

Fuzzing Forth

Apply Fuzz Tests to Forth

EuroForth'22 conference 2022-09

Ulrich Hoffmann



Overview

Fuzzing Forth

- Introduction
- Correctness Notions
- Generators
- Mutators
- Sanitizers
- Tests
- Fuzzing
- Conclusion



Introduction

What fuzzing is all about?

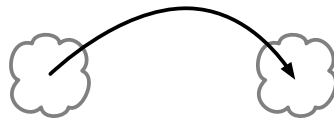
- We assure quality of applications by **testing**
 - Manually, especially for embedded systems → time consuming
 - Automatically, correct functions, regression, TestDrivenDevelopment
 - We mostly test the good cases, infrequently behaviour in bad situations
- **Fuzz Tests** or **Fuzzing** tests applications with arbitrary data to see if they break

"Crash often crash early!" - but automated

Correctness Notions

What's in a word?

- When is a word correct?
 - need to describe the behaviour of a word
 - an approach: a word does a state transition
 - from a current state
 - to a next state
 - can be deterministic or non deterministic



States

- States can be complicated
 - not just labels as with finite state machines
 - **Forth System State**: includes the stack and return stack content, all dictionary content, the existing definitions, etc.
 - **Computer State**: contents of files, memory content, etc.
 - **Environment State**: relevant state of external components
- Think of states as huge records or vectors
- A state or sets of states can be described by conditions
"the (set of all) states that satisfy the condition"

State Transitions

- many aspects of a state are not relevant for the transition and stay as they are
- a transition (i.e. the behaviour of a word) can be described by a pre-condition P and a post-condition Q

Stack Comments

- We describe conditions already in Forth with stack comments

$0 < (n \text{ -- flag })$

n : The condition *TOS is a natural number* in the current state

-- : symbolizes the transition

flag: the condition *TOS is with all bits set (true) or all bits reset (false)*, i.e. a *flag* in the next state

- stack diagrams are not sufficient to specify the operation exactly
 $0 >$ has the same stack effect but a different behaviour.

Stack Comments

$0 < (n \text{ -- flag })$

...

flag: the condition *TOS is with all bits set (true) or all bits reset (false)*, i.e. a *flag* in the next state

- the post-stack-condition is too weak
- stronger post condition for $0 <$:
flag where $TOS' = \text{true}$ if $TOS < 0$, $TOS' = \text{false}$ if $TOS \geq 0$
- appropriate pre- and post-condition can describe the behaviour of a word as precisely as desired, but they may be difficult to specify

Partial Correctness of a word (a transition)

A word is partially correct with respect to conditions P and Q:

- if the current state satisfies the pre-condition P and **if the word terminates** (i.e. does not crash) then the next state satisfies the post-condition Q

if the current state does not satisfy the pre-condition then the next state is undefined (computer scientists model this often with non-terminating programs or arbitrary results).

Total Correctness of a word (a transition)

A word is totally correct with respect to conditions P and Q:

- if the current state satisfies the pre-condition P **then the word terminates** (i.e. does not crash) **and** the next state satisfies the post-condition Q

if the current state does not satisfy the pre-condition then the next state is undefined.

Robustness of a word (a transition)

A word is robust with respect to P and Q

- if it always does a transition to a next state.
- is totally correct with respect to P and Q
- if the current state does not satisfy the pre-condition then *an error is signalled*.
 - *throw an exception*
 - *return an distinct error value*
 - ...

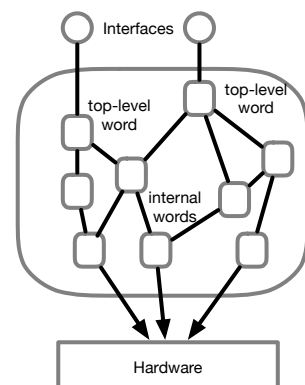
Fuzzing checks if an application i.e. its top-level words are robust

Fuzzing

What the fuzz?

Fuzzing checks if an application i.e. its top-level words are robust

- top level words are accessible for outside
- realize outside interfaces
- **Fuzzing**
 - **invoke a top-level words with arbitrary data**
 - **check if the system crashes**
- Random data → generators and mutators
- best no crashes → sanitizers



Generators

How to create random data?

- We need to generate random (stack) items.
- even distribution but also normal and other distributions
- classical Starting Forth random number generator

```
( Random number generation -- High level )  
  
VARIABLE rnd  HERE rnd !  
: RANDOM  rnd @ 31421 * 6927 + DUP rnd ! ;  
: CHOOSE  ( u1 -- u2 )  RANDOM UM* NIP ;
```

- linear congruence generator not good as factor for normal distribution

[1] <http://www0.cs.ucl.ac.uk/staff/D.Jones/GoodPracticeRNG.pdf>

Generators

How to create random data?

- KISS generators are simple and have better properties
- JKISS32 [1] is based on 32-bit integer arithmetic
 - passes all of the Dieharder tests and the BigCrunch tests

```
\ JKISS32 for 32Bit Systems (algorithm by David Jones)  
  
Variable x 123456789 x !  
Variable y 234567891 y !  
Variable z 345678912 z !  
Variable w 456789123 w !  
Variable c 0 c !  
  
: kiss ( -- x )  
  y @ dup 5 lshift xor  dup 7 rshift xor  dup 22 lshift xor  y !  
  z @ w @ + c @ +  w @ z !  dup 0< 1 and 0= 0= 1 and c !  2147483647 and w !  
  x @ 1411392427 + x !  
  x @ y @ + w @ + ;
```

Generators

How to create random data?

- create normal distribution from even distribution
- different algorithms such as Marsaglia polar method or Box Muller Transform require floating point
- Rule of 12 (sum and average) is simple and works on integers
 - but only valid if random delivers independent random values
- linear congruence generators (such as random earlier) do not have this property, KISS does.

```
: CHOOSE  ( u1 -- u2 )  KISS UM* NIP ;  
  
: NORMAL  ( u1 -- u2 )  
  0 12 0 DO  OVER CHOOSE +  LOOP 12 / SWAP DROP ;
```

Generators

How to create random data?

- from CHOOSE and NORMAL we can build generators for typical Forth data
- cell data on the stack

```
: cgen ( -- b ) 256 choose ;  
: ngen ( -- n ) -1 choose ;  
: +ngen ( -- n ) -1 1 rshift choose ;  
: ugen ( -- u ) -1 choose ;
```

or also with specific distribution

- strings of given exact or maximal length

```
: 'x'gen ( -- c ) 128 bl - choose bl + ;  
  
: $=gen ( u1 -- c-addr u2 ) \ string is exactly u1 characters. allocates, must be freed after use  
dup allocate throw swap 2dup bounds ?DO 'x'generate I c! LOOP ;  
  
: $gen ( u1 -- c-addr u2 ) \ string is shorter than u1 characters. allocates, must be freed after use  
choose $=gen ;
```

Generators

How to create random data?

- Generate random composed data structures such as
 - structs and
 - arrays or
 - linked lists, etc.

by defining appropriate (recursive) generators.

```
: person-gen ( -- addr ) ... ;
```

Mutators

How to change existing data?

- Mutators are similar to generators but modify existing data
- cell data on the stack
 - **cmut** (char1 rate -- char2)
 - **nmult** (n1 rate -- n2)
 - **+nmult** (+n1 rate -- +n2)
 - **umult** (u rate -- u2)
- or strings
 - **\$=mult** (c-addr1 u rate -- c-addr2 u) just changes the characters
 - **\$mult** (c-addr1 u1 rate -- c-addr2 u2) changes length and character

Mutators

How to change existing data?

- Modify random composed data structures such as
 - structs and
 - arrays or
 - linked lists, etc.

by defining appropriate (recursive) mutators.

```
: person-mut ( addr1 rate -- addr2 ) ... ;
```

Sanitizers

How to detect and signal crashes?

- If application crash, then it is hard to monitor them.
 - Turn crashes into reported errors
- **Sanitizers** check if inputs are valid
- Memory sanitizers - detect illegal memory access , throw for memory faults
- Stack Sanitizers - detect stack over and underflow
- Control flow sanitizers - check for valid return addresses on EXIT
- ...

Sanitizers

How to detect and signal crashes?

- Memory sanitizers - detect illegal memory access , throw for memory faults
 - @ ! c@ c! with valid memory test, throw memory-fault on invalid address
 - linked list of valid regions that are checked on access
 - bit-field of valid memory words or bytes

```
: ?valid ( addr -- addr ) dup valid? 0= #memory-fault and throw ;  
: @ ( addr -- x ) ?valid @ ;  
: ! ( x addr -- ) ?valid ! ;
```

Sanitizers

How to detect and signal crashes?

- Stack Sanitizers - detect stack underflow

```
: arguments ( i*x u -- i*x )
  >r depth r> u< #stack-underflow and throw ;

1 2 3    3 arguments . . . ( 3 2 1 )
```

- and overflows
 - much harder to detect, every push must check
 - might need hardware support

Sanitizers

How to detect and signal crashes?

- Control flow sanitizers - check for valid return addresses on EXIT

```
: exit ( i*x u -- i*x ) rdrop
  r@ invalid-return-address? #return-stack-imbalance and throw
;
```

- others:
 - check exception stack
 - ...

[1] <ftp://ftp.taygeta.com/pub/Forth/Applications/ANS/tester.fr>

Tests

How do we test?

- A popular framework for tests of forth words are testers derived from John Hayes ANS Forth tester [1]

```
{ 3 4 + -> 7 }
```

- Tests + on a single value.
- We can elaborate test suites from this, as Gerry Jackson does for Forth200x compliance.

Fuzzing

What the fuzz?

- In order to fuzz our applications
 - run the Hayes style unit tests - fix all bugs
 - define the top-level words (TLW) using sanitizers
 - run the Hayes style unit tests - no issue expected
 - run fuzz tests in this style

```
\ assuming TLW ( c-addr1 u1 c-addr2 u2 -- c-addr3 u3)
many times DO
  { 100 $gen S" secret" 75 $mut TLW clearstack -> }
LOOP
```

Fuzzing Forth

Apply Fuzz Tests to Forth

Conclusion

What's below the bottom line?

- Correctness Notions - partial, total correctness, robustness
- Generators - KISS generator, rule of 12
- Mutators - change existing data
- Sanitizers - make crashes into reported errors
- Tests - Hayes style testing
- Fuzzing - stressing top level words (interfaces)

Questions?

