Fuzzing with Data Dependency Information

whoami

- Alessandro Mantovani
- PhD at Eurecom
- Currently work at Qualcomm
- Interests
    - Program [source|binary] analysis, fuzzing, ...

Agenda

- Background
  - Coverage
  - Fuzzing
  - Feedback mechanisms
- Research problem
- Methodology and Implementation
  - Technology
  - Data Dependency Graph construction
  - Data Dependency Graph filtering
  - Instrumentation
- Evaluations
- Future directions

Coverage: path coverage

- Coverage measures the amount of executed code that a certain input triggers
- Many metrics exist for this
- Intuitive idea: **path coverage**
  - All linearly independent paths present in a program (tests generation, symbolic execution)
- It can easily incur in state explosion
- Other?

Coverage: edge coverage

- De-facto standard for some popular dynamic analysis techniques (e.g., fuzzing)
- Basic Blocks in the Control Flow Graph (CFG) are instrumented
- Each couple of connected basic blocks identifies an **edge**
- When a new edge is discovered, we take note of this

Fuzzing

- Dynamic Analysis Technique proposed in the early ‘90
- A fuzzer executes a program many times
- For each execution it submits a mutated input to the program
    - White-box fuzzers: heavyweight code instrumentation (symbolic executor)
    - Black-box fuzzers: no code instrumentation
    - **Grey-box fuzzers: lightweight code instrumentation to produce a feedback**

Feedback

- It rates an input as “good” or “bad”
- It drives a fuzzer towards exploring more nested code and states
- Some examples
  - Basic Block Coverage
  - Edge Coverage (libFuzzer)
  - Edge Coverage + number of times a certain edge has been hit (AFL)

Feedback: AFL approach

- A bitmap stores the discoveries in terms of new edges
- Each index of the bitmap stores how many times a certain edge has been hit
- A new testcase is interesting if at least one of the entries has a new value
- AFL encodes an edge with the XOR between previous and current code block
- This is the index for the bitmap

Feedback: coverage-based

- Researchers proposed thousands of improvements to edge coverage
  - Context-sensitivity -> It considers the function calls
  - N-grams -> It considers N edges instead of 1
  - CmpCov -> It rewards the fuzzer for each matched byte for the comparisons
- **Coverage is not enough!**
  - **Bugs depend also on other factors such as the state OR different sub-approximations of coverage**

Feedback: alternative non coverage-based

- Alternative feedbacks research trend
- IJON -> manually annotate the source code to give the fuzzer program state info
- InvsCov -> adopt the *likely* invariants of a program as state feedback
- ParmeSAN -> Sanitizer oriented directed fuzzer (ASAN, UBSAN,..)

Research Question

- Alternative feedback mechanisms show promising results since they can uncover **different** bugs than traditional edge-coverage approaches
- RQ1 -> Can we find alternative program representations that sub-approximate the coverage in a different way?
    - **Data Dependency Graph (DDG)**
- RQ2 -> Are these representations sufficiently expressive for vuln discovery?
    - **DDG very often used for static software testing (SAST tools)**

Overall Idea

- We use DDG coverage as an alternative feedback mechanism
- This allows to stress different combinations of source-sink pairs that normally would not be fuzzed by edge-coverage
- Our goal is not to increase the reached code coverage, BUT, to trigger potentially different bugs
    - Thus we still rely on edge-coverage and we embed DDG coverage on top

Methodology and implementation: LLVM

- LLVM provides a compiler backend and API to implement custom passes
- SSA form
  - Each variable is assigned only once
  - All variables must be defined before their first use
- Easy to plug into other existing fuzzing engines (e.g., AFL++)

Methodology and implementation: AFL++

- State-of-the-art fuzzing engine at the time of writing
  - Today there's LibAFL
- It exports many instrumentation options
  - Edge-coverage
  - Context-sensitivity
  - N-grams
- We use it both for our implementation and then for our comparisons

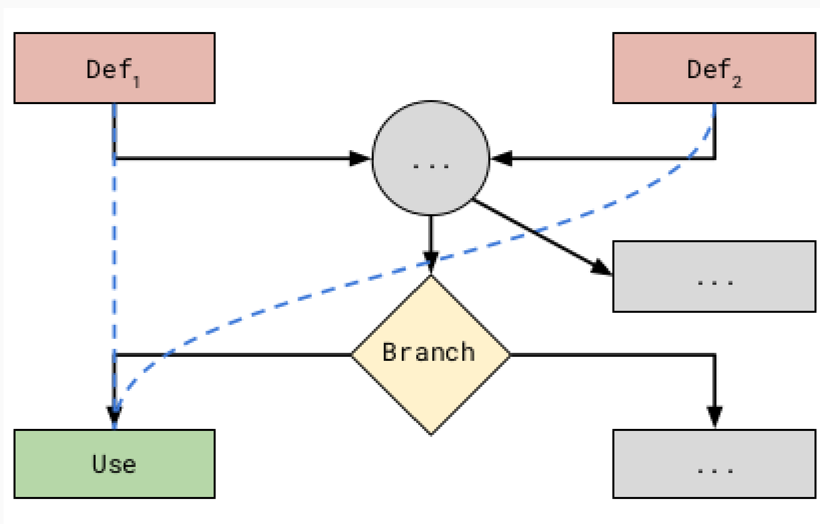
Methodology and implementation: DDG construction

- Intuitive idea -> Track use-def chains for each SSA variable
  - Cons -> too many dependencies that result in a poor feedback
- Focus on a subset of **interesting** instructions that reflect **when the actual data flow happens** at the binary level
  - Load (def) and Store (use)
  - Alloca (def)
  - Call (use)
- When we find a use, we recover its def and we add a new edge
  - An edge connects the two blocks that contain the instructions

Methodology and implementation: DDG filtering

- DDG instrumentation has to co-exist along with Edge Coverage
  - DDG gives us feedback about dependencies but is not good for deep code exploration
  - Edge Coverage does this job
- We have to remove all edges which are overlapping with CFG
  - Edges connecting two consequent basic blocks
  - Are we done?

Methodology and implementation: DDG filtering



- If a definition D has only **one** single use U this is already tracked by CFG (eventually)
- However, if a use U has **multiple** definitions D_1, .., D_n a normal fuzzer cannot distinguish the two
  - Phi-node variables

Methodology and implementation: DDG instrumentation

- Edge coverage feedback (AFL mechanism) + DDG feedback
- DDG edges different from CFG edges
  - We cannot simply XOR current and previous blocks as in the DDG they are not consequent
- In the function entry point we zero-initialize a variable for each basic block
- Moreover, the instrumentation sets this variable to a compile-time block identifier when we go through that specific code block

How our instrumentation looks like

- on_block_X, on_block_Y are the variables for the two blocks where we found a definition
- Block Z is where the use happens
- Only one of the two among X and Y has been hit (i.e., only one definition)
  - Thus, either on_block_X or on_block_Y is set to the corresponding random id
- With the XOR we account for this and we exclude the variables which are set to 0

```
// X, Y, Z are random IDs set at compile
    time
void function() {
    u16 on_block_X = 0;
    u16 on_block_Y = 0;

    int val;
    if (...) {
        // block X
        on_block_X = X;
        int a = load_a();
        val = a;
    } else {
        // block Y
        on_block_Y = Y;
        int b = load_b();
        val = b;
    }

    if (...) {
        ...
    } else {
        ...
    }

    if (val == 0) {
        // block Z
        u16 idx = on_block_X ^
                  on_block_Y ^
                  Z;
        __afl_area_ptr[idx]++;
        use(val);
    } else {
        ...
    }
}
```

Evaluation

- We measure different aspects of **DDFuzz** (our prototype)
  - Bugs, Coverage, Performances, Usability, ..
- Custom dataset
  - 5 trials of 24 hours each, ASan and UBSan
  - 5 compilers (high data dependency) vs 5 binary parsers (high code dependency)
- Fuzzbench
  - 20 trials of 23 hours each, 22 programs
- Previous dataset

Custom Dataset

Application	Package	Hash commit	KLOC	Command line	Sanitizers	Initial Queue
bison	bison	0ac1584	100	@@	ASAN	3
pcre2(*)	pcre2	65457aa	68		ASAN, UBSAN	3
c2m	mir	7670d7e	200	@@	ASAN, UBSAN	65
qbe	qbe	e0b94a3	10	@@	ASAN	52
faust	faust	f9aac26	115	@@	ASAN, UBSAN	27
readelf	binutils	68b975a	120	-s @@	ASAN	3
objdump	binutils	68b975a	120	-d @@	ASAN	3
libmagic(*)	file	d1ff3af	14		ASAN, UBSAN	3
tiff2pdf	libtiff	7d3b9da	63	@@	ASAN, UBSAN	10
openssl(*)	openssl	0d87763	234		ASAN	9

Comparison against Edge Coverage

- When should we use DDFuzz ?
- DD ratio
- Number of instrumented blocks in our DDG over total number of BBs
  - Strongly vs Weakly data dependent applications
- Does DDFuzz finds an higher number of bugs compared to edge coverage?
  - Depending on the DD ratio
- Are these bugs different from edge coverage ?
- What are the performances of the strongly data dependent programs?

Comparison against Edge Coverage and DD ratio

Target	DD ratio	DDFuzz	Edge	DDFuzz ∩ Edge	Slowdown
bison	15%	5	3	3	2%
pcre2	14%	30	28	13	6%
c2m	23%	26	26	23	21%
qbe	15%	5	3	2	2%
faust	20%	4	2	2	18%
Total	-	70	62	43	-
Geomean	17.06%	9.51	6.66	5.14	6%
readelf	7%	4	4	4	2%
objdump	7%	5	5	4	1%
file	5%	1	1	1	1%
tiff2pdf	8%	13	9	9	4%
openssl	5%	2	2	2	5%
Total	-	25	21	20	-
Geomean	6.28%	3.49	3.24	3.10	2%

Effects of our filter strategies

- We compare DDFuzz on two different DDG instrumentation
  - Before and after applying our filter strategies
- Performances increase of the 49.8%
- Bugs discovered are a superset
- Our filters are meaningful!

Comparison Against different Coverage approaches

- Ngrams-2 and Ngrams-4
    - These consider multiple edge and thus there could be an overlapping with DD fuzz as it considers basic blocks which are “far”
- Context sensitivity
    - Same as before
    - Previous papers show its benefits

Comparison Against different Coverage approaches

Target	DDFuzz	Ngram2	Ngram4	Ctx	DDFuzz ∩ Ngram2	DDFuzz ∩ Ngram4	DDFuzz ∩ Ctx
bison	5	5	4	3	3	3	3
pcre2	30	29	23	33	15	14	17
c2m	26	27	27	17	23	22	12
qbe	5	3	4	5	3	4	4
faust	5	2	2	1	2	2	1
Total	71	66	60	59	46	45	37
Geomean	9.94	7.48	7.23	6.09	5.73	5.93	4.76

No fuzzer outperforms the other ones BUT our DDFuzz always finds **different bugs**

Other usability metrics

- Queue explosion
  - Increase of x1.5 in the median case
  - Normally more than x8.0 results in queue explosion
- Lines and Functions coverage
  - Similar to Edge Coverage (average variation +0.2%)
  - Less than Context Sensitivity (-1.4% in the worst case)
  - DDFuzz explores mostly **the same surface but finds different bugs**

Fuzzbench: DD ratio

- We compare DDFuzz against EdgeCov
  - If we include other approaches all results are aggregated and we cannot separate each fuzzer
- We compute the dd ratio on the fuzzbench targets
  - 11 weakly data dependent apps (dd ratio <= 10%)
  - 11 strongly data dependent apps (dd ratio > 10%)
- Does the dd ratio correlate to an higher number of bugs ?

Fuzzbench: bugs

- 33 bugs are only detected by DDFuzz
- Among the 11 data dependent applications
  - DDFuzz finds more or different bugs in 7 cases
  - In 4 cases no bugs at all
- Among the 11 weakly data dependent programs
  - DDFuzz is a subset of EdgeCov
  - Except for arrow_parquet

Benchmark	Total bugs	Edge	DDFuzz	DDFuzz ∩ Edge	DD ratio
arrow_parquet-arrow-fuzz	105	93	86	74	1%
proj4_standard_fuzzer	0	0	0	0	2%
muparser_set_eval_fuzzer	0	0	0	0	3%
openh264_decoder_fuzzer	10	10	8	8	3%
aspell_aspell_fuzzer	0	0	0	0	4%
systemd_fuzz-varlink	0	0	0	0	4%
tpm2_tpm2_execute_command_fuzzer	0	0	0	0	4%
file_magic_fuzzer	0	0	0	0	7%
libgit2_objects_fuzzer	2	2	2	2	5%
grok_grk_decompress_fuzzer	4	4	2	2	7%
stb_stbi_read_fuzzer	11	11	11	11	8%
njs_njs_process_script_fuzzer	0	0	0	0	11%
php_php-fuzz-execute	21	13	16	8	11%
libxml2_libxml2_xml_reader_for_file_fuzzer	13	11	12	10	12%
libhtp_fuzz_htp	7	6	7	6	13%
matio_matio_fuzzer	22	20	21	19	14%
php_php-fuzz-parser-2020-07-25	13	12	12	10	14%
poppler_pdf_fuzzer	5	4	3	2	14%
libarchive_libarchive_fuzzer	0	0	0	0	15%
zstd_stream_decompress	0	0	0	0	15%
usrsctp_fuzzer_connect	0	0	0	0	17%
libhevc_hevc_dec_fuzzer	3	1	3	1	19%
Total	214	187	183	150	-
Geomean	9.72	7.98	8.20	6.38	7%

Fuzzbench: other considerations

- DD ratio correlates with the number of bugs
- In terms of coverage EdgeCov mostly performs better than DDFuzz (17 out of 22 cases)
    - This is expected
- Still, DDFuzz finds bugs even though it explores less code

Third Dataset

Target	DD ratio	DDFuzz	Edge	DDFuzz ∩ Edge	Slowdown	P-value
nasm	3%	4	4	4	1%	0.12
sqlite	7%	1	2	1	15%	0.001
lua	11%	9	2	2	20%	0.005
boolector	14%	3	1	1	6%	0.13
mruby	17%	3	3	3	35%	0.45
tcc	12%	11	8	8	7%	0.05
Total	-	31	20	19	-	-
Geomean	9.3%	3.9	2.6	2.4	8.7%	-

- Dataset previously adopted by other works
  - Results confirmed

Limitations and future directions

- DDFuzz works only when there is high data dependency
  - Not a real limitation actually
- Avoid edge collisions
- Binary-only fuzzing
  - Recover DDG from the binary is less precise
- Extend to other fuzzers (Libfuzzer, Hongfuzz, LibAFL)

Conclusions

- Current state-of-the-art fuzzers are good at detecting bugs that depend on code reachability
- BUT alternative coverage approximations bring advantages
- DDFuzz implements a possible coverage approximation based on the DDG
- DD ratio tells us when to adopt it
  - DDFuzz can reach many different program points
  - And consequently find different bugs
- Hopefully, it will be integrated into existing fuzzing systems

Thanks for your attention

Questions?

Bug Types

Bug Class	Total	DDFuzz Only
Heap BOF	38	11
Global BOF	5	3
Stack BOF	1	0
Heap UAF	4	2
Stack Overflow	19	11
Invalid Ptr Deref	11	3
Invalid Allocation Size	1	0
ABRT	2	1
ILL	36	5

- Heap BOF and Stack overflow
  - Top detected bugs

A bug case study

- We select a bug discovered only by DDFuzz
  - An overflow in tcc, a compiler
- Re-constructed the testcase tree
  - 168 mutations (133 havoc + 35 splicing)
- Which mutations generate novelty for traditional EdgeCov ?
- 3 out of 168 do NOT generate novelty
- Thus they got discarded from EdgeCov but retained from DDFuzz