Fuzzing for Automated Vulnerability Discovery

Lisa Overall

02 February 2024

Outline

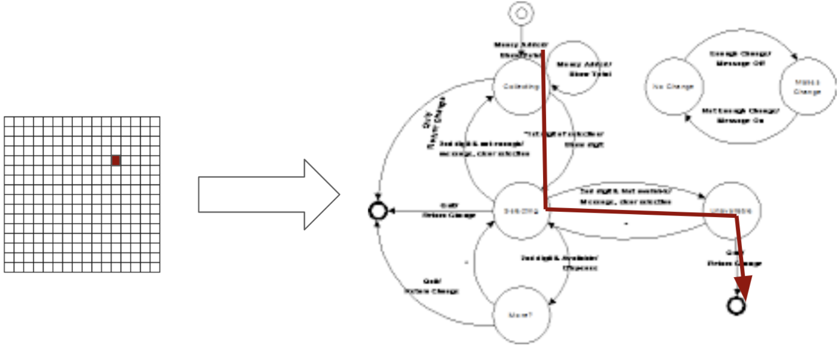
1. Motivation
2. What is fuzzing? (35 years ago)
3. Mutational fuzzing (10 years ago)
4. Research topics in fuzzing (to present)

Outline

1. **Motivation**
2. What is fuzzing? (35 years ago)
3. Mutational fuzzing (10 years ago)
4. Research topics in fuzzing (to present)

What are the components of a program?

- Input drawn from some input space (e.g., stdin, network state)
- States + transitions between them
- Outputs and/or side effects
- Termination conditions



Software security in a nutshell

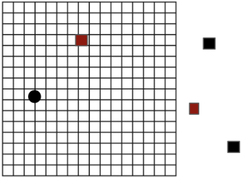
Two important questions:

1. What inputs cause “bad” behavior?
2. When is a state “bad”?

Security engineering (simplified)

Unrestricted input

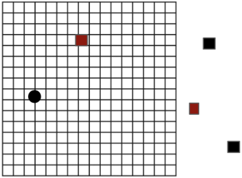
- Anything can happen
- Minimal understanding of what code "should" do



Security engineering (simplified)

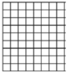
Unrestricted input

- Anything can happen
- Minimal understanding of what code "should" do



Option 1: Restrict inputs

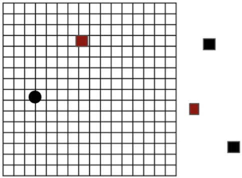
- Typed language
- Delete code
- Privilege separation



Security engineering (simplified)

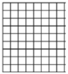
Unrestricted input

- Anything can happen
- Minimal understanding of what code “should” do



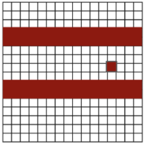
Option 1: Restrict inputs

- Typed language
- Delete code
- Privilege separation



Option 2: Test inputs

- Unit tests
- Fuzzing
- Property tests



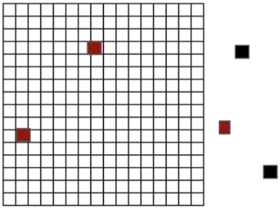
Levels of testing maturity

Level 0: No tests

- Developers think really hard and don't introduce bugs into the codebase!

Level 1: Try a few inputs

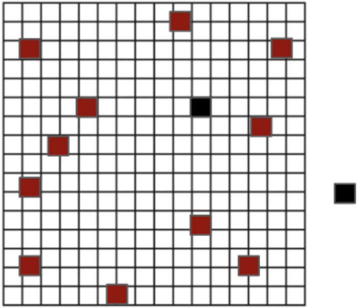
- Developers enumerate some common-sense checks and write unit tests.



Levels of testing maturity

Level 2: Try lots of random inputs

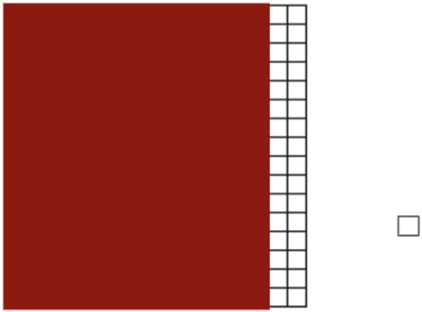
- Fuzzers, property-based testing
- Tons of papers, talks, and libraries
- Surprisingly effective [Din18]
- Fuzzing is gaining industry acceptance



Phases of testing maturity

Level 3: Test all the inputs

- Verification, symbolic execution
- Endgame, but not a cure-all
- Mostly rejected as impractical
- Incredible when it works



Outline

1. Motivation
2. **What is fuzzing? (35 years ago)**
3. Mutational fuzzing (10 years ago)
4. Research topics in fuzzing (to present)

It was a dark and stormy night...

Prof. Barton Miller, UWM [Mil88, MFS90]

Exercise

Adapted from Zeller et al.'s "The Fuzzing Book" [ZGB+24].

In Python 3,

1. Write a function that generates a random string of up to N ASCII characters.
2. Generate a random string, send it to the `bc` utility, and obtain the results.
3. Run 100 trials of the previous step. What do you observe about the results?

Discussion

Thinking about probability

- How likely is it to get a valid input for `bc` with the input generation method we used?

Discussion

Thinking about probability

- How likely is it to get a valid input for `bc` with the input generation method we used?
- How likely is it that we get the same input twice?

Discussion

Thinking about probability

- How likely is it to get a valid input for `bc` with the input generation method we used?
- How likely is it that we get the same input twice?

Program outputs and side effects

- Return code semantics are not universal.

Discussion

Thinking about probability

- How likely is it to get a valid input for `bc` with the input generation method we used?
- How likely is it that we get the same input twice?

Program outputs and side effects

- Return code semantics are not universal.
- Suppose we fuzz `rm -R`. What is the probability of generating an input that deletes the entire filesystem?

Discussion

Thinking about probability

- How likely is it to get a valid input for `bc` with the input generation method we used?
- How likely is it that we get the same input twice?

Program outputs and side effects

- Return code semantics are not universal.
- Suppose we fuzz `rm -R`. What is the probability of generating an input that deletes the entire filesystem?
- Inconsistency in observed behavior for a given input across campaigns might indicate that there are changes in some uninstrumented state upon which the program depends.

Congratulations, you've written your first fuzzer!

What are the components of a fuzzer?

- Input generation
- Harness: some way of sending those inputs to the program or library under test
- Instrumentation: some way to collect data about the program execution for a given input
- Some way to decide whether a particular (input, execution) pair is "interesting"
- (Maybe) some way to track the progress of data collection (e.g., coverage)

Outline

afl [Zal13a]

[1]

- Released by Michał Zalewski in November 2013 (aka lcamtuf; formerly of Google, now at Snap)
- “a de-facto standard for fuzzing” ([FMMB23])
- Bundled with tools to help monitoring fuzzing campaigns and analyze results

[1] https://en.wikipedia.org/wiki/File:Conejillo_de_indias.jpg

afl: input generation

Strategy: genetic algorithm

Define a fitness function over (input, execution) pairs.
Given a corpus containing at least one sample input to the program under test, add all inputs to a queue.

1. Load next test case from the queue.
2. Minimize the test case:
   - Remove bytes from input
   - Check if trimmed input has equal fitness
   - If fitness preserved, discard original input and save trimmed input in queue
3. Mutate test case. Add mutants with greater fitness to the queue. Save mutants resulting in crashes or hangs.
4. If campaign termination conditions (e.g., timeout) not met, return to Step 1.

afl: input generation

Mutation

1. Deterministic stage:
   - Flip 1+ bits
   - Increment / decrement {8, 16, 32}-bit integers in {little, big}-endian encodings
   - Overwrite input chunks with "interesting" values (e.g., zero, {max, min} (u)int{8, 16, 32} in {little, big}-endian encodings)
   - Replace parts of input with data from user-supplied dictionary and/or auto-detected tokens (e.g., magic bytes, keywords)

afl: input generation

Mutation

2. Havoc stage: Apply 2-128 mutations, including:
   - Deterministic mutations (above)
   - Random byte replacement
   - Bytestring of length N replaced by N repetitions of single byte
   - Bytestring deletion
   - Bytestring duplication
   - (As a last resort if queue exhausted without increasing fitness) Splice two inputs together

afl: harnessing

Two standard ways to feed inputs to program:

1. `stdin`
2. File

Vulnerability researcher's task: write a program that reads from stdin or a file, then passes the resulting input to the code you're interested in testing.

May need to mock/stub initial state - be careful about implicit assumptions!

afl: instrumentation

What is a graph?

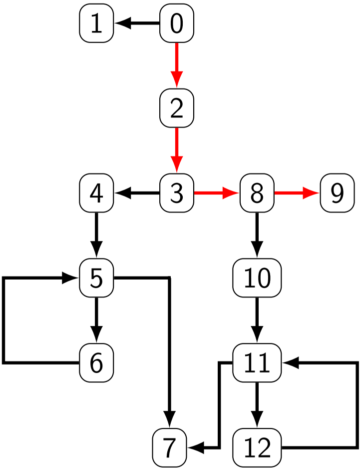
Definition

A **graph** is a pair G = (V, E) , where V is a set whose elements are called *vertices* and E is a set whose elements are paired vertices, called *edges*. If the elements of E are ordered pairs, we call the graph *directed*; otherwise, we call it *undirected*.

Definition

A **path** in a graph is a sequence of edges which joins a sequence of (distinct) vertices. Paths in directed graphs have an added restriction: the edges must all be directed in the same direction.

afl: instrumentation



$$G = (V, E)$$

$$V = \{0, 1, ..., 12\}$$

$$E = \{(0,1), (0,2), (2,3), (3,4), (3,8), (8,9), (8,10), (10,11), (11,12), (12,11), (11,7), (4,5), (5,6), (6,5), (6,7)\}$$

Node 9 is **reachable** from node 0 by path $((0,2), (2,3), (3,8), (8,9))$.

afl: instrumentation

Control-flow graph:

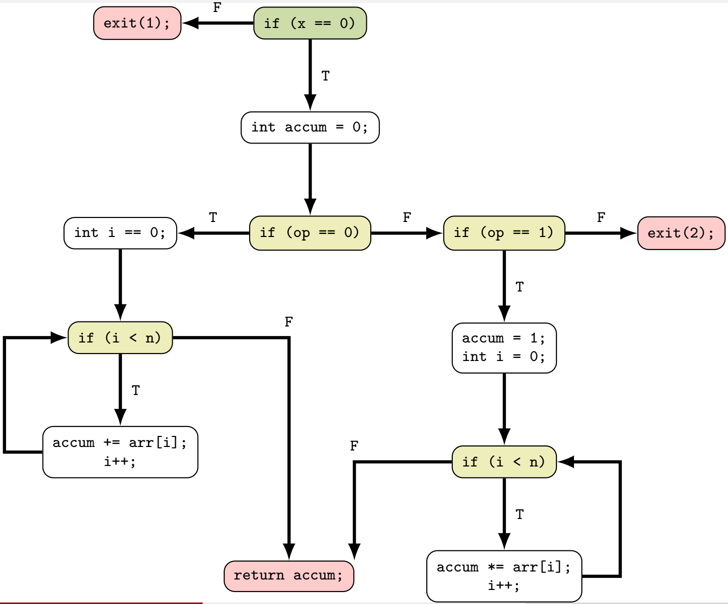
- Nodes - *basic blocks* of a program
- Edges - *control flow* between basic blocks

afl: instrumentation

```
foo:
.LFB6:
        .cfi_startproc
        endbr64
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        sub     rsp, 48
        mov     DWORD PTR -20[rbp], edi
        mov     DWORD PTR -24[rbp], esi
        mov     QWORD PTR -32[rbp], rdx
        mov     DWORD PTR -36[rbp], ecx
        cmp     DWORD PTR -20[rbp], 0
        jne     .L2
        mov     edi, 1
        call    exit@PLT
.L2:
        mov     DWORD PTR -12[rbp], 0
        cmp     DWORD PTR -24[rbp], 0
        jne     .L3
        mov     DWORD PTR -8[rbp], 0
        jmp     .L4
.L5:
        mov     eax, DWORD PTR -8[rbp]
        cdqe
        lea     rdx, 0[0+rax*4]
        mov     rax, QWORD PTR -32[rbp]
        add     rax, rdx
        mov     eax, DWORD PTR [rax]
        add     DWORD PTR -12[rbp], eax
```

```
        cmp     eax, DWORD PTR -36[rbp]
        jl      .L5
        jmp     .L6
.L3:
        cmp     DWORD PTR -24[rbp], 1
        jne     .L7
        mov     DWORD PTR -12[rbp], 1
        mov     DWORD PTR -4[rbp], 0
        jmp     .L8
.L9:
        mov     eax, DWORD PTR -4[rbp]
        cdqe
        lea     rdx, 0[0+rax*4]
        mov     rax, QWORD PTR -32[rbp]
        add     rax, rdx
        mov     eax, DWORD PTR [rax]
        mov     edx, DWORD PTR -12[rbp]
        imul    eax, edx
        mov     DWORD PTR -12[rbp], eax
        add     DWORD PTR -4[rbp], 1
.L8:
        mov     eax, DWORD PTR -4[rbp]
        cmp     eax, DWORD PTR -36[rbp]
        jl      .L9
        jmp     .L6
.L7:
        mov     edi, 2
        call    exit@PLT
.L6:
        mov     eax, DWORD PTR -12[rbp]
        leave
        .cfi_def_cfa 7, 8
```

afl: instrumentation



afl: instrumentation

- With source code: drop-in compiler replacement adds some code at each branch point to track edge coverage[2]
- Without source code: on-the-fly instrumentation via binary translator (e.g. QEMU, DynamoRIO, PIN)[3]
- NB: other tools (e.g., `afl-cov` [Ras15]) necessary for human-interpretable coverage reports from AFL results

[2] See https://afl-1.readthedocs.io/en/latest/about_afl.html#coverage-measurements

[3] See https://afl-1.readthedocs.io/en/latest/about_afl.html#binary-only-instrumentation

afl: instrumentation

Sanitizers [Inc12]: compiler passes that insert instrumentation to detect bugs at run-time

Examples:

- Address sanitizer (ASAN): detects addressability issues, e.g., buffer overflows [4]
- Memory sanitizer (MSAN): detects use of uninitialized memory
- Undefined behavior sanitizer (UBSan)

Sanitizers can be used alone or in combination (see [Zal13b], [Mor20] for caveats)

[4] See https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm

afl: interesting inputs

- Input causes a crash
- Input causes a hang
- Input causes a new edge to be covered

afl: campaign progress

- Edge coverage over time
- Unique crashes / hangs

Exercise

1. Run the buffer overflow example.
2. Run the Heartbleed example. Adapted from Michael Macnair's AFL workshop [Mac17].

Outline

1. Motivation
2. What is fuzzing? (35 years ago)
3. Mutational fuzzing (10 years ago)
4. **Research topics in fuzzing (to present)**

Input generation

Greybox fuzzing leverages lightweight instrumentation to prioritize further program exploration.

- "Shape aware"
- Novelty generally based on strategy for selecting next input
- Examples: AFL (new coverage), AFLFast [BPR16] (unusual path), AFLGo [BPNR17] (path close to uncovered basic blocks)

Input generation

Grammar-based fuzzing: Use specification for input language to produce syntactically valid inputs. Examples:

- CSmith [YCER11] for compilers
- Fuzzilli [GKB+23], LangFuzz [HHZ12] for JavaScript interpreters
- H26FORGE [VCS23] for video codecs
- Grammarinator [HKG18] for programs with `antlr`-specified grammars

Input generation

Fuzzing with the help of constraint solvers

Concolic fuzzing (“concrete + symbolic”)

- Accumulate constraints on inputs by tracking conditionals during execution
- Novelty generally based on strategy for appealing to the constraint solver
- Examples: DART [GKS05], SAGE [GLM12], CUTE [SMA05], EXE [CGP+08], Driller [SGS+16], MAYHEM [CARB12]

Harnessing

FUZZ ALL THE THINGS!

- Auto-harnessing via program synthesis techniques
- Handling non-standard inputs (e.g., firmware rehosting)

Instrumentation

- Source code available: program-specific sanitization
- Binary only: using dynamic binary instrumentation, binary lifting techniques to support new architectures, add finer-grained instrumentation
- Improving performance (executions/sec)
- Running fuzzers at scale (e.g., ClusterFuzz [Inc19])
- Running fuzzers in ensemble (e.g. EnFuzz [CJM+19], CollabFuzz [OGJ+21])

Interesting inputs, campaign progress

Directed (aka search-based) fuzzing: measuring novelty and/or progress via alternative metrics to coverage. Examples:

- IJON [ASAH20]
- FuzzFactory [PLS$^+$19]
- GOLLUM [HMK19]

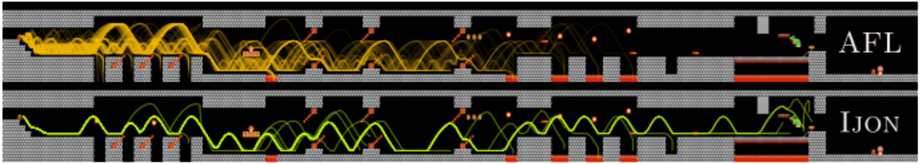


Fig. 1: AFL and AFL + IJON trying to defeat Bowser in Super Mario Bros. (Level 3-4). The lines are the traces of all runs found by the fuzzer.

Fuzzer evaluation

From an art to a science

- Reproducibility and statistical significance [KRC+18, HGM+21]
- Comparability: FuzzBench [MSS+21]
- Software engineering: libAFL [FM24]

Conclusion

What is the “best” fuzzer?

Conclusion

What is the "best" fuzzer?

The fuzzer that you've tailored
to the program under test
and your analysis objectives!

References I

Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz.
Ijon: Exploring deep state spaces via fuzzing.
In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612, 2020.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury.
Directed greybox fuzzing.
In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

References II

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury.
Coverage-based greybox fuzzing as markov chain.
In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley.
Unleashing mayhem on binary code.
In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler.
Exe: Automatically generating inputs of death.
ACM Trans. Inf. Syst. Secur., 12(2), dec 2008.

References III

Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su.
EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers.
In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.

Artem Dinaburg.
Fuzzing like it's 1989, Dec 2018.

Andrea Fioraldi and Dominik Maier.
The libafl fuzzing library, 2024.

References IV

Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti.
Dissecting american fuzzy lop: A fuzzbench evaluation.
ACM Trans. Softw. Eng. Methodol., 32(2), mar 2023.

Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns.
Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities.
In *Network and Distributed Systems Security (NDSS) Symposium*, 2023.

Patrice Godefroid, Nils Klarlund, and Koushik Sen.
Dart: directed automated random testing.
In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

References V

Patrice Godefroid, Michael Y. Levin, and David Molnar.
Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft.
Queue, 10(1):20–27, jan 2012.

Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking.
Seed selection for successful fuzzing.
In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 230–243, New York, NY, USA, 2021. Association for Computing Machinery.

Christian Holler, Kim Herzig, and Andreas Zeller.
Fuzzing with code fragments.
In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, August 2012. USENIX Association.

References VI

Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy.
Grammarinator: a grammar-based open source fuzzer.
In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 45–48, New York, NY, USA, 2018. Association for Computing Machinery.

Sean Heelan, Tom Melham, and Daniel Kroening.
Gollum: Modular and greybox exploit generation for heap overflows in interpreters.
In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1689–1706, New York, NY, USA, 2019. Association for Computing Machinery.

Google Inc.
sanitizers, 2012.

References VII

Google Inc.
Clusterfuzz, feb 2019.

George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks.
Evaluating fuzz testing.
In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

Michael Macnair.
Fuzzing with afl workshop, 2017.

Barton P. Miller, Lars Fredriksen, and Bryan So.
An empirical study of the reliability of unix utilities.
Commun. ACM, 33(12):32–44, dec 1990.

References VIII

Barton P. Miller.
Cs736 project list, 1988.

Antonio Morales.
Fuzzing software: advanced tricks (part 2), jul 2020.

Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya.
Fuzzbench: an open fuzzer benchmarking platform and service.
In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1393–1403, 2021.

References IX

Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos.
Collabfuzz: A framework for collaborative fuzzing.
In *Proceedings of the 14th European Workshop on Systems Security*, EuroSec '21, page 1–7, New York, NY, USA, 2021. Association for Computing Machinery.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar.
Fuzzfactory: domain-specific fuzzing with waypoints.
Proc. ACM Program. Lang., 3(OOPSLA), oct 2019.

Michael Rash.
afl-cov - afl fuzzing code coverage, 2015.

References X

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna.
Driller: Augmenting fuzzing through selective symbolic execution.
In *NDSS*, volume 16, pages 1–16, 2016.

Koushik Sen, Darko Marinov, and Gul Agha.
Cute: a concolic unit testing engine for c.
In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 263–272, New York, NY, USA, 2005. Association for Computing Machinery.

References XI

Willy R. Vasquez, Stephen Checkoway, and Hovav Shacham.
The most dangerous codec in the world: Finding and exploiting vulnerabilities in h.264 decoders.
In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6647–6664, Anaheim, CA, August 2023. USENIX Association.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr.
Finding and understanding bugs in c compilers.
In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

Michał Zalewski.
american fuzzy lop - a security oriented fuzzer, 2013.

References XII

Michał Zalewski.
notes for asan, 2013.

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler.
Fuzzing: Breaking things with random inputs.
In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024.
Retrieved 2024-01-18 18:11:45+01:00.