Marcel Garus
betreut von **Patrick Rein**

HPI, SoSe 2022/23
Fortgeschrittene Programmierwerkzeuge

COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN-MADISON

CS 736
Fall 1988

Bart Miller

Project List

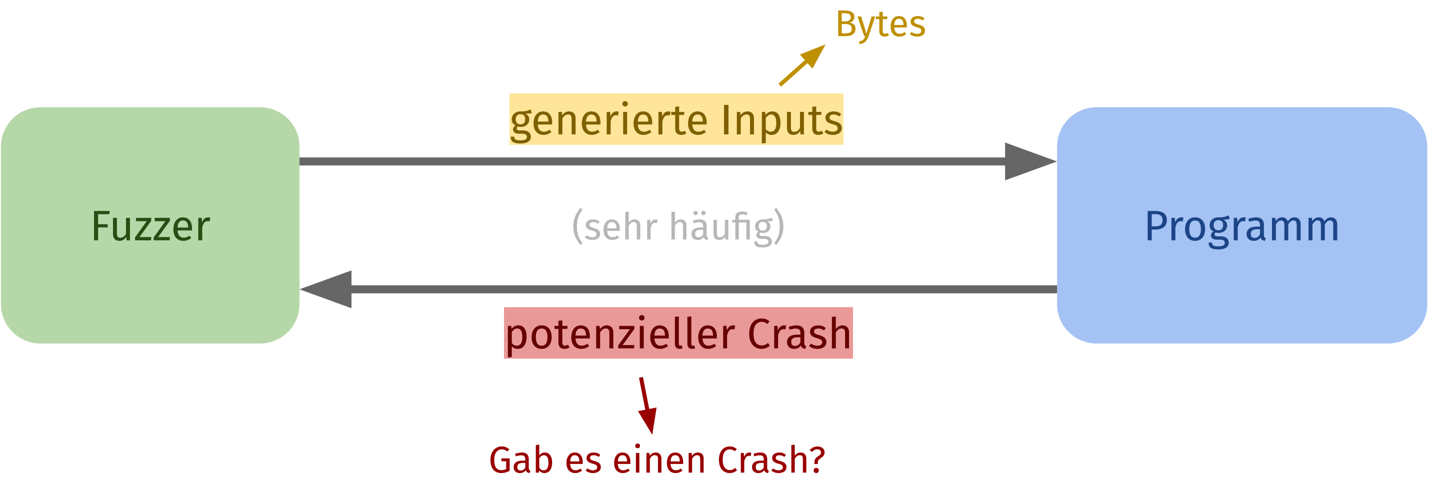
(Brief Description Due: Wednesday, October 26)
(Midway Interview: Friday, November 18)
(Final Report Due: Thursday, December 15)

Projects

(1) *Operating System Utility Program Reliability – The Fuzz Generator:* The goal of this project is to evaluate the robustness of various UNIX utility programs, given an unpredictable input stream. This project has two parts. First, you will build a *fuzz* generator. This is a program that will output a random character stream. Second, you will take the fuzz generator and use it to attack as many UNIX utilities as possible, with the goal of trying to break them. For the utilities that break, you will try to determine what type of input cause the break.

Fuzzing

Automatisiertes Testen von Code mit vielen Eingaben.



An Empirical Study of the Reliability of UNIX Utilities
von Miller, Fredriksen, So

Beispiel: URL-Encoding

`https://www.google.com/search?q=Ist+Fuzzing+schick%3F`

URL-encoded

`Ist Fuzzing schick?` → `Ist+Fuzzing+schick%3F`

`(3+4)*2` → `%283%2B4%29%2A2%0A`

RFC 1738 §2.2

```rust
/// Replaces '+' with ' ' and '%xx' with the character with hex number xx.
fn url_decode(input: &str) -> Option<String> {
    let input = input.chars().collect_vec();
    let mut output = "".to_string();
    let mut i = 0;

    while i < input.len() {
        let char_to_add = match input[i] {
            '+' => ' ',
            '%' => {
                let high = hex_to_decimal(input[i + 1])?;
                let low = hex_to_decimal(input[i + 2])?;
                i += 2;
                number_to_char(high * 16 + low)?
            }
            c => c,
        };
        output.push(char_to_add);
        i += 1;
    }
    Some(output)
}
```

Ist+Fuzzing+schick%3F

%283%2B4%29%2A2%0A

URLs+sind+weird

%3f

Hi%

Demo

Systeme robuster machen

Statische Analysen
untersuchen Code ohne ihn auszuführen

Typsysteme, Linter, …

beweisen Eigenschaften

kann nicht alle Fehler finden

(meist) bottom-up

Dynamische Analysen
führen Code aus und erlangen dadurch Erkenntnisse

Tests, **Fuzzing**, …

zeigen Eigenschaften empirisch

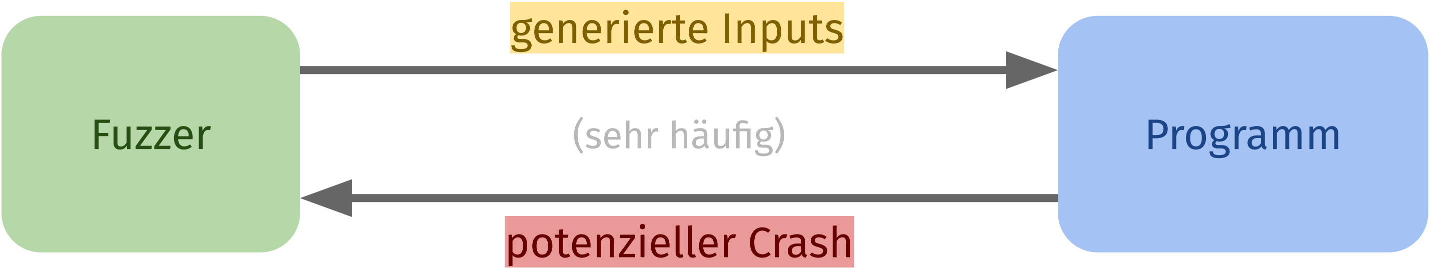
theoretisch alle Fehler auffindbar

top-down

Typische Anwendung in sicherheitsrelevanter Software: Fuzzing wird häufig bei Trust Boundaries eingesetzt, z.B. beim Parsen von Inputs.

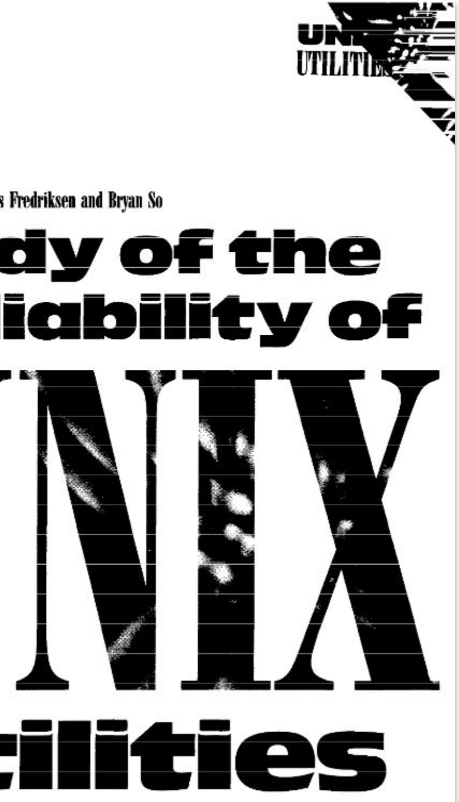
Fuzzing

Wie werden Inputs generiert?



Random Fuzzing

Random Fuzzing




“While our testing strategy sounds somewhat naive, its

command that began with 'o%8f.'" Since it does not find such a command, csh forms an error message string of the form: "o%8f: Event not found." This string is passed to the error-printing routine, which uses the string as the first parameter to the printf() function. The first parameter to printf() can include format items, denoted by a "%." The "%8f" describes a floating point value printed in a field that is 8 characters wide. Each format item expects an additional parameter to printf(), but in the csh error, none is supplied (or expected). This string was generated during the normal random testing.

The second example string follows the same path, but causes csh to try to print the floating point value in a field that is 888,888,888 characters wide. The seemingly infinite loop is the printf() routine's attempt to pad the output field with sufficient leading space characters. This second string was one that we generated by hand after discovering the first string.

Both of these errors could be prevented by substituting the printf() call with a simple string printing routine (such as puts()). The printf() was used for historical reasons related to space efficiency. The error-printing routine assumed that it would always be passed strings that were safe to print.

Bad Error Handler

Sometimes the best intentions do not reach completion. The units program detects and traps floating point arithmetic errors. Unfortunately, the error recovery routine only increments a count of the number of errors detected. When control is returned to the faulty code, the error recurs, resulting in an infinite loop.

Signed Characters

The ASCII character code is designed so that codes normally fall in the range that can be represented in seven bits. The equation processor (eqn) depends on this assumption. Characters are read into an array of signed 8-bit integers (the default of signed vs. unsigned characters in C varies from compiler to compiler). These characters are then used to compute a hash function. If an 8-bit character value is read, it will appear as a negative number and result in an erroneous hash value. The index to the hash table will then be out of range. This problem can be easily fixed by using unsigned values for the character buffer. In a more sophisticated language than C, characters and strings would be identified as a specific type not related to integers.

This error does not crash all versions of adb. The consequence of the error depends on where in the address space is accessed by the bad hash value. (This error could be considered a subcase of the pointer/array errors.)

Race Conditions

Unix provides a signal mechanism to allow a program to asynchronously respond to unusual events. These events include keyboard-selected functions to kill the program (usually control-C), kill the program with a core dump (usually control-\), and suspend the program (usually control-Z). There are some programs that do not want to allow themselves to be interrupted or suspended; they want to process these control characters directly, perhaps taking some intermediate action before terminating or suspending themselves. Programs that make use of the cursor motions features of a terminal are examples of programs that directly process these special characters. When these programs start executing, they place the terminal device in a state that overrides processing of the special characters. When these programs exit, it is important that they restore the device to its original state.

So, when a program, such as emacs, receives the suspend character, it appears as an ordinary control-Z character (not triggering the suspend signal). Emacs will, on reading a control-Z, do the following: (1) reset the terminal to its original state (and will now respond to suspend or terminate signals), (2) clean up its internal data structures, and (3) generate a suspend signal to let the kernel actually stop the program.

If a control-\ character is received on input between steps (1) and (3), then the program will terminate, generating a core dump. This race condition is inherent in the Unix signal mechanism since a process cannot reset the terminal and exit in one atomic operation. Other programs, such as vi and more, are also subject to the same problem. The problem is less likely in these other programs because they do less processing between steps (1) and (3), providing a smaller window of vulnerability.

Undetermined Errors

The last two columns of Table IV list the programs where the source code was currently not available to us or where we have not yet determined the cause of the crash.

Conclusions

This project started as a simple experiment to try to better understand an observed phenomenon—that of programs crashing when we used a noisy dial-up line. As a result of testing a comprehensive list of utility programs on several versions of Unix, it appears that this is not an isolated problem. We offer two tangible products as a result of this project. First, we provide a list of bug reports to fix the utilities that we were able to crash. This should qualitatively improve the reliability of Unix utilities. Second, we provide a simple-to-use, yet surprisingly effective test method (and tools).

We do not claim that our tests are exhaustive; formal verification is

ability to discover fatal program bugs is impressive.”

required to make such strong claims. We cannot even estimate how many bugs remain to be found in a given program. But our simple testing technique has discovered a wealth of errors and is likely to be more commonly used (at least in the near term) than more formal procedures. Our tests appear to discover errors that are not easily found by traditional testing practices. This conclusion is based on the results from testing AIX 1.1 Unix.

Comments on the Results

Our examination of the results of the tests have exposed several common mistakes made by programmers. Most of these mistakes involve areas already known to experienced programmers, but an occasional reminder is sometimes helpful. From our inspection of the errors found, we suggest the following guidelines:

(1) Check all array references for valid bounds. This is an argument for using range checking full-time. Even (especially!) pointer-based array references in C should be checked. This spoils the terse and elegant style often used by experienced C programmers, but correct programs are more elegant than incorrect ones.
(2) Be sure that all input fields are bounded—this is just an extension of guideline (1). In Unix, using "%s" without a length specification in an input format is a bad idea.
(3) Check all system call return values; do this checking even when an error result is unlikely and the response to an error result is awkward.
(4) Check pointer values often before using them. If all the paths to a reference are not obvious, an extra sanity check can help catch unexpected problems.
(5) Judiciously extend trust to others; not all programmers exercise the same standards of carefulness. If using someone else's program is necessary, make sure that the data its fed has been checked. This is sometimes called "defensive programming."
(6) In redefining something to look too much like something else, a programmer may eventually forget about the redefinition. He or she then becomes subject to problems that occur because of the hidden differences. This may be an argument against excessive use of procedure overloading in languages such as Ada or C++.
(7) Error handlers should handle errors. These routines should be thoroughly tested so that they do not introduce new errors or obfuscate old ones.
(8) Goto statements are generally a bad idea. Dijkstra observed this many years ago [1], but it is difficult to convince some programmers. Our search for the cause of a bad pointer in the prolog interpreter's main loop was complicated by the interesting weaving of control flow caused by the goto statements.

Comments on Lurking Bugs

An interesting question is: why are there so many buggy programs in Unix? This section contains commentary and speculation; it should be considered more editorial than factual. It is our experience that we often encounter bugs in programs, but ignore them; we do so, not because they are not serious (they often cause crashes). There are, however, two reasons for ignoring bugs: First, it is often difficult to isolate exactly what activity caused the program to crash. Second, it's quicker to try a slightly different method to get the current job done than it is to find and report a bug.

As part of an informal survey of the Unix user community in our department (comprising researchers, staff, and students on several hundred Unix workstations), we asked if they had encountered bugs that they had not reported to anyone. We also asked about the severity of the bugs and why they had not reported them. Many users responded to the survey and all (but one) reported finding bugs that they did not report; about two-thirds of these bugs were serious ones. The commentary of the various users speaks for itself. Following are quotes from the responses of several users:

"Because (*name of research tool*) was involved, I figured it is too complicated. Besides, by changing a few parameters, I would get a core image that dbx would **not** crash on, thus preventing me from really having to deal with the problem."

"My experience is that it is largely useless to report bugs unless I can supply algorithms to reproduce them."

"I haven't reported this because recovery from this error is usually fast and easy. . . That is, the time and effort wasted due to a single occurrence of the bug is usually smaller than the time needed to report it."

"I don't generally report problems because I have gotten the impression over the years that unless it's a security hole in mail or something, either no-one will look at it, they will chalk it up to a one time event or user mistake, or it will take forever to fix."

Some users are easy to please. We received one response from our survey that stated:

"I have not encountered any bugs in Unix software."

The number of bugs in Unix might also be explained by its evolution. Unix has suffered from a "features are more important than

An Empirical Study of the Reliability of UNIX Utilities
von Miller, Fredriksen, So

Random Fuzzing

```
o0g Q_b9NY_^"1Gcg{FW^^_<][1DZod Do!
kk#)\x08dy.)q4z5oxX$}t[?w
p:Picfea2m_xb.x2vsGs9&Sac=-x80.
4LbVTri&ez/|O{/xh=59x$Crabax(H0e
r*I/4p.deF+I*#x2f|OI=97}.RP8-ambnr<<0
zf1w.p.azE,qYrefXN8
```

APPENDIX: USER COMMANDS

FUZZ(1)

NAME
fuzz—random character generator

SYNOPSIS
fuzz length [option] . . .

DESCRIPTION
The main purpose of *fuzz* is to test the robustness of system utilities. We use *fuzz* to generate random characters. These are then piped to a system utility (using *pty(1)* is necessary). If the utility crashes, the saved input and output streams can then be analyzed to decide what sorts of input cause problems.

Length is taken to be the length of the output stream, usually in bytes. When –l is selected the length is in number of strings.

The following options can be specified.

–0 Include NULL (ASCII 0) characters
–a Include all ASCII characters except NULL (default)
–d *delay*

Schwierigkeiten mit diesem Ansatz?

An Empirical Study of the Reliability of UNIX Utilities
von Miller, Fredriksen, So

Problem: Nur wenige Inputs interessant

```rust
fn parse_url(url: &str) -> Option<Url> {
    if url.starts_with("https://") || url.starts_with("http://") {
        ...
    }
}
```

Interessante Inputs starten mit `https://` oder `http://`.

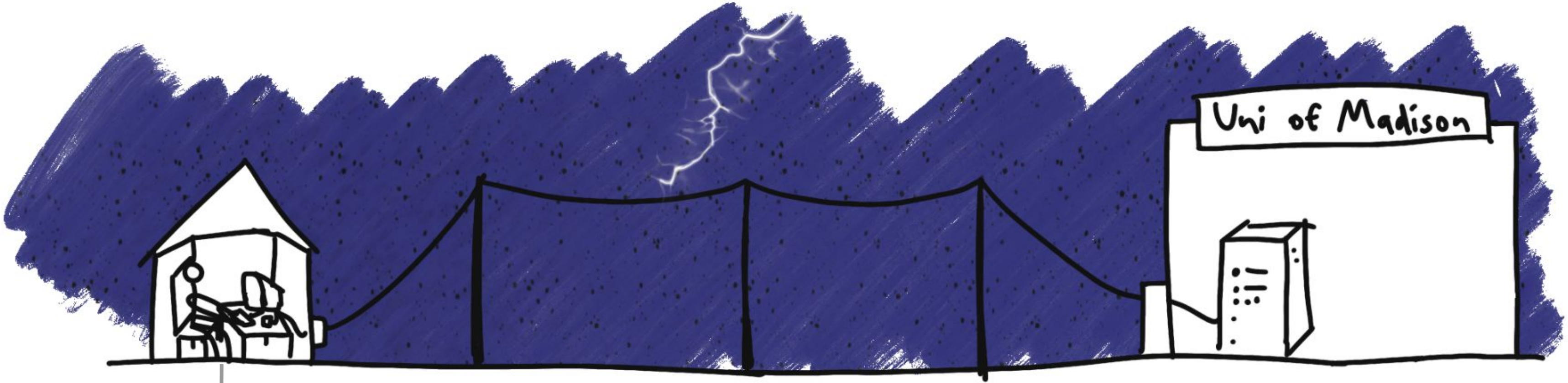
Das trifft nur auf $\left(\frac{1}{256}\right)^8 + \left(\frac{1}{256}\right)^7$ = 0.000 000 000 000 00139... % aller Inputs zu.

Von ~70 000 000 000 000 000 Inputs kommt einer über die erste Zeile hinaus.

Erwartungswert bei 1 000 000 Inputs pro Sekunde: Wir brauchen ~100 000 000 Jahre, bis der restliche Code ausgeführt wird.

Dictionaries & Mutation-Based Fuzzing

Dictionaries & Mutation-Based Fuzzing



Dictionary → Inputs mutieren → Programm

Mutationen

`Hello, world!`

`Blubbelub`

Byte hinzufügen → `Hello, ;world!`

Byte verändern → `Hello, wo%ld!`

Byte entfernen → `Helo, world!`

Zwei Inputs mergen → `Hellobelub`

…

Mutationen

Original-Inputs

```
Hello, world!
Blubbelub
```

Eine Mutation

```
Hello, ;world!
Hello, wo%ld!
Helo, world!
Hellobelub
```

10 Mutationen

```
Ruqbjuold})
gXl(worYhd!
!ellwmoba&u
|Hellhn, )wopl_d!
Hell Adc
```

30 Mutationen

```
sygb)`8*ub
>wMU[*ZH%ljeS`aC
f>f7\:b0&f}uF/
1Ea
Hmki8$oGO|_/\f'
```

Dictionaries & Mutation-Based Fuzzing

```
fn parse_url(url: &str) -> Option<Url> {
    if url.starts_with("https://") || url.starts_with("http://") {
        ...
    }
}
```

Dictionary mit https://google.com, https://hpi.de und http://example.com.

5x mutierte Inputs:

http5s://hp9hY.hde

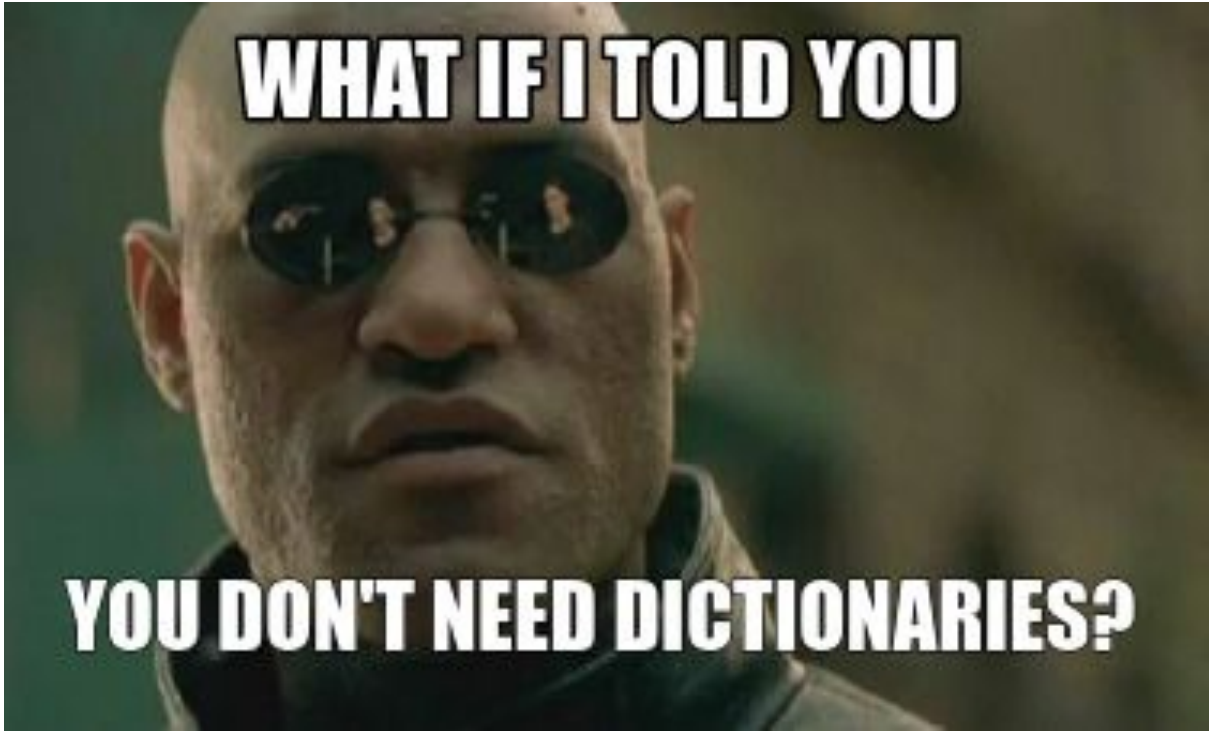
http://ooogew._com

https:/4/goo?gle.c},

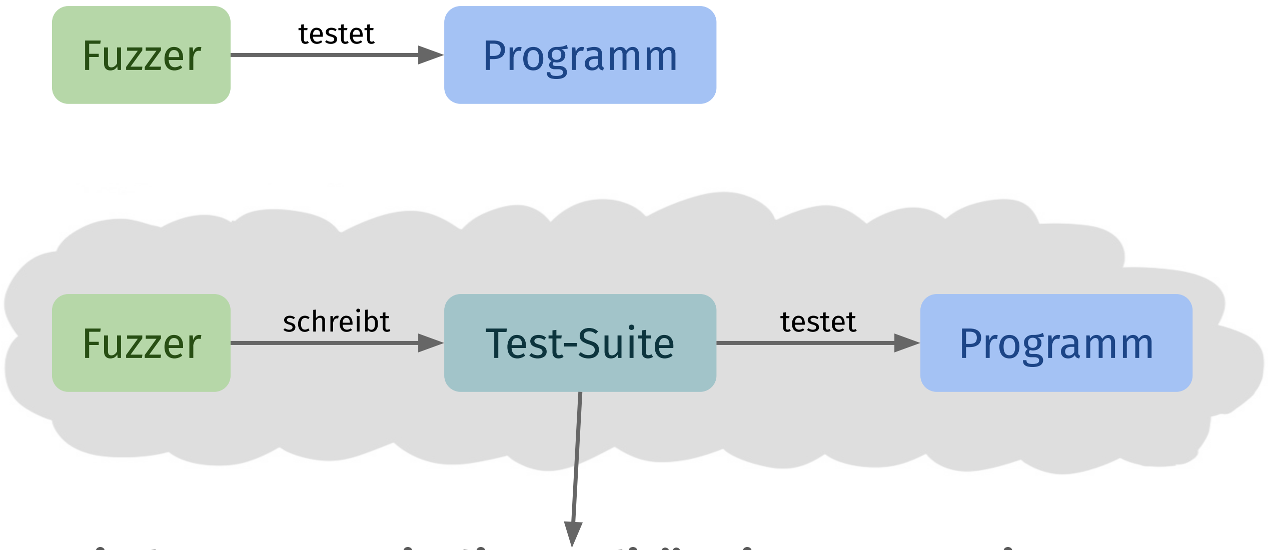
http://Bxajmpe.cm

hts://hpi.EEe

http:/fexai`le5.c;om


WHAT IF I TOLD YOU
YOU DON'T NEED DICTIONARIES?

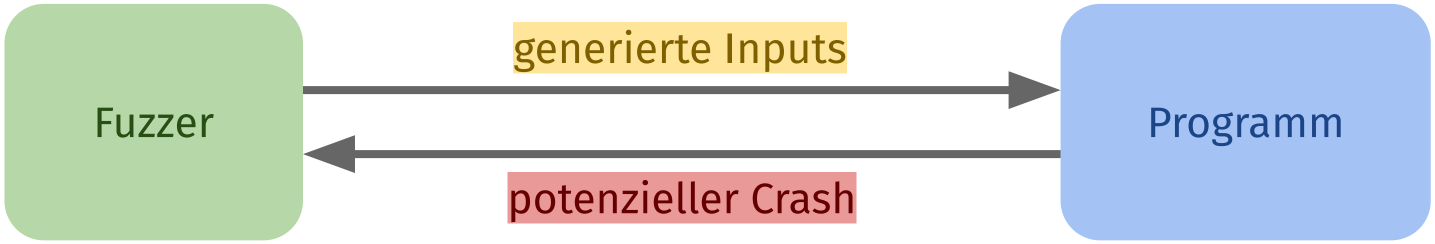
Perspektivenwechsel



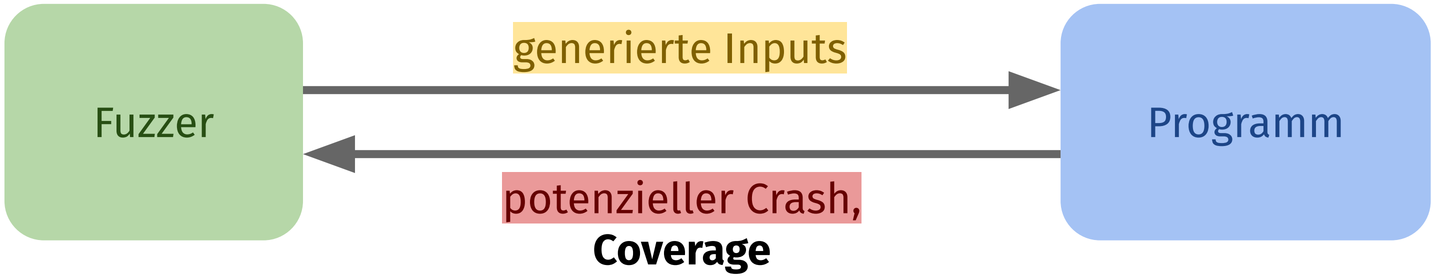
Wie bewerten wir die Qualität einer Test-Suite?

Bekanntes Problem! Lösung: Code-Coverage, Mutation-Tests, …

Schlaueres Fuzzing



Schlaueres Fuzzing



Coverage-Guided Fuzzing

Coverage-Guided Fuzzing

- **Hat kleinen Runtime-Overhead**
  Coverage zu sammeln ist bereits etablierte Praxis. Bei Binaries und vielen Programmiersprachen können wir effizient Coverage sammeln.
- **Lenkt Fuzzer zu interessanten Inputs**
  Coverage reicht häufig aus, um interessante Inputs zu finden.
  Beispiel: Fuzzer für libjpeg generiert unter anderem valide JPEGs.

Granularität von Coverage

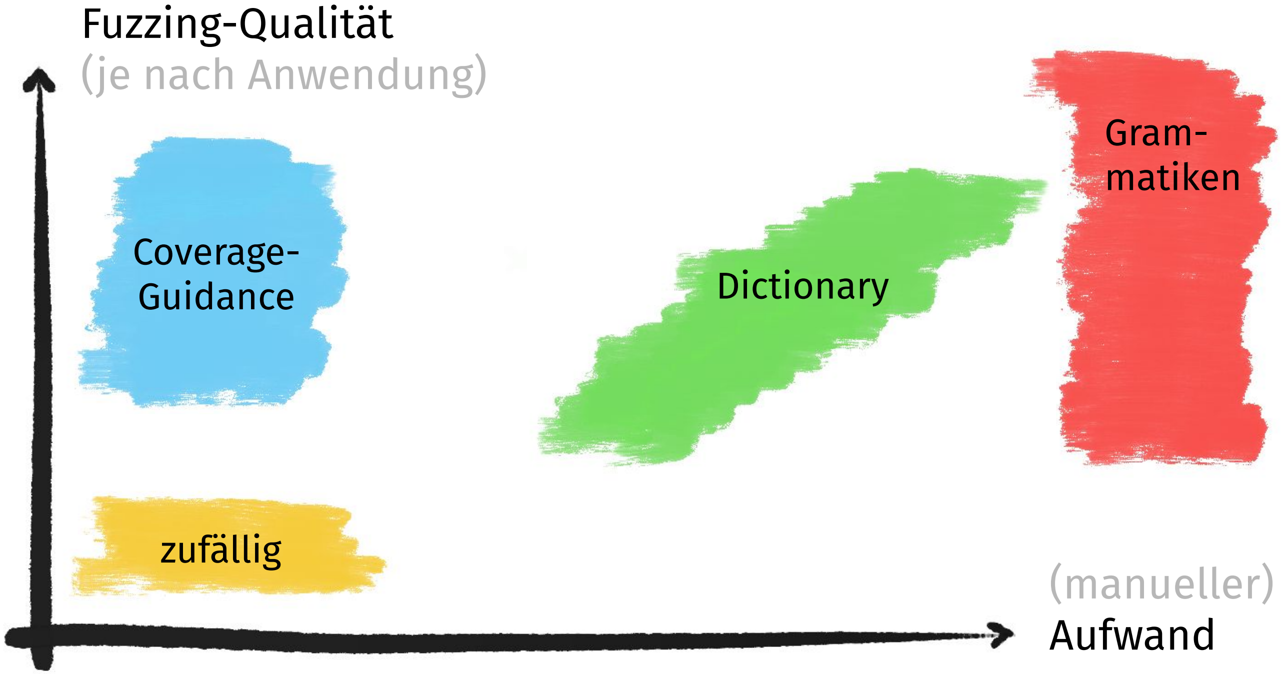
`url.starts_with("https://")`

Je nach Coverage (z.B. Line Coverage) existiert das Wahrscheinlichkeitsproblem immer noch.

Coverage muss auch innerhalb von aufgerufenen Methoden und in Loops mehrfach gesammelt werden.

Klappt, wenn gilt: Mehr von `https://` am Anfang → mehr Coverage

Wie werden Inputs generiert?



Grammar-Based Fuzzing

Grammar-Based Fuzzing

Viele Eingabeformate (alle kontextfreien Sprachen) können als Grammatik in Backus-Naur-Form definiert werden. Es gibt Terminale und Non-Terminale.

```
<start>       ::= <scheme>://<authority><path><query>
<scheme>      ::= http | https | ftp | ftps
<authority>   ::= <host> | <host>:<port>
<host>        ::= www.google.com | hpi.de | example.com
<port>        ::= 80 | 8080 | <nat>
<nat>         ::= <digit> | <digit><digit>
<digit>       ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>        ::=   | / | /<id>
<id>          ::= abc | def | x<digit><digit>
<query>       ::=   | ?<params>
<params>      ::= <param> | <param>&<params>
<param>       ::= <id>=<id> | <id>=<nat>
```

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

```
<start>
```

```
<start>       ::= <scheme>://<authority><path><query>
<scheme>      ::= http | https | ftp | ftps
<authority>   ::= <host> | <host>:<port>
<host>        ::= www.google.com | hpi.de | example.com
<port>        ::= 80 | 8080 | <nat>
<nat>         ::= <digit> | <digit><digit>
<digit>       ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>        ::=   | / | /<id>
<id>          ::= abc | def | x<digit><digit>
<query>       ::=   | ?<params>
<params>      ::= <param> | <param>&<params>
<param>       ::= <id>=<id> | <id>=<nat>
```

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

```
<scheme>://<authority><path><query>
```

```
<start>      ::= <scheme>://<authority><path><query>
<scheme>     ::= http | https | ftp | ftps
<authority>  ::= <host> | <host>:<port>
<host>       ::= www.google.com | hpi.de | example.com
<port>       ::= 80 | 8080 | <nat>
<nat>        ::= <digit> | <digit><digit>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>       ::=   | / | /<id>
<id>         ::= abc | def | x<digit><digit>
<query>      ::=   | ?<params>
<params>     ::= <param> | <param>&<params>
<param>      ::= <id>=<id> | <id>=<nat>
```

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

```
http://<authority><path><query>
```

```
<start>      ::= <scheme>://<authority><path><query>
<scheme>     ::= http | https | ftp | ftps
<authority>  ::= <host> | <host>:<port>
<host>       ::= www.google.com | hpi.de | example.com
<port>       ::= 80 | 8080 | <nat>
<nat>        ::= <digit> | <digit><digit>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>       ::=   | / | /<id>
<id>         ::= abc | def | x<digit><digit>
<query>      ::=   | ?<params>
<params>     ::= <param> | <param>&<params>
<param>      ::= <id>=<id> | <id>=<nat>
```

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

```
http://<host>:<port><path><query>
```

```
<start>      ::= <scheme>://<authority><path><query>
<scheme>     ::= http | https | ftp | ftps
<authority>  ::= <host> | <host>:<port>
<host>       ::= www.google.com | hpi.de | example.com
<port>       ::= 80 | 8080 | <nat>
<nat>        ::= <digit> | <digit><digit>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>       ::=   | / | /<id>
<id>         ::= abc | def | x<digit><digit>
<query>      ::=   | ?<params>
<params>     ::= <param> | <param>&<params>
<param>      ::= <id>=<id> | <id>=<nat>
```

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

```
http://hpi.de:<port><path><query>
```

```
<start>      ::= <scheme>://<authority><path><query>
<scheme>     ::= http | https | ftp | ftps
<authority>  ::= <host> | <host>:<port>
<host>       ::= www.google.com | hpi.de | example.com
<port>       ::= 80 | 8080 | <nat>
<nat>        ::= <digit> | <digit><digit>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>       ::=  | / | /<id>
<id>         ::= abc | def | x<digit><digit>
<query>      ::=  | ?<params>
<params>     ::= <param> | <param>&<params>
<param>      ::= <id>=<id> | <id>=<nat>
```

String-Matching mit DFAs, reg. Ausdrücke, DFAs vs. NFAs
aus den TI-2-Folien
Generating test programs from syntax von Burkhardt

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

```
http://hpi.de:80<path><query>
```

```
<start>      ::= <scheme>://<authority><path><query>
<scheme>     ::= http | https | ftp | ftps
<authority>  ::= <host> | <host>:<port>
<host>       ::= www.google.com | hpi.de | example.com
<port>       ::= 80 | 8080 | <nat>
<nat>        ::= <digit> | <digit><digit>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>       ::=  | / | /<id>
<id>         ::= abc | def | x<digit><digit>
<query>      ::=  | ?<params>
<params>     ::= <param> | <param>&<params>
<param>      ::= <id>=<id> | <id>=<nat>
```

String-Matching mit DFAs, reg. Ausdrücke, DFAs vs. NFAs
aus den TI-2-Folien
Generating test programs from syntax von Burkhardt

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

```
http://hpi.de:80/<id><query>
```

```
<start>      ::= <scheme>://<authority><path><query>
<scheme>     ::= http | https | ftp | ftps
<authority>  ::= <host> | <host>:<port>
<host>       ::= www.google.com | hpi.de | example.com
<port>       ::= 80 | 8080 | <nat>
<nat>        ::= <digit> | <digit><digit>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>       ::=   | / | /<id>
<id>         ::= abc | def | x<digit><digit>
<query>      ::=   | ?<params>
<params>     ::= <param> | <param>&<params>
<param>      ::= <id>=<id> | <id>=<nat>
```

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

```
http://hpi.de:80/abc<query>
```

```
<start>       ::= <scheme>://<authority><path><query>
<scheme>      ::= http | https | ftp | ftps
<authority>   ::= <host> | <host>:<port>
<host>        ::= www.google.com | hpi.de | example.com
<port>        ::= 80 | 8080 | <nat>
<nat>         ::= <digit> | <digit><digit>
<digit>       ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>        ::=   | / | /<id>
<id>          ::= abc | def | x<digit><digit>
<query>       ::=   | ?<params>
<params>      ::= <param> | <param>&<params>
<param>       ::= <id>=<id> | <id>=<nat>
```

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

`http://hpi.de:80/abc`

```
<start>      ::= <scheme>://<authority><path><query>
<scheme>     ::= http | https | ftp | ftps
<authority>  ::= <host> | <host>:<port>
<host>       ::= www.google.com | hpi.de | example.com
<port>       ::= 80 | 8080 | <nat>
<nat>        ::= <digit> | <digit><digit>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<path>       ::=  | / | /<id>
<id>         ::= abc | def | x<digit><digit>
<query>      ::=  | ?<params>
<params>     ::= <param> | <param>&<params>
<param>      ::= <id>=<id> | <id>=<nat>
```

String-Matching mit DFAs, reg. Ausdrücke, DFAs vs. NFAs
aus den TI-2-Folien
Generating test programs from syntax von Burkhardt

Grammar-Based Fuzzing

Mit der Grammatik lassen sich Inputs generieren:

`http://hpi.de:80/abc`

`https://example.com/?def=2`

`ftp://www.google.com:8080/def?x20=abc&abc=24`

`http://example.com/x48`

`https://www.google.com:93`

`ftps://hpi.de:2/?x97=5&x82=9&abc=x80`

Grammar-Based Fuzzing

Unendliche Expansionen durch Rekursionen können vermieden werden, indem ab irgendeinem Zeitpunkt immer die kleinste Expansion gewählt wird.

Coverage der Grammatik ermöglicht es, Inputs zu erstellen, die möglichst viele Fälle (große Teile der Grammatik) abdecken.

Wahrscheinlichkeiten für probabilistische Grammatiken können aus Dictionaries gelernt werden. Damit lassen sich natürlich aussehende sowie sehr unübliche Inputs erstellen.

Keywords können gelernt werden indem die Coverage mehrerer Ausführungen verglichen wird.

Two fast tree-creation algorithms for genetic programming von Luke

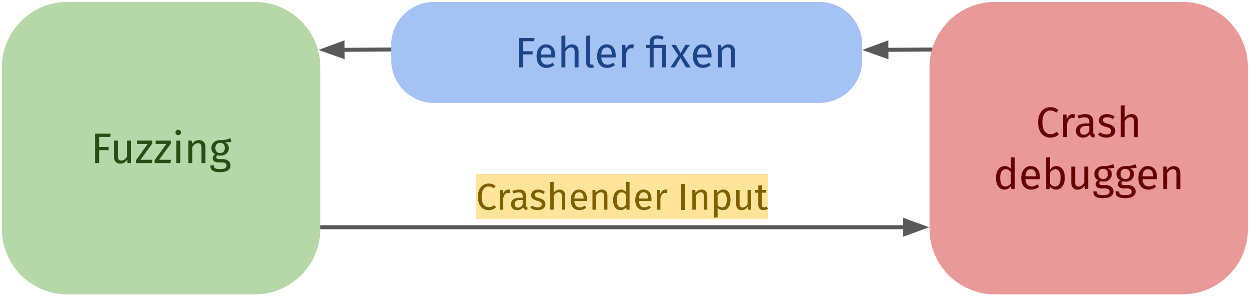
A sentence generator for testing parsers von Purdom

Inputs from Hell: Generating Uncommon Inputs from Common Samples von Pavese, Soremekun, Havrikov, Grunske, Zeller

afl-fuzz: making up grammar with a dictionary in hand von lcamtuf

Input-Vereinfachung

The Big Picture



Kleiner Input:

- Besonderheiten sind offensichtlich
- Ausführung ist kurz
- Duplikate sind filterbar

Delta Debugging

```
http://ooogew._com        ...        http://_
         ogew._com
http://oo
    //ooogew._com
http:    ogew._com
http://oo     _com
```

Basically Binary Search. Wenn keine Hälfte mehr weggelassen werden kann, werden kleinere Teile des Inputs entfernt, bis hin zu einzelnen Zeichen.

Simplifying and isolating failure-inducing input
von Hildebrandt, Zeller

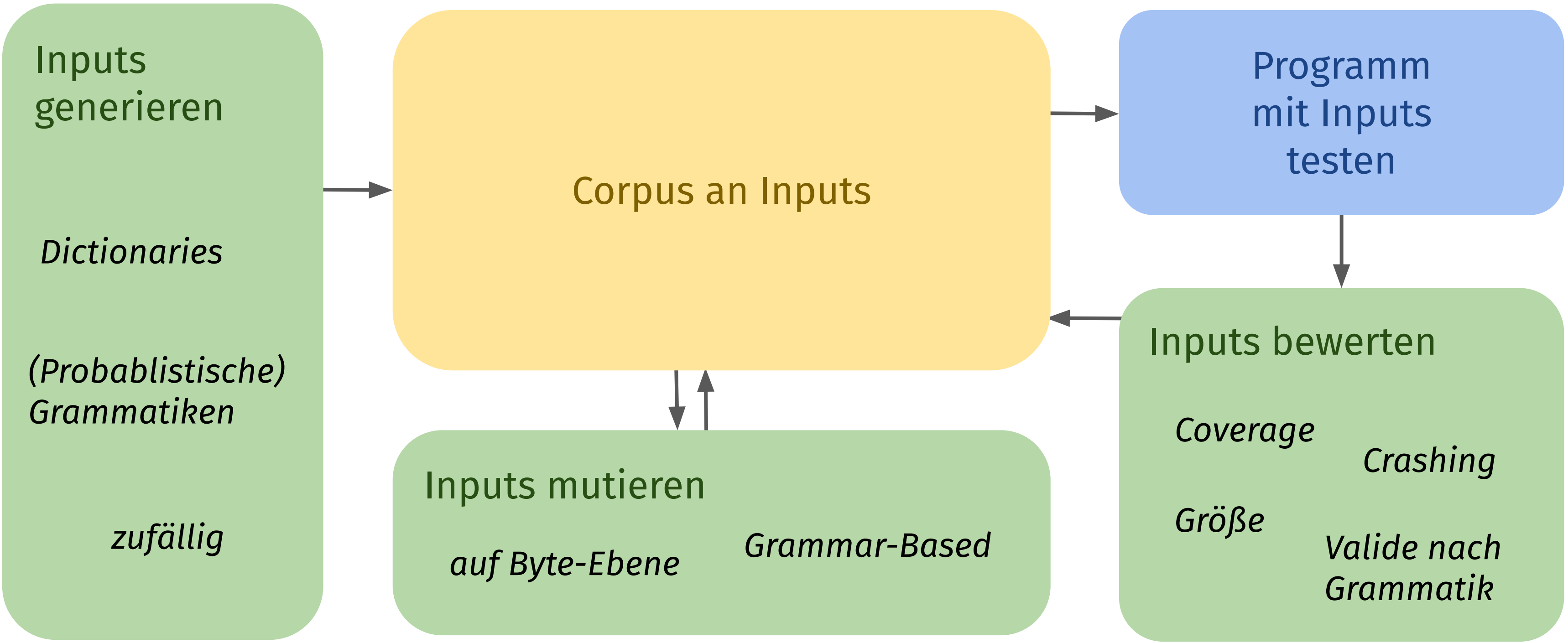
Technical Considerations

Gutes Fuzzing sollte das gesamte Programmverhalten abdecken.
Dazu brauchen wir viele interessante Inputs.

schnell Inputs generieren

schlau Inputs generieren

Architektur



Was sind eigentlich Inputs?

Standardeingabe
Dateisystem
Umgebungsvariablen
Netzwerkzugriff
Dauer der Ausführung + Timer
Scheduling-Reihenfolge von Threads
...

Programm

Wie kriegen wir diese Inputs unter Kontrolle?

Emulation-Based Fuzzing

“There is no problem in computer science that can’t be solved using another level of indirection.” — David Wheeler

Eigene VM hat volle Einsicht in und Kontrolle über die Ausführung:

- Welche Instruktionen wurden ausgeführt?
- Welche Speicherzugriffe finden statt?
- Snapshotting von Programmen
- …

BaseSAFE: Baseband SAnitized Fuzzing through Emulation von Maier, Park, Seidel

NYX: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types von Abbasi, Aschermann, Holz, Schumilo, Wörner

Blackbox- vs. Whitebox-Fuzzing

generiert Inputs aufgrund einer Spezifikation

Implementation hat oft mehr Sonderfälle
→ testet nicht das ganze Verhalten

generiert Inputs aufgrund einer Implementierung

covert Struktur der Implementierung
→ testet meist Spezifikation mit
→ kann fehlende Funktionalität nicht finden

Funktioniert bei unvollständiger Spezifikation
→ deckt Edge Cases auf

Greybox-Fuzzing: Begriff für Fuzzer, die simples Coverage-Feedback von Programmen bekommen, aber keine ausgefeilte Programmanalyse oder Constraint Solver nutzen.

Fuzzing in der echten Welt

Wann fuzzen?

Code ist …

- anfällig für Sicherheitslücken
- sicherheitskritisch

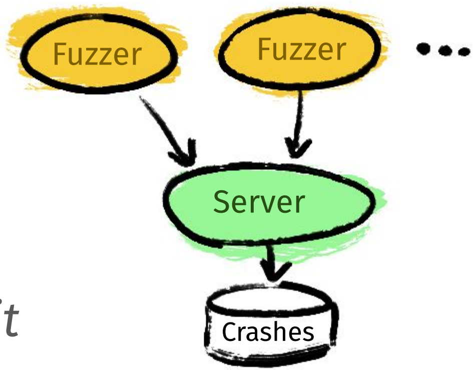
Wann aufhören?

Wahrscheinlichkeit, dass Coverage noch weiter erhöht wird: $\sim \frac{\text{nur einmal gecoverter Branch}}{\text{alle Branches}}$

Wie fuzzen?

Fuzzing in der Cloud

z.B. ClusterFuzz von Google mit 6000 Chrome-Instanzen 24/7



Automated Penetration Testing with White-Box Fuzzing von Neystadt
Fuzzing for Security von Arya, Neckar (Chrome Security Team)
Estimating the discovery probability von Good

American Fuzzy Lop

- Coverage-Guided Fuzzing von beliebigen Binaries
- Jump/Branch-Instruktionen werden ersetzt durch Zwischenjumps zu anderen Stellen, die dann Coverage sammeln

Fand kritische Sicherheitslücken in Mozilla Firefox, Apple Safari, iOS kernel, sqlite, Linux ext4, Tor, PHP, OpenSSL, OpenSSH, LibreOffice, libpng, curl, GPG, OpenCV, zstd, lz4, MySQL, ...

american fuzzy lop (offizielle Webseite)
Technical "whitepaper" for afl-fuzz on GitHub

American Fuzzy Lop: Erweiterungen

- **AFL:** Nicht mehr maintained, aber Grundlage für viele andere Fuzzing-Ansätze und Paper
- **AFLFast:** AFL mit Fokus auf selten genommene Coverage-Pfade, findet dadurch Crashes um eine Größenordnung schneller als AFL
- **AFLGo:** Leitet Input-Generierung zu definierten Stellen im Code
- **AFL++:** Maintainte Version von AFL, mehr Hardwarebeschleunigung, Strategien von AFLFast, mehr Mutationen

Coverage-based Greybox Fuzzing as Markov Chain von Böhme, Pham, Roychoudhury
Directed Greybox Fuzzing von Böhme, Nguyen, Pham, Roychoudhury
American Fuzzy Lop plus plus (AFL++) on GitHub

LangFuzz

- Grammar-Based Fuzzing für Programmiersprachen
- Nutzt Dictionary mit Inputs aus früheren (gefixten) CVEs
  → einige Fixes waren nur oberflächlich
  → ähnliche Inputs können den gleichen Fehler ausnutzen

Entdeckte im Mozilla-JavaScript-Interpreter in drei Monaten 105 neue Sicherheitslücken (>50 000$ in Bug Bounties).

Wie geht’s weiter mit Fuzzing?

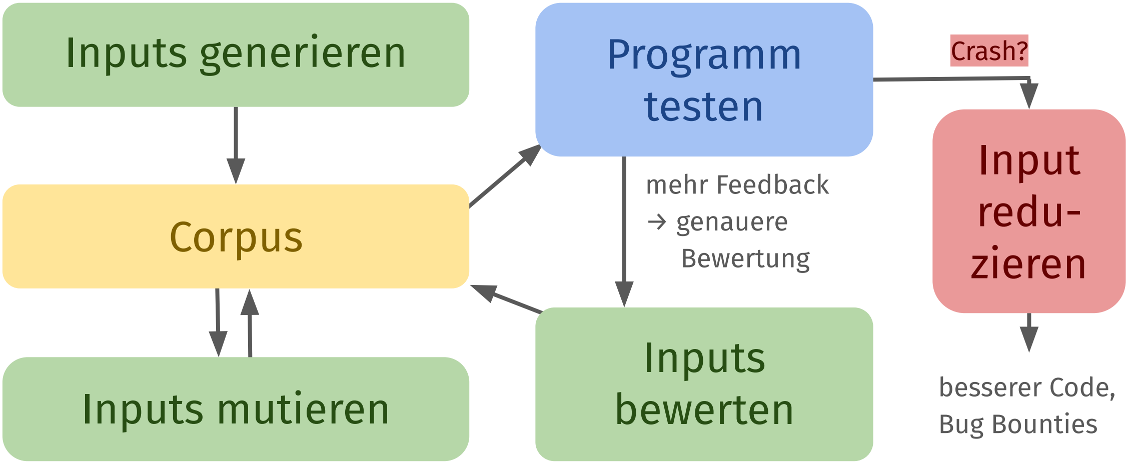
- Closed-Source-Fuzzer sind denen aus Papern meist einige Jahre voraus, denn Bug Bounties abstauben lohnt sich
- Fuzzing findet nur Crashes
  → *Testen von komplexeren Invarianten ist Property-Based Testing*
- Constraint Solving kann noch bessere Inputs generieren als Grammatiken
  → *kann mit Symbolic Execution und Solvern kombiniert werden*

Paper Review: Nix von gamozolabs
Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution von Pak
Symbolic Execution, Christophs Präsi aus diesem Seminar

Zusammenfuzzung

Fuzzing: Automatisiertes Testen von Code mit vielen Eingaben, häufig eingesetzt im Security-Bereich bei Trust Boundaries

Random, Dictionaries, Mutation-Based, Coverage-Guided, Grammar-Based



Implementation

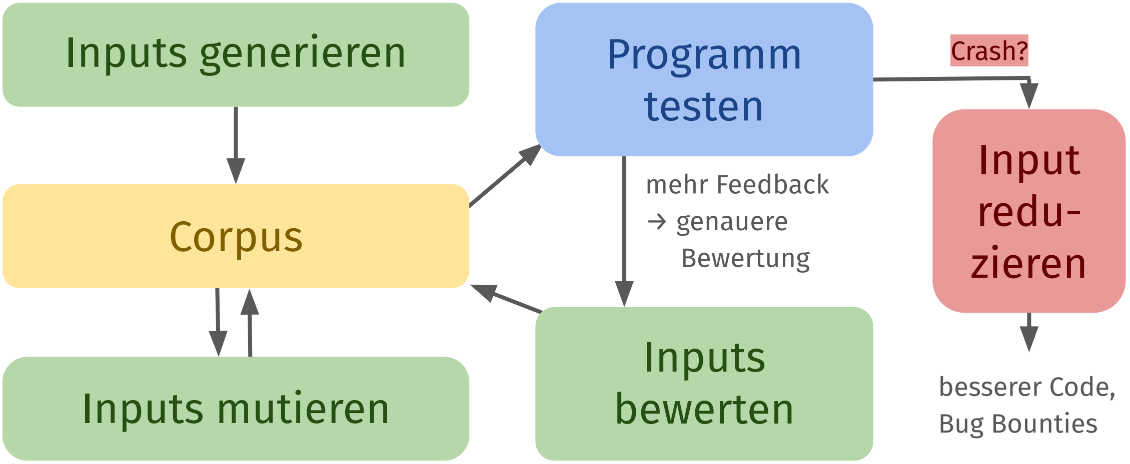
Fokus: Performance

Whitebox vs. Blackbox
→ *Quantität vs. Qualität von Inputs*
→ *Emulation bringt Inputs unter Kontrolle*

Danke fürs Zuhören!

Fuzzing: Automatisiertes Testen von Code mit vielen Eingaben, häufig eingesetzt im Security-Bereich bei Trust Boundaries

Random, Dictionaries, Mutation-Based, Coverage-Guided, Grammar-Based



Implementation

Fokus: Performance

Whitebox vs. Blackbox
→ *Quantität vs. Qualität von Inputs*
→ *Emulation bringt Inputs unter Kontrolle*