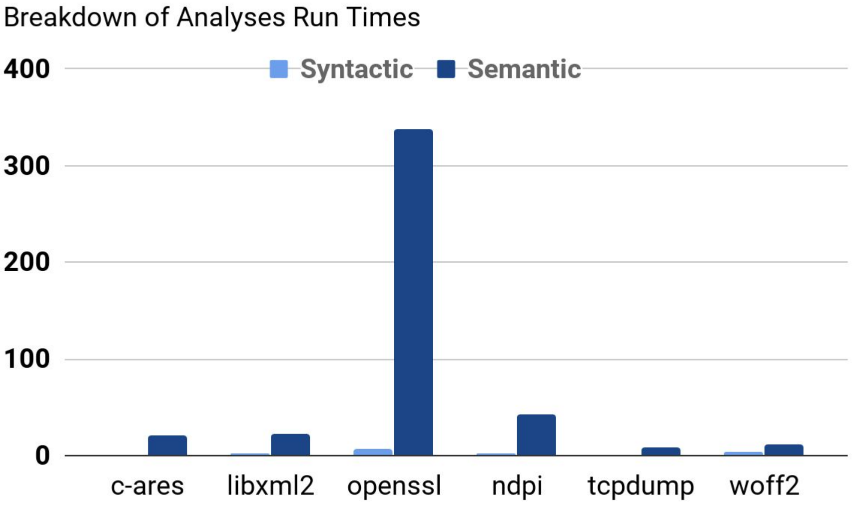
Questions?

Thank you!

Analysis Run Time



Syntactic vs Semantic Analysis Run Time



Test and Analysis Techniques

- Fuzz testing
  - Requires **diverse seeds** but provides actionable diagnostics
- Static analysis
  - Can analyse entire codebase but suffers from **false positives**