research

Automated program analysis

Program fuzzing and symbolic execution

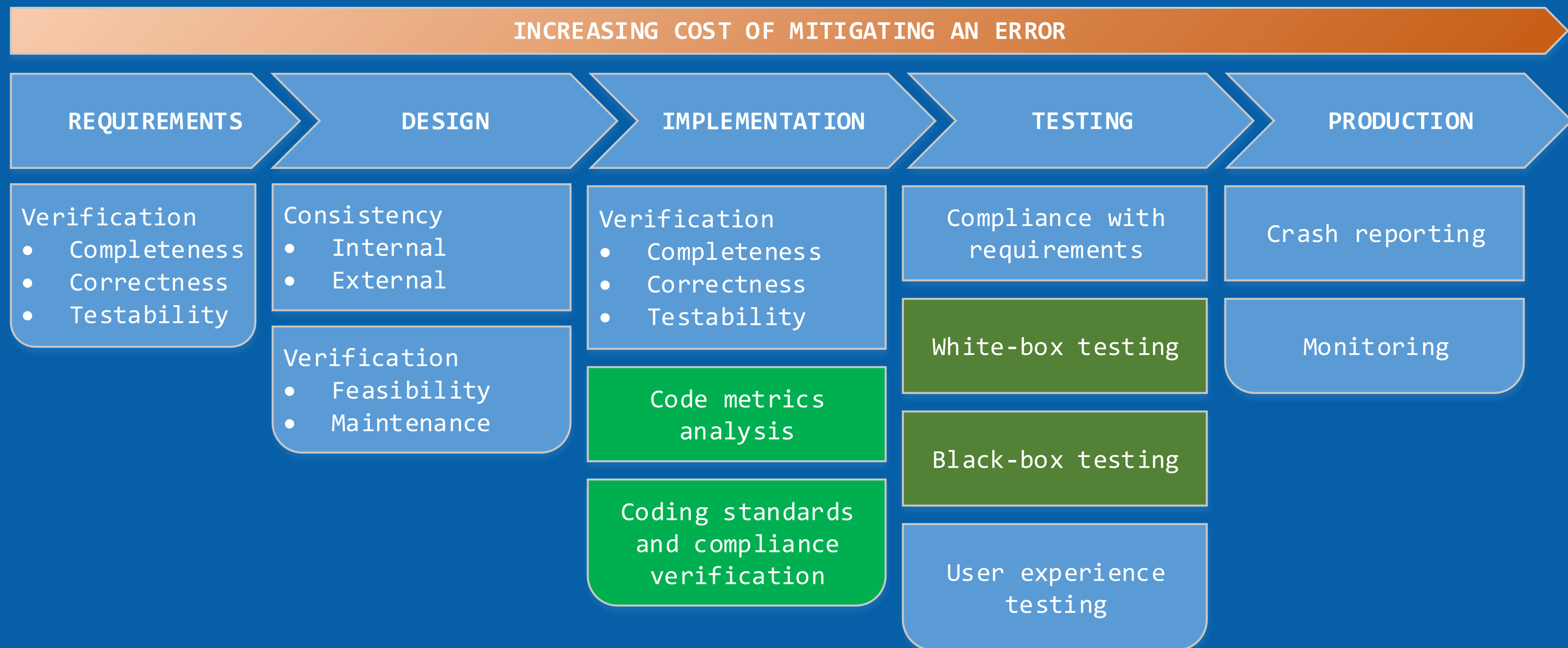
Alexandra Săndulescu
Teodor Stoenescu

Bitdefender®

Outline

- Motivation
- Fuzzing
- Symbolic execution
- Hybrid approaches

Motivation



Defensive programming

- Treat all data as malicious
- Use design-by-contract programming by enforcing
  - Preconditions
  - Postconditions
  - Invariants
- Fail often and loud
  - Trigger exceptions instead of returning error values
  - Use asserts
- Avoid language undefined behaviors
  - (In C/C++) buffer overruns
  - Integer overflows

Motivation redux

- Software complexity and size grow over time
- As shallow bugs are found, deeper bugs remain harder to find

Manual testing is rapidly becoming unfeasible!

Motivation redux (II)

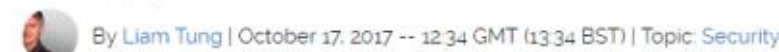
Is security on the verge of a fuzzing breakthrough?

18 OCT 2017

Opinion

Linus Torvalds says targeted fuzzing is improving Linux security

Linux 4.14 release candidate five is out. "Go out and test," says Linus Torvalds.



Recommended Content:
White Papers: Vulnerability management for modern IT. Start your 60-day free trial today.
The IT landscape is changing. Vulnerability management needs to change too. Tenable.ioVulnerability Management provides the most accurate information about all your assets and vulnerabilities in ever-changing environments. From data center to cloud...
Get Started

RECOMMENDED FOR YOU
Maximising Security and Minimising Overheads with Kaspersky Endpoint Security for Cloud

Microsoft Security Risk Detection

Sign up now for the Windows or new Linux preview.

SIGN UP FOR THE PREVIEW >

What is Microsoft Security Risk Detection?

Security Risk Detection is Microsoft's unique fuzz testing service for finding security critical bugs in software. Security Risk Detection helps customers quickly adopt practices and technology battle-tested over the last 15 years at Microsoft.

READ SUCCESS STORIES >

"Million dollar" bugs
Security Risk Detection uses "Whitebox Fuzzing" technology which discovered 1/3rd of the "million dollar" security bugs during Windows 7 development.

Battle tested tech
The same state-of-the-art tools and practices honed at Microsoft for the last decade and instrumental in hardening Windows and Office — with the results to prove it.

Scalable fuzz lab in the cloud
One click scalable, automated, Intelligent Security testing lab in the cloud.

Cross-platform support
Linux Fuzzing is now available. So, whether you're building or deploying software for Windows or Linux or both, you can utilize our Service.

Success stories

The bug-o-rama trophy case

Yeah, it finds bugs. I am focusing chiefly on development and have not been running the fuzzer at a scale, but here are some of the notable vulnerabilities and other uniquely interesting bugs that are attributable to AFL (in large part thanks to the work done by other users):

IJG jpeg	libjpeg-turbo	libpng
libtiff	mozjpeg	PHP
Mozilla Firefox	Internet Explorer	Apple Safari
Adobe Flash / PCRE	sqlite	OpenSSL
LibreOffice	poppler	freetype
GnuTLS	GnuPG	OpenSSH
PuTTY	ntpd	nginx
bash (post-Shellshock)	tcpdump	JavaScriptCore
pdfium	ffmpeg	libmatroska
libarchive	wireshark	ImageMagick
BIND	QEMU	lcms
Oracle BerkeleyDB	Android / libstagefright	iOS / ImageIO
FLAC audio library	libsndfile	less / lesspipe
strings (+ related tools)	file	dpkg
rcs	systemd-resolved	libyaml
Info-Zip unzip	libtasn1	OpenBSD pfctl
NetBSD bpf	man & mandoc	IDA Pro [reported by authors]
clamav	libxml2	glibc
clang / llvm	nasm	ctags
mutt	procmail	fontconfig
pdksh	Qt	wavpack
redis / lua-cmsgpack	taglib	privoxy
perl	libxmp	radare2
SleuthKit	fwknop [reported by author]	X.Org
exifprobe	jhead [?]	capnproto
Xerces-C	metacam	djvulibre
exiv	Linux btrfs	Knot DNS
curl	wpa_supplicant	libde265 [reported by author]
dnsmasq	libbpg (1)	lame
libwmf	uudecode	MuPDF
imlib2	libraw	libbson
libsass	yara	W3C tidy-html5
VLC	FreeBSD syscons	John the Ripper
screen	tmux	mosh
UPX	indent	openjpeg
MMIX	OpenMPT	rxvt
dhcpcd	Mozilla NSS	Nettle
mbed TLS	Linux netlink	Linux ext4
Linux xfs	botan	expat
Adobe Reader	libav	libical
OpenBSD kernel	collectd	libidn
MatrixSSL	jasper	MaraDNS
w3m	Xen	OpenH232
irssi	cmark	OpenCV
Malheur	gstreamer	Tor
gdk-pixbuf	audiofile	zstd
lz4	stb	cJSON
libpcre	MySQL	gnulib
openexr	libmad	ettercap
lrzip	freetds	Asterisk
ytnef	raptor	mpg123
Apache httpd	exempi	libgmime
pev	Linux mem mgmt	sleuthkit
Mongoose OS	iOS kernel	

Highlights (AFL only)

- Firefox (4)
- Internet Explorer (4)
- Adobe Flash (7)
- OpenSSL (7)
- Tcpdump (9)
- OpenSSH (5)
- Clang/LLVM (8)

Fuzzing

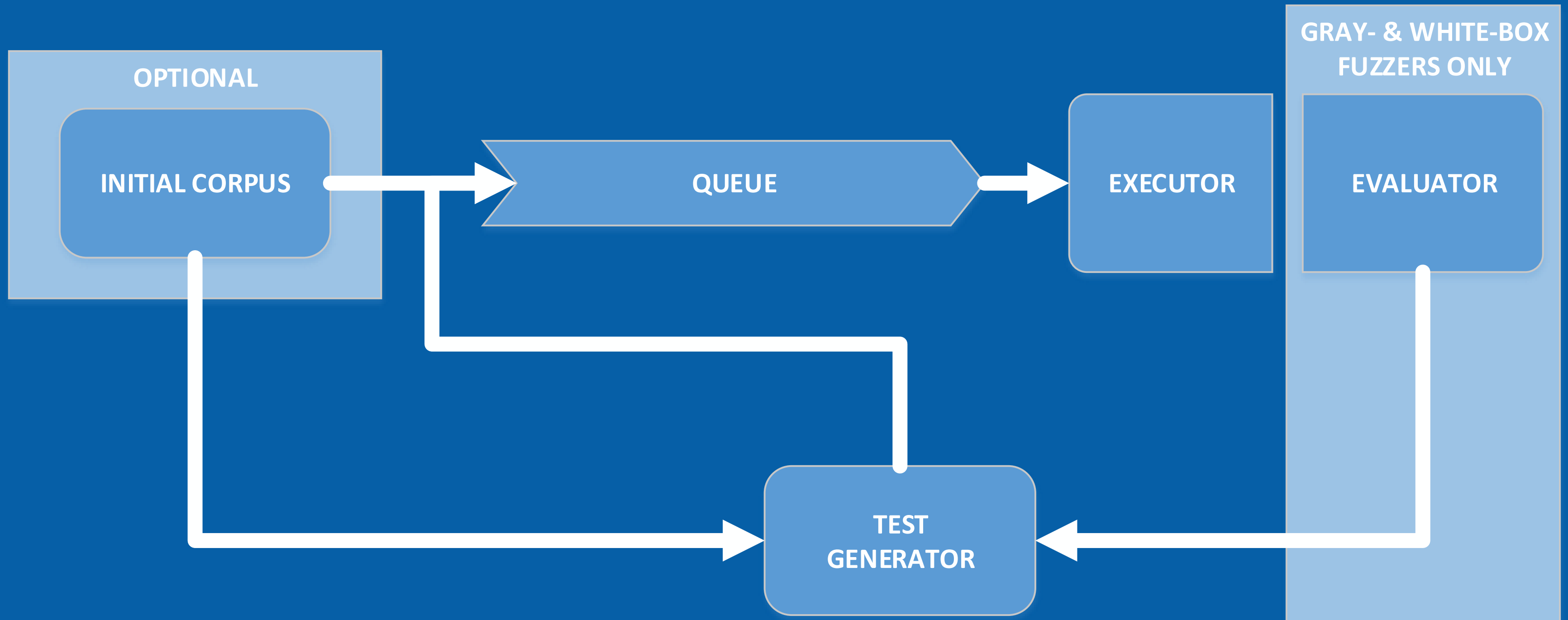
"Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program." – Wikipedia

1. Start with a computer program and (optionally) a test corpus
2. Feed the inputs in the test corpus to the program
3. (Optionally) monitor program behavior
4. Generate new tests taking into account the results @ step 3
5. Repeat steps 2-4 until an interesting behavior is found

Fuzzer taxonomy

- Depending on the fuzzed target
  - Source code fuzzers vs. binary fuzzers
- Depending on the awareness of the input structure
  - Dumb fuzzers vs. smart fuzzers
- Depending on the awareness of the program structure
  - Black-box fuzzers vs. Gray-box fuzzers vs. White-box fuzzers
- Depending on way new tests are generated
  - Generation-based fuzzers vs. Mutation-based fuzzers

Main fuzzer architecture



Source code fuzzers vs. binary fuzzers

Source code available

- Necessary instrumentation can be inserted at compile time
- The fuzzing target is a function call
  - The input consists of one or more function parameters

Binary only available

- Instructions translated using dynamic binary instrumentation
- The fuzzing target is a function inside a static/dynamic library
  - The input consists of one or more function parameters
- The fuzzing target is a separate executable
  - The input can be a file (including stdin), the command line, etc.

Dumb fuzzers vs. smart fuzzers

- Dumb fuzzers treat inputs as a simple buffer array
- Smart fuzzers know the input structure
  - Can be hinted manually
  - Can be extracted by a compiler pass in case of a source code fuzzer
- Knowing input structure can lead to faster bug discovery
  - By avoiding invalid values (floating point)
  - By testing invalid pointers only once
  - By testing limit cases (0x00000000 and 0xFFFFFFFF for DWORDs)

Black- vs. Gray- vs. White-box fuzzers

Program coverage *is a measure used to describe the degree to which the program is tested. Program coverage is typically measured in basic blocks or state transitions.*

- White-box fuzzers are aware of program structure and systematically increase program coverage
- Gray-box fuzzers use instrumentation in order to increase program coverage
- Black-box fuzzers use random searches for increasing program coverage

Generation-based vs. Mutation-based

- Generation-based fuzzers limit themselves to creating new tests
  - (Optionally) use a dictionary
- Mutation-based fuzzers create new tests based on previous iterations
  - Bit flips
  - Addition of small integers
  - Insertion of interesting integers (0, 1, INT_MAX)
- Mixed approaches
  - Genetic algorithms
  - Deep neural networks

Detecting abnormal behaviors

- Maximizing program coverage is only half the problem
- Additionally we need components that detect unwanted features
  - Buffer overruns
  - Uninitialized variables
  - Functions ending without returning a value
  - Division by zero
  - many-many more…
- When source code is available these components can be implemented as compiler passes

Coverage sanitizer (COVSAN)

- Compiler pass for GCC and Clang
- Compile with `-fsanitize-coverage=trace-pc-guard` or `-fsanitize-coverage=inline-8bit-counters`
- Inserts calls for logging the instruction pointer at function-, basic-block- or edge- levels, with optional 8-bit counters
- https://clang.llvm.org/docs/SanitizerCoverage.html

Address sanitizer (ASAN)

- Compiler pass for GCC and Clang (`-fsanitize=address`)
- Able to detect memory corruption bugs such as
  - Use after free
  - Buffer overflows (stack, heap and global)
  - Use after return
- https://clang.llvm.org/docs/AddressSanitizer.html

Address sanitizer (ASAN) (II)

- Uses shadow memory to store a map of “poisoned” memory
  - `Shadow = (Mem >> 3) + 0x20000000; (32-bit)`
- Every memory access is rewritten in order to check whether the memory is poisoned
  - Touching poisoned memory generates an error report
- Global and local variables are wrapped in areas of poisoned memory
- Allocated memory is prepended and appended with poisoned regions
- Freed memory is marked as poisoned

Undefined behavior sanitizer (UBSAN)

- Compiler pass for GCC and Clang
- A lot of available options (signed integer overflow, division by zero, static array out of bounds indexing, shifting with values larger than the bit-width)
- Implemented as additional checks before the instruction is actually performed
- A failed check triggers an error report
- https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

Memory Sanitizer (MSAN)

- Compiler pass for GCC and Clang
- Tracks uninitialized memory usage of both stack and heap variables
- Newly allocated memory is marked as uninitialized
- Writes to uninitialized memory turn it to initialized memory
- Reads of uninitialized memory triggers an error report
- Moving uninitialized memory is allowed (with optional origin tracking)
- https://clang.llvm.org/docs/MemorySanitizer.html

Libfuzzer

Source code, dumb, gray-box and evolutionary.

```
$ git clone https://git.llvm.org/git/compiler-rt.git/
$ cd lib/fuzzer
$ ./build.sh
## builds libFuzzer.a
## compile your library and test function in the `fuzzer` executable
$ clang++ -fsanitize=address -fsanitize-coverage=trace-pc-guard test_function.cc
library.c libFuzzer.a -o fuzzer
## test_function.cc contains the implementation of function:
## extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size);
## this function should cover the functionality of your library
## run the fuzzer using the parameters available in the documentation:
## https://llvm.org/docs/LibFuzzer.html#options
$ ./fuzzer
```

AFL

Symbolic execution

- An automated way of testing software

  Works both on source code as well as compiled binaries

- Able to generate test cases for debugging purposes

  Very useful for human inspection

- Similar to fuzz-testing but with a lot more brain-power

Fuzz testing vs. Symbolic execution

- Generate random inputs
  - Either in some structured format or completely random
  - Mutate a previous input
- Native run of the exercised program
- Collect information
  - Crashes
  - Memory & resource leaks
  - Failed assertions
  - Abnormal runtimes
- Repeat until confident enough
  - Fixed number of tests reached
  - Code coverage above a certain threshold
  - Bug reached

- Treat inputs as symbolic
  - Having no concrete value
- Run the program in a specialized virtual machine
- When execution branches, duplicate VM state and take both branches
  - Accumulate conditions along the execution paths
- Use a theorem prover to determine whether the conditions are consistent

Satisfiability modulo theories (SMT) solvers

- SMT solvers can be viewed as solvers of large equation systems
  - Most popular Z3, CVC4, CVC3
- Not limited to Boolean equations like SAT solvers

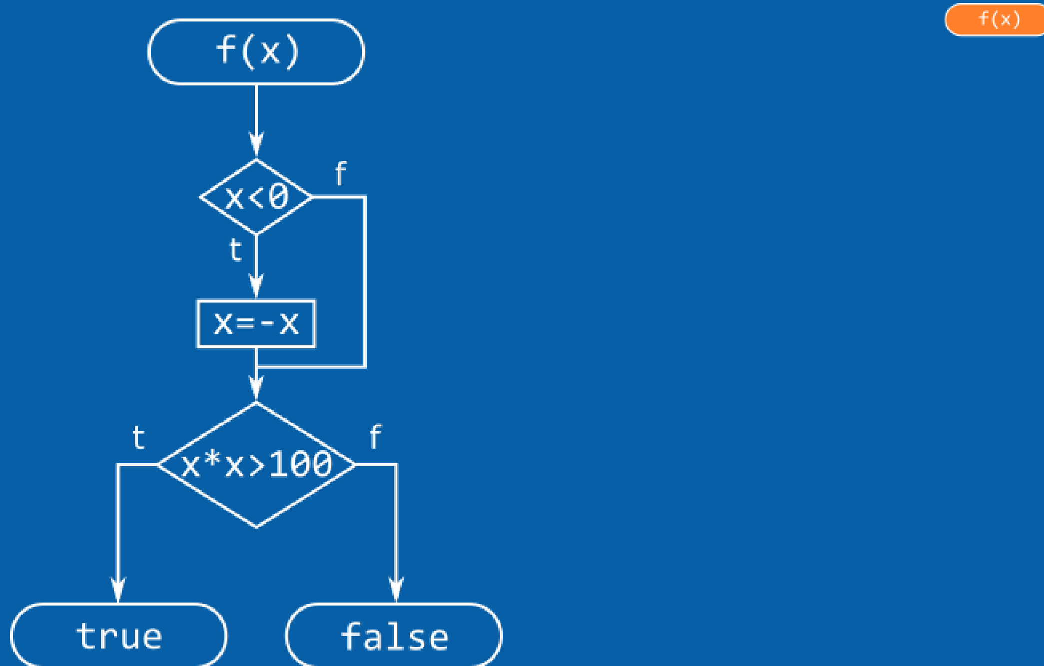
$$
\begin{aligned}
3x + 2y - z &= 1 \\
2x - 2y + 4z &= -2 \\
-x + \frac{1}{2}y - z &= 0
\end{aligned}
$$

```
#!/usr/bin/python
from z3 import *
x = Real('x')
y = Real('y')
z = Real('z')
s = Solver()
s.add(3*x + 2*y - z == 1)
s.add(2*x - 2*y + 4*z == -2)
s.add(-x + 0.5*y - z == 0)
print s.check()
print s.model()

sat
[z = -2, y = -2, x = 1]
```

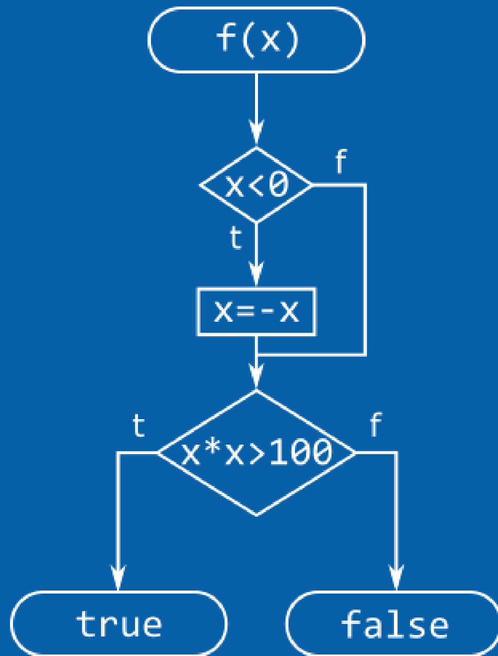
Symbolic execution example

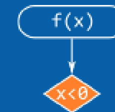
```
bool f(int x) {                  // x = λ
    if (x < 0) {
        x = -x;                  // x = -λ
    }                            // x = λ1 =
                                 //     ((λ < 0) => -λ) |
                                 //     (!(λ < 0) => λ)

    if (x * x > 100) {
        return true;
    } else {
        return false;
    }                            // return
                                 //     ((λ1 * λ1 > 100) => true) |
                                 //     (!(λ1 * λ1 > 100) => false)
}
```

Symbolic execution example

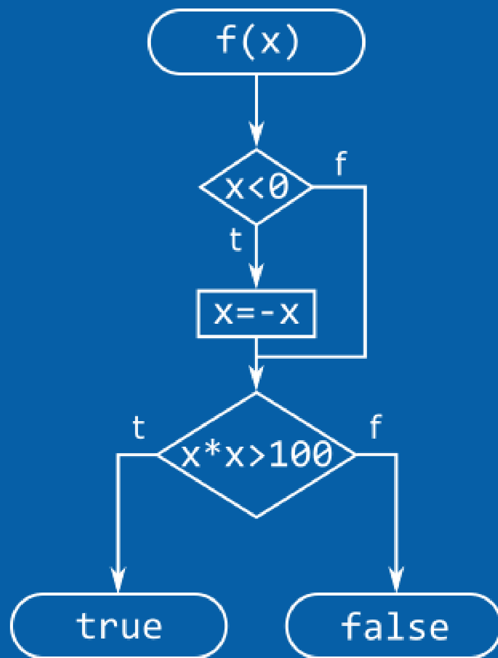


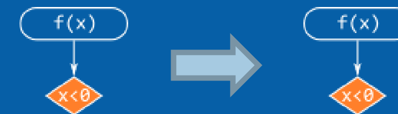
Symbolic execution example



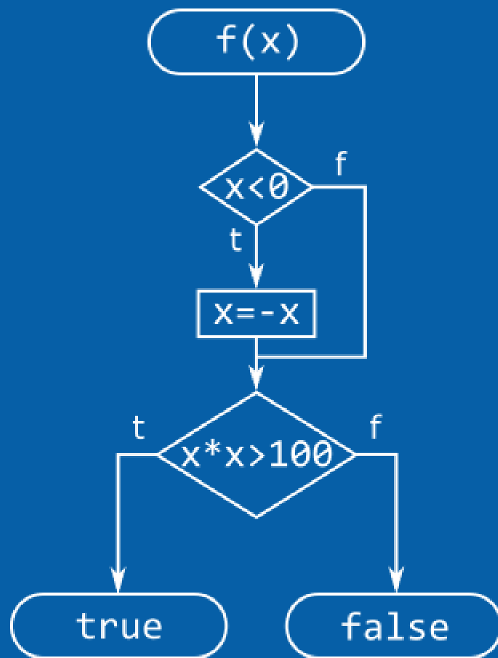


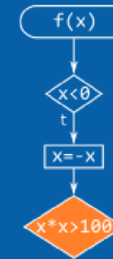
Symbolic execution example

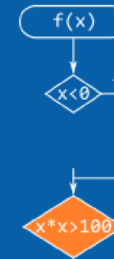




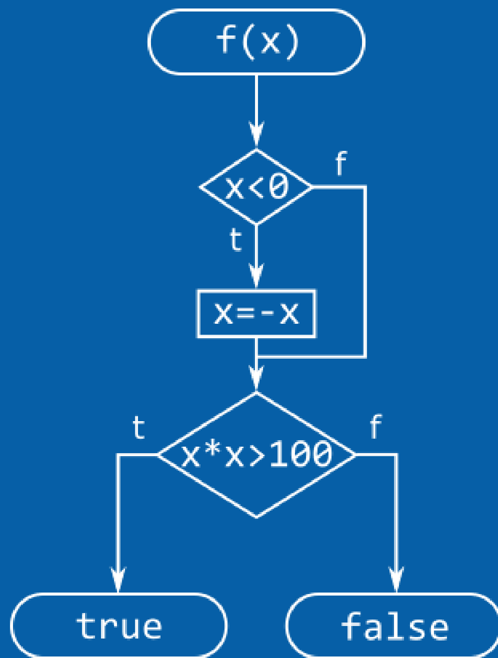
Symbolic execution example

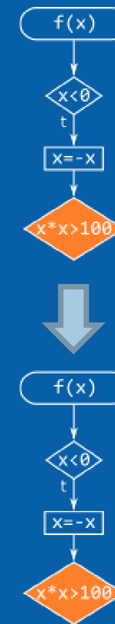


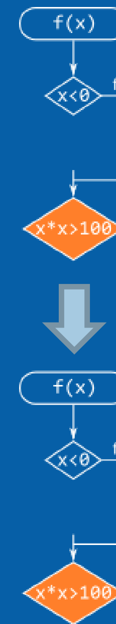




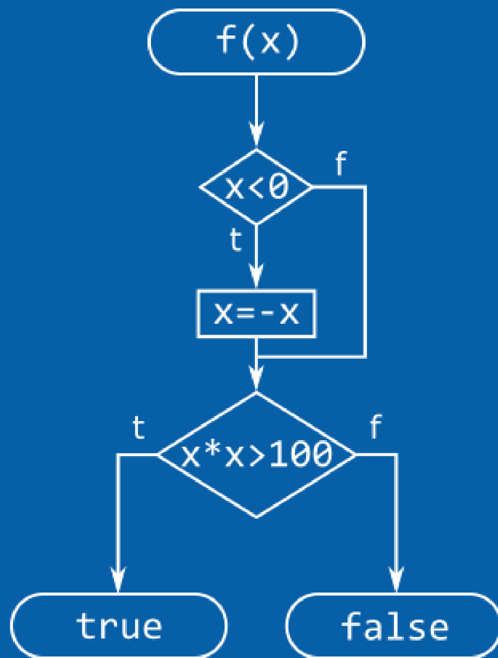
Symbolic execution example

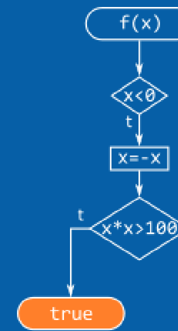


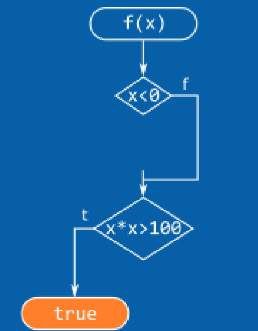


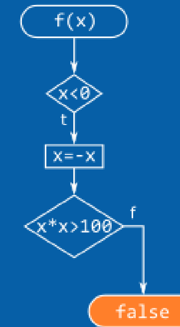


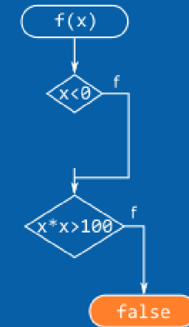
Symbolic execution example











Symbolic execution example

- Four execution paths have been analyzed
- Each path has a distinct set of conditions
- Conditions are fed into a satisfiability modulo theorem (SMT) solver
- The SMT solver determines whether the conditions are consistent
  - Optionally the solver can generate a test case that satisfies the input conditions

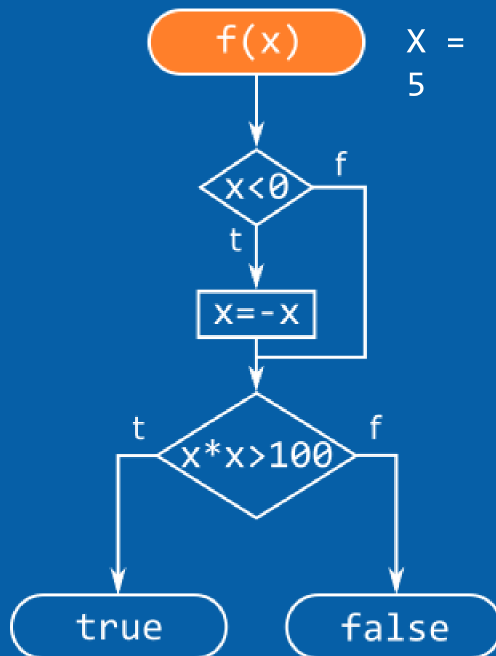
Symbolic execution drawbacks

- Path execution is an issue
  - Usually number of execution cores << path count. Some kind of scheduling is necessary.
  - Even if hardware is not an issue, duplicating a path may be a costly operation.
- If left unaddressed, path explosion can be an issue
  - Mitigation strategies include:
    - Selective symbolic execution – carefully select execution paths to be evaluated
    - Path collapsing – find common traits between paths and treat them as a single entity
- There is no hardware support for symbolic values
  - Exercised code is run in a heavily simulated environment

Enter "Concolic execution"

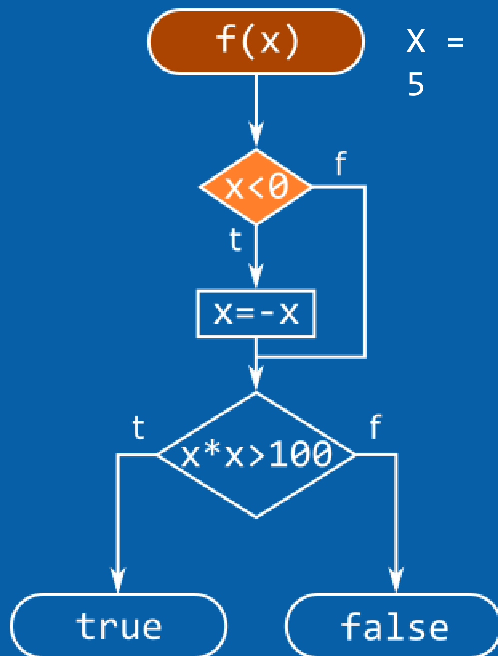
- Used to better take advantage of the underlying hardware
- The term concolic is a contraction of "**conc**rete symb**olic**"
  - I know, right?
- Instead of being 100% symbolic, the inputs have a concrete value
  - The concrete value is a representative of the symbolic domain
  - Use some form of taint analysis to track symbolic values
- Execution is sped up
  - Exercised code is run natively using concrete values
  - Taint analysis structures are usually pretty fast

Concolic execution example



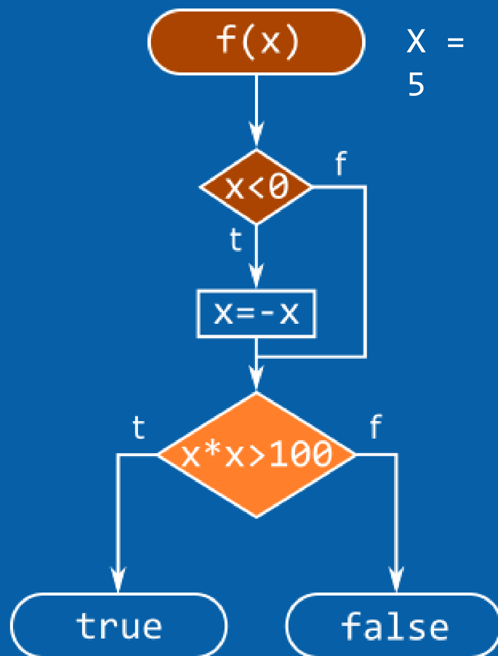
- Execution starts with a random value as input

Concolic execution example



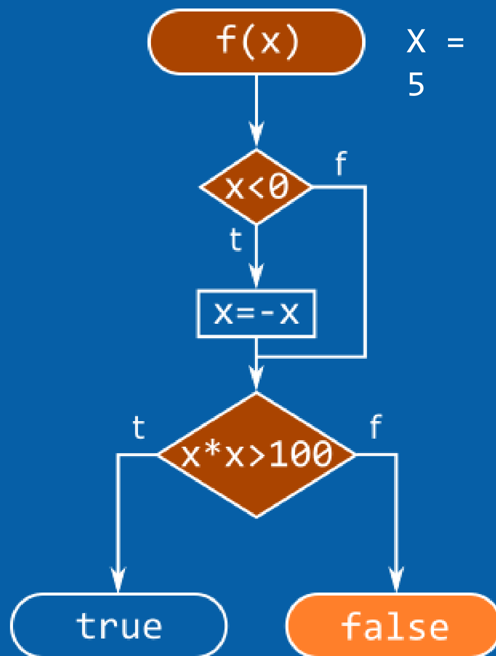
- Execution starts with a random value as input
- Conditions are accumulated at runtime
  - !(X < 0)

Concolic execution example



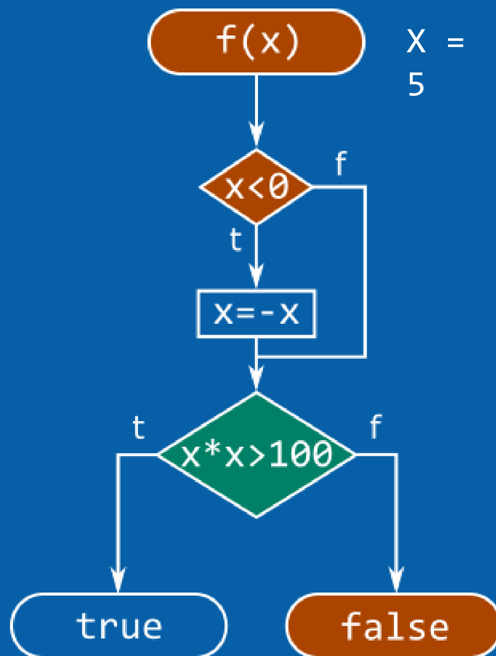
- Execution starts with a random value as input
- Conditions are accumulated at runtime
  - !(X < 0)
  - !(X * X > 100)

Concolic execution example



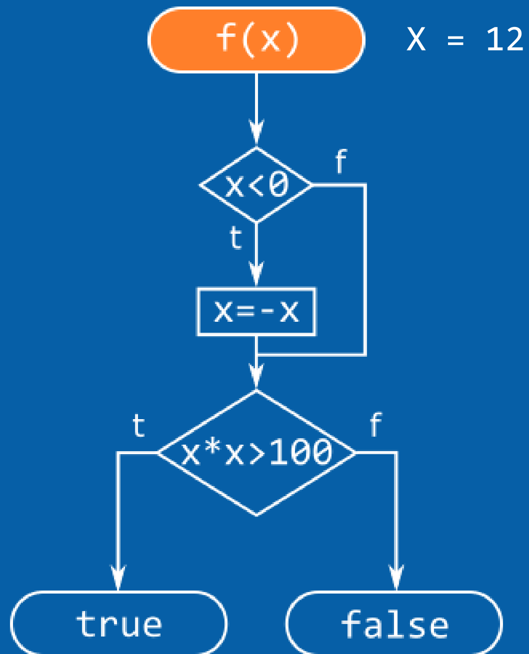
- Execution starts with a random value as input
- Conditions are accumulated at runtime
  - !(X < 0)
  - !(X * X > 100)
- The execution has stopped, we have a test case!

Concolic execution example



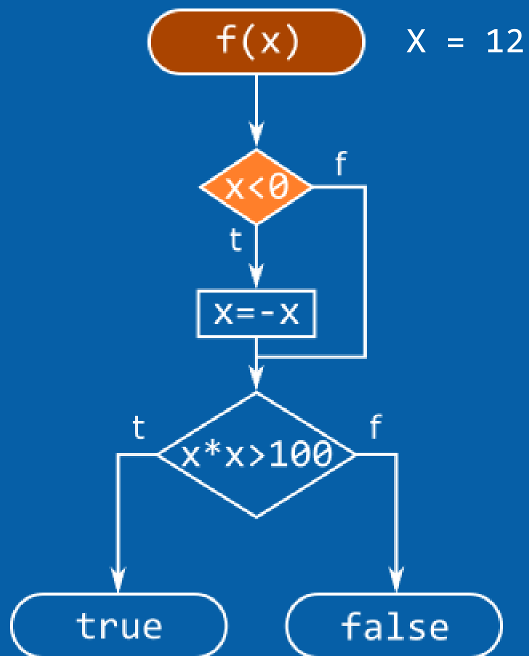
- In order to discover more test cases without repetitions an existing condition is selected
  - In our particular case !(X * X > 100)
- Inverting the condition and using the SMT solver will give us a new test case

Concolic execution example



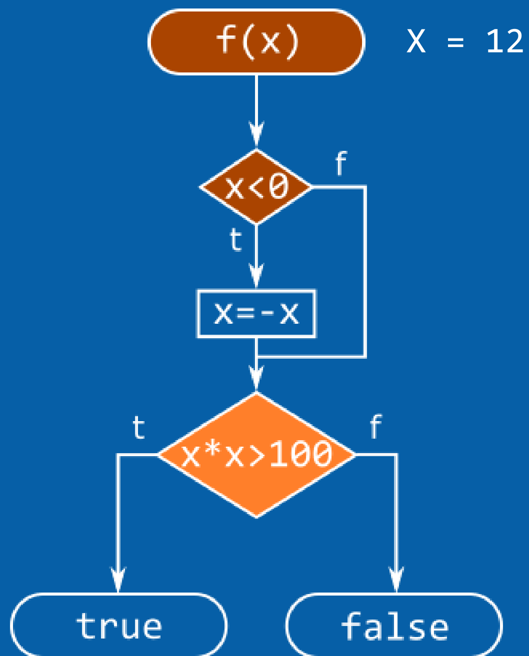
- If solvable, the SMT solver picks a test case
- The execution is restarted, having a new input value

Concolic execution example



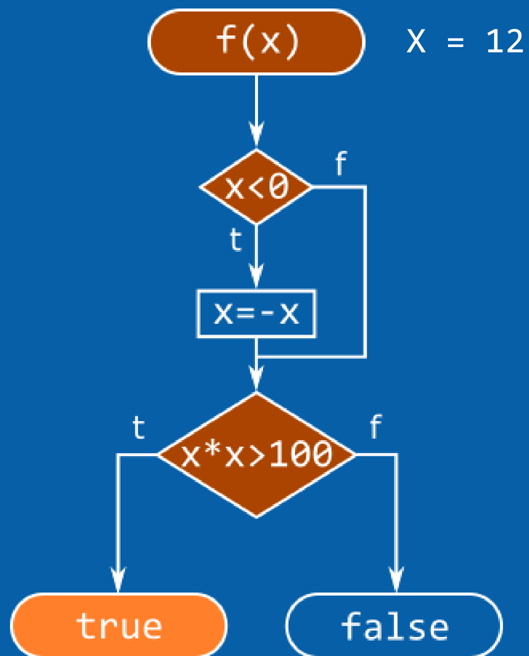
- If solvable, the SMT solver picks a test case
- The execution is restarted, having a new input value
- Conditions are accumulated at runtime
  - !(X < 0)

Concolic execution example



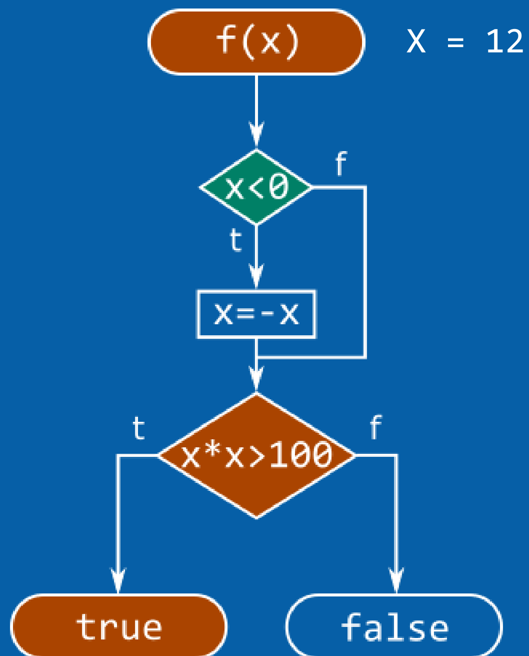
- If solvable, the SMT solver picks a test case
- The execution is restarted, having a new input value
- Conditions are accumulated at runtime
  - !(X < 0)
  - X * X > 100

Concolic execution example



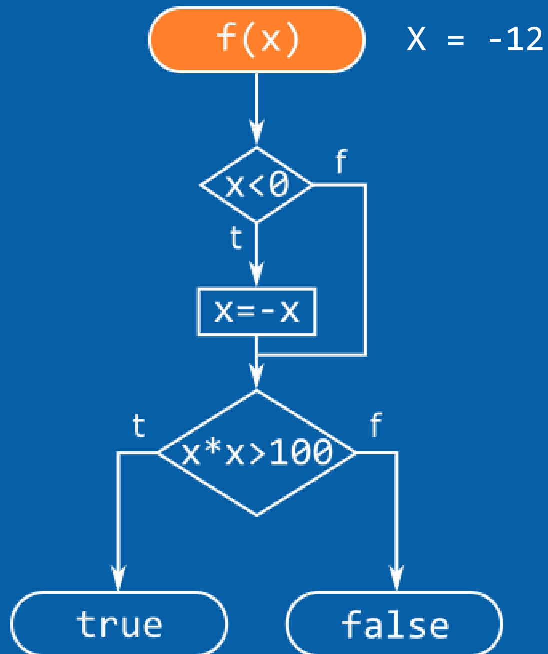
- If solvable, the SMT solver picks a test case
- The execution is restarted, having a new input value
- Conditions are accumulated at runtime
  - !(X < 0)
  - X * X > 100
- Execution terminates

Concolic execution example



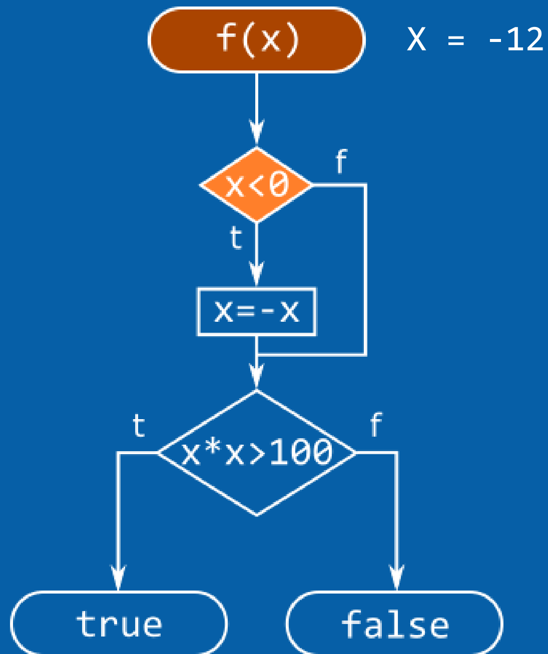
- Another condition is selected for reversing

Concolic execution example



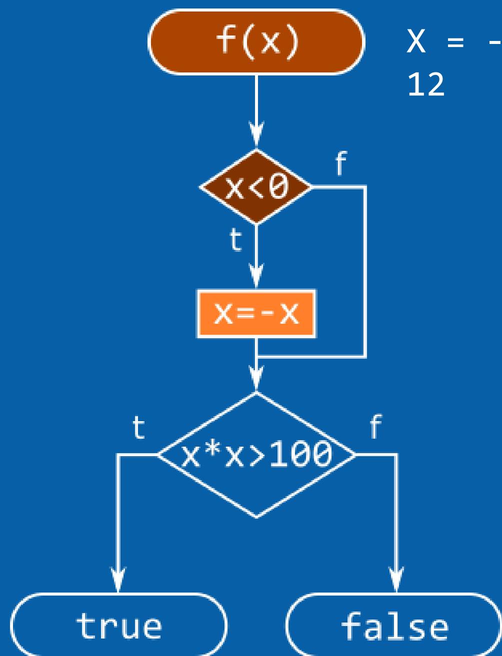
- Execution is restarted having a new input value

Concolic execution example



- Execution is restarted having a new input value
- Once more conditions are accumulated along the way
  - X < 0

Concolic execution example



You get the idea!

Concolic execution example

- We have reached the same result as pure symbolic execution!
- There is no need to simulate symbolic variables
- The whole symbolic execution VM is replaced with
  - A symbolic context for accumulating conditions
  - Some method of keeping the symbolic context in sync with the native execution

Existing technologies

- Klee, https://klee.github.io/
  - Symbolic virtual machine capable of running LLVM code
- S2E, http://s2e.epfl.ch/
  - Based on KLEE
  - Uses a x86-to-LLVM translator in order to run x86 code
  - Capable of running a full OS (using a modified QEMU)
- Triton, http://triton.quarkslab.com/
  - In development framework from University of Bordeaux
  - Lots of tools such as a taint engine, a symbolic execution engine, a snapshot engine
  - Interacts with a lot of SMT solvers (common interface named SMT-LIB2)
- Angr, https://github.com/angr/angr
  - Binary analysis framework from UC Santa Barbara
  - Python framework providing symbolic execution, control-flow analysis, data-dependency analysis and value-set analysis

Hybrid approaches

Driller = AFL + angr

- Determine when AFL is stuck
- Use Driller to feed new inputs to AFL
- Inputs are generated by resolving AFL’s unsatisfied conditions
- https://github.com/shellphish/driller

Libfuzzer & tracing CMP instructions

- Compiler flag for extracting operands of CMP instructions (`-fsanitize-coverage=trace-cmp`)
  - The fuzzer will guide mutations based on the CMP arguments
- Additional compiler flag (`-use-value-profile=1`) will use CMP operands as part of the coverage

Fuzzing Kernel Drivers with Interface Awareness

- Fuzzing specialized for ioctl's

```
int ioctl(int fd, unsigned long command, unsigned long param);
```

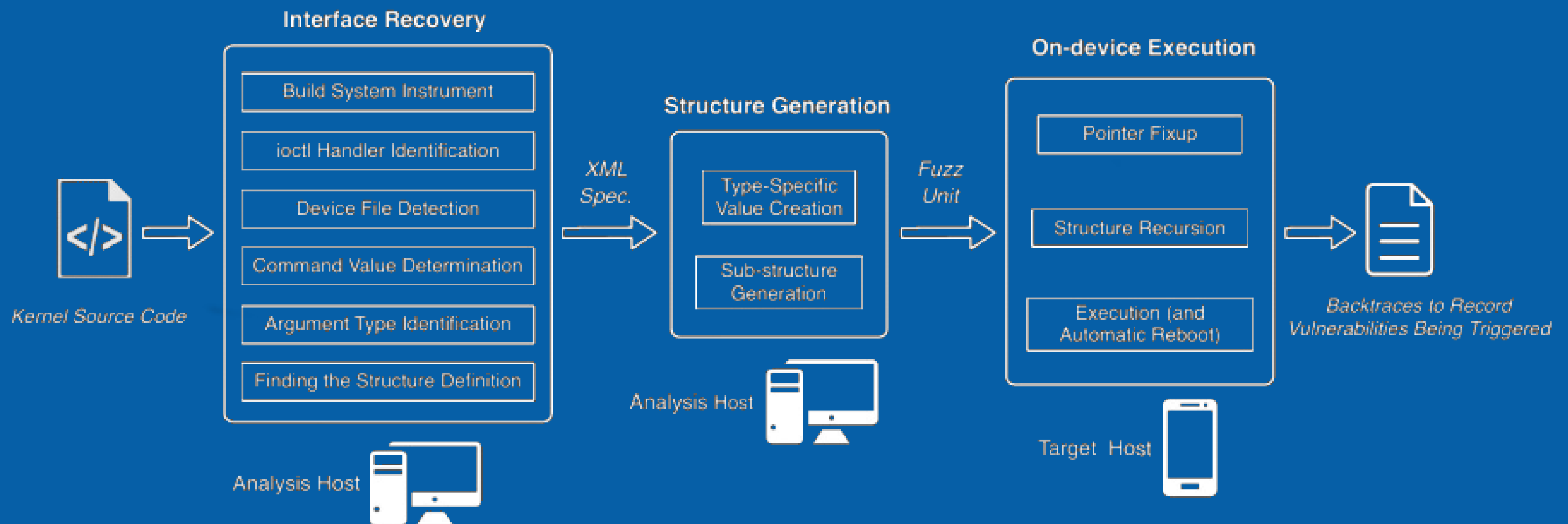
- Use static analisys to determine command and parameter type
- https://www.blackhat.com/docs/eu-17/materials/eu-17-Corina-Difuzzing-Android-Kernel-Drivers.pdf

```
static const struct file_operations IspFileOper = {
    .owner = THIS_MODULE,
    .open = ISP_open,
    .release = ISP_release,
    .mmap = ISP_mmap,
    .unlocked_ioctl = ISP_ioctl
};
```

DIFUZZER

```
static long ISP_ioctl(struct file *pFile, unsigned int Cmd, unsigned long Param)
{
    ...
    switch (Cmd) {
        case ISP_READ_REGISTER:
            if (copy_from_user(&RegIo, (void *)Param, sizeof(ISP_REG_IO_STRUCT)) == 0) {
                Ret = ISP_ReadReg(&RegIo);
            } else {
                LOG_ERR("copy_from_user failed");
                Ret = -EFAULT;
            }
            break;
        case ISP_WRITE_REGISTER:
            if (copy_from_user(&RegIo, (void *)Param, sizeof(ISP_REG_IO_STRUCT)) == 0) {
                Ret = ISP_WriteReg(&RegIo);
            } else {
                LOG_ERR("copy_from_user failed");
                Ret = -EFAULT;
            }
            break;
        case ISP_WAIT_IRQ:
            if (copy_from_user(&IrqInfo, (void *)Param, sizeof(ISP_WAIT_IRQ_STRUCT)) == 0) {
                ...
            }
            break;
        ...
    }
...
}
```

DIFUZZER (II)



Thank you for your time!