CE 874 - Secure Software Systems

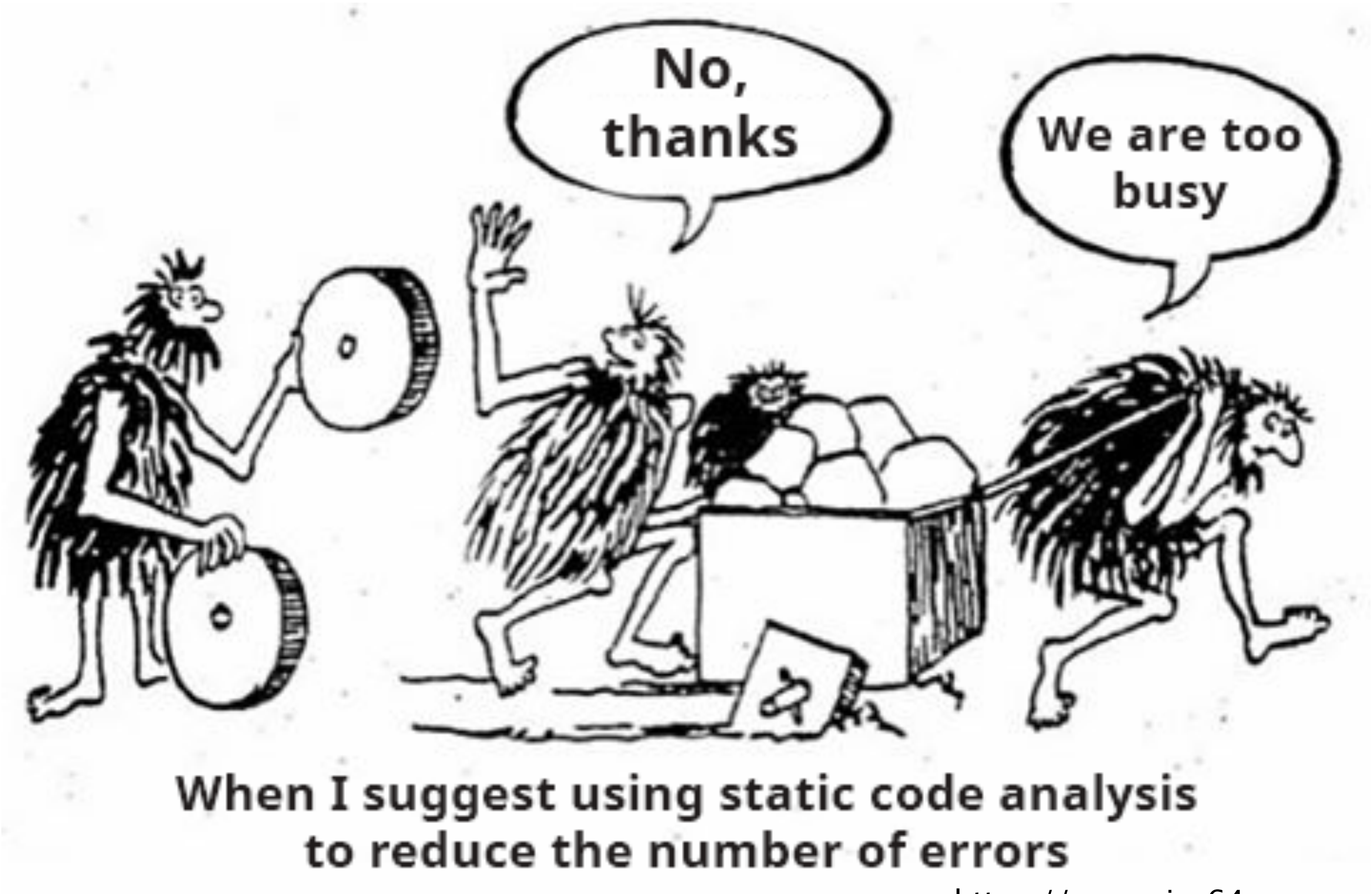
Fuzzing

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

Program Analysis

- How could we analyze a program (with source code) and look for problems?
- How accurate would our analysis be without executing the code?
- If we execute the code, what input values should we use to test/analyze the code?
- What if we don't have the source code?



When I suggest using static code analysis to reduce the number of errors

https://www.viva64.com

Fuzzing

I wrote a vulnerability scanner that abstracts all the predicates in a all the predicates in a binary, traverses the callgraph and generates phormulaes to run then with a SMT solver.
I found 1 vuln in 3 days with this tool.
He wrote a dumb ass fuzzer and found 5 vulns in 1 day.
Good thing I'm not a n00b like that guy.

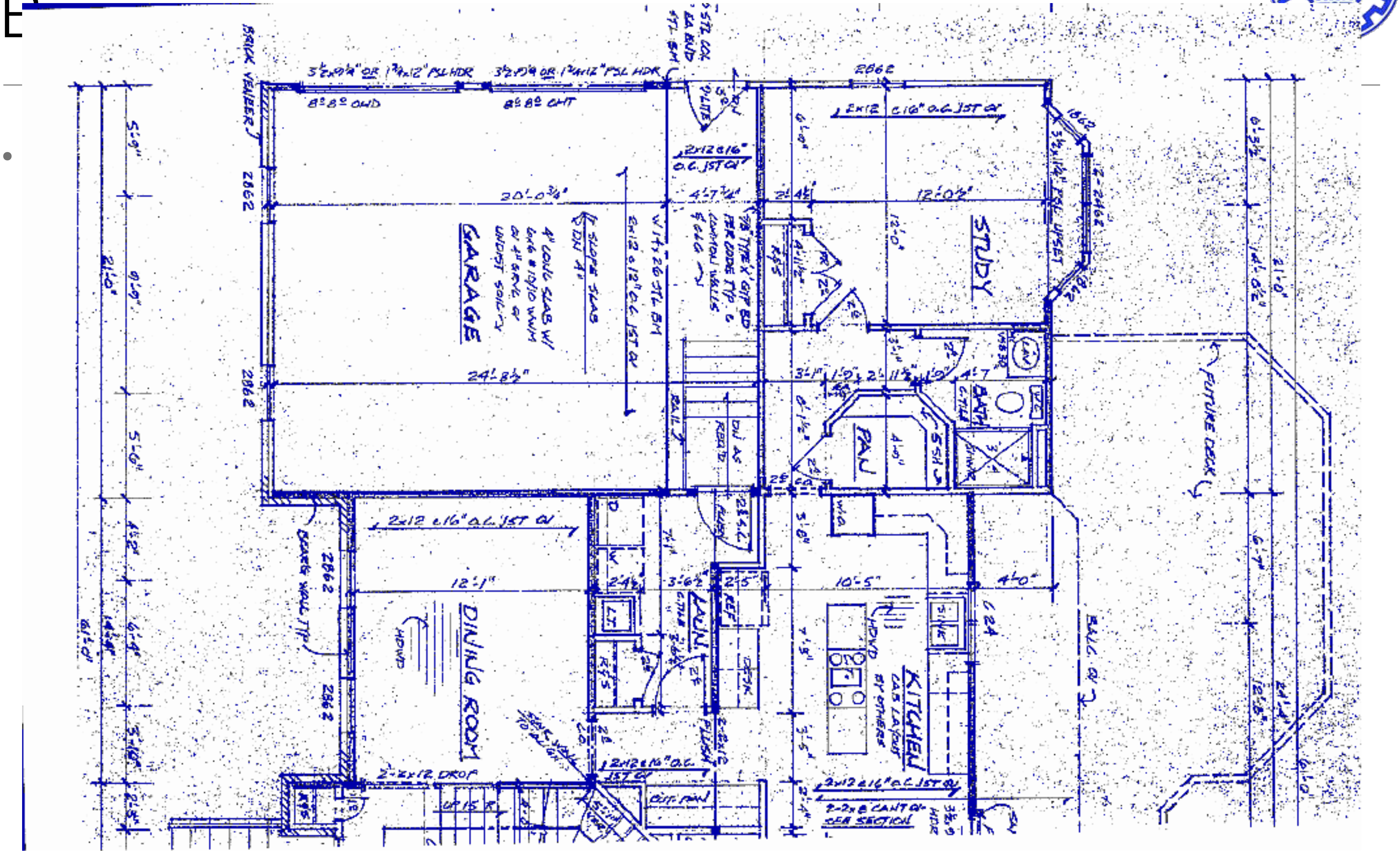
By: Joxean Koret

Vulnerability Discovery Methodologies – White Box

- Source code review
  - Static analysis
  - Pros
    - Coverage
  - Cons
    - Dependencies
    - Are we testing reality?
      - Compiler issues
      - Implementation scenarios

[Sutton'06]

Vulnerability Discovery Methodologies – White B



[Sutton'06]

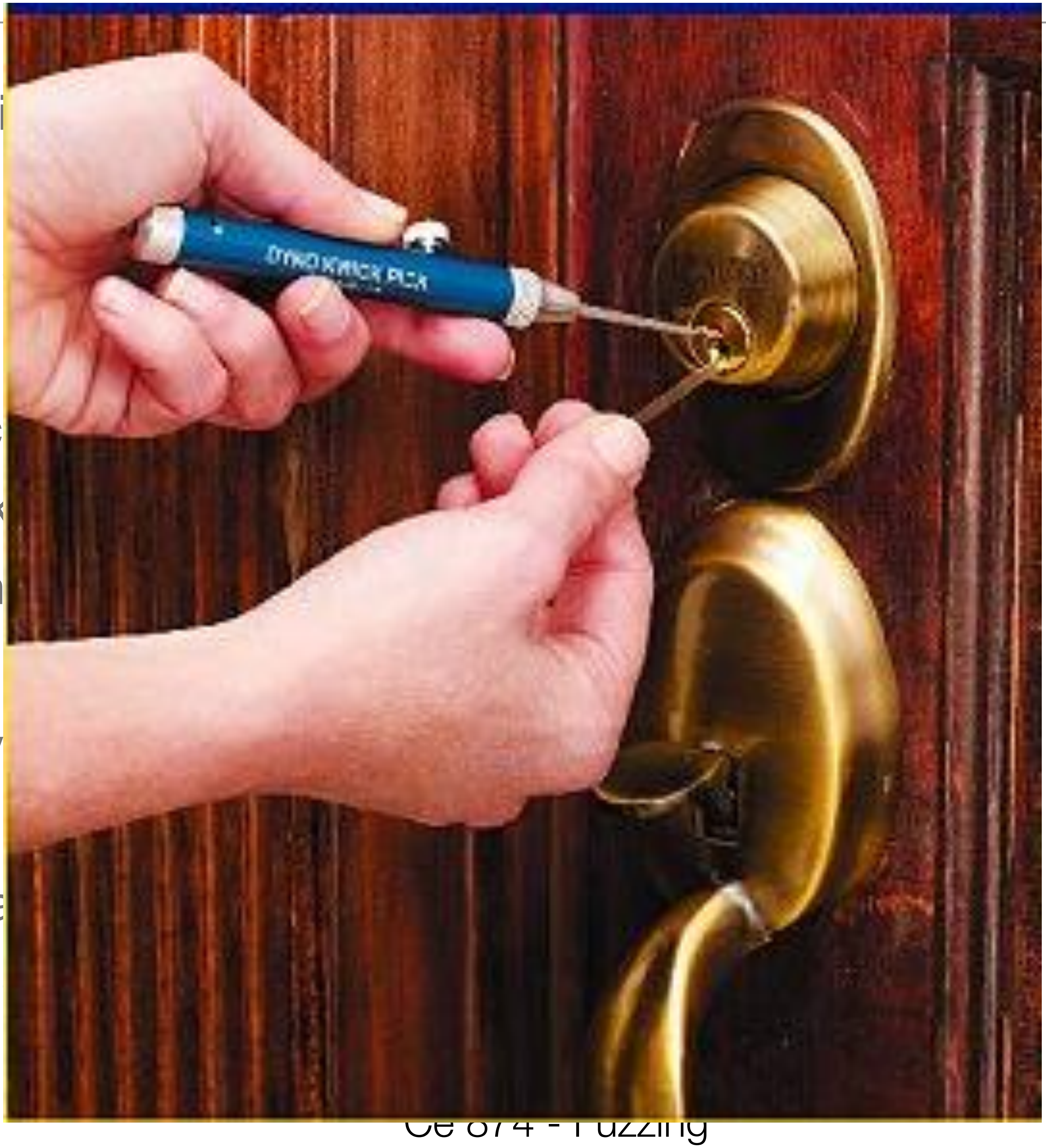
Vulnerability Discovery Methodologies – Black Box

- Reverse engineering (Static analysis)
  - Pros
    - Complex vulnerabilities uncovered
  - Cons
    - Time consuming
    - Deep knowledge required
- Fuzzing (Dynamic analysis)
  - Pros
    - Relatively simple and Realistic
  - Cons
    - Coverage
    - Complex vulnerabilities missed

[Sutton'06]

Vulnerability Discovery Methodologies – Black Box

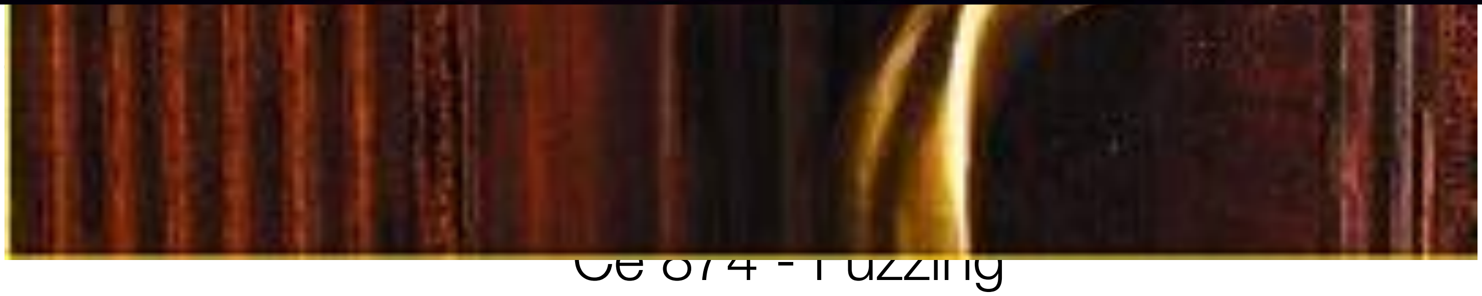
- Reverse engi
  - Pros
    - Compl
  - Cons
    - Time c
    - Deep k
- Fuzzing (Dyn
  - Pros
    - Relativ
  - Cons
    - Covera
    - Compl



[Sutton'06]

Vulnerability Discovery Methodologies – Black Box

- Compl



[Sutton'06]

What is Fuzzing?

- “Unexpected input causes unexpected results.”
  - Michael Sutton

[Sutton’06]

What is Fuzzing?

- “Unexpecte
  - Michael

[Sutton’06]

Fuzzing

- Fuzzing is an optimization problem
  - Available resources are used to discover as many bugs as possible, covering as much of the program as possible, covering as much of the program functionality as possible through a probabilistic exploration process.
- Challenges:
  - Input generation
    - balance between exploring new paths (control flow) and executing the same paths with different input (data flow)
  - Execution engine
    - Not every bug results in an immediate segmentation fault and detecting state violation is a challenging task
    - have to distinguish between crashes

[Payer’19]

Fuzzing

- Challenges (con’t)
  - Coverage wall:
    - fuzzing can get stuck in local minima where continuous input generation will not result in additional crashes or new coverage
  - Evaluating fuzzing effectiveness:
    - How do you compare fuzzers?

[Payer’19]

What data can be fuzzed?

- Virtually anything!
- Basic types: bit, byte, word, dword, qword
- Common language specific types: strings, structs, arrays
- High level data representations: text, xml

[Heasman'09]

Where can data be fuzzed?

- Across any security boundary, e.g.:
  - An RPC interface on a remote/local machine
  - HTTP responses & HTML content served to a browser
  - Any file format, e.g. Office document
  - Data in a shared section
  - Parameters to a system call between user and kernel mode
  - HTTP requests sent to a web server
  - File system metadata
  - ActiveX methods
  - Arguments to SUID binaries

[Heasman'09]

Simple fuzzing ideas

- What inputs would you use for fuzzing?
- very long or completely blank strings
- max. or min. values of integers, or simply zero and negative values
- depending on what you are fuzzing, include special values, characters or keywords likely to trigger bugs, eg
  - nulls, newlines, or end-of-file characters
  - format string characters
  - semi-colons, slashes and backslashes, quotes
  - application specific keywords halt, DROP TABLES, ...
  - ....

Crash/error detection

- Crash detection is critical for fuzzing!
- To help in crash detection, debugging tools aka runtime checkers can be used, such as valgrind, Purify, AddressSanitizer, ...
    - Such tools instrument code or run to code on simulators to catch more bugs
        - Valgrind provides:
            - memcheck memory error detector, which detects buffer overruns, malloc/free errors, memory leaks, reads of uninitialised memory, ...
            - helgrind detector to help detect data races, deadlocks, and incorrect use of the POSIX threads API

[Poll'18]

Fuzzing web-applications

- How could a fuzzer detect SQL injections or XSS weaknesses?
  - For SQL injection: monitor database for error messages
  - For XSS, see if the website echoes HTML tags in user input
- There are various tools to fuzz web-applications: Spike proxy, HP Webinspect, AppScan, WebScarab, Wapiti, w3af, RFuzz, WSFuzzer, SPI Fuzzer Burp, Mutilidae, ...
- Some fuzzers crawl a website, generating traffic themselves, other fuzzers modify traffic generated by some other means.
- Can we expect false positives/negatives?
  - false negatives due to test cases not hitting the vulnerable cases
  - false positives & negatives due to incorrect test oracle, eg
    - for SQL injection: not recognizing some SQL database errors (false neg)
    - for XSS: signaling quoted echoed response as XSS (false pos)

[Poll’18]

Mutational Fuzzing

Smarter fuzzing

- Mutation-based fuzzers: apply random mutations to existing valid inputs
  - Observe network traffic, than replay with some modifications
  - More likely to produce interesting invalid inputs than just random input
- Generation-based aka grammar-based fuzzers
  generate semi-well-formed inputs from scratch, based on some knowledge of file format or network protocol
  - Downside: more work to construct the fuzzer
- Evolutionary fuzzers: observe how inputs are processed to learn which mutations are interesting
  - For example, afl, which uses a greybox approach
- Whitebox approaches: analyse source code to determine interesting inputs
  - For example, SAGE

[Poll'18]

Example: find bugs in pdf viewer

→ PDF Viewer

Crash viewer and isolate cause

Modify conventional pdf files

PDF Readers

- PDF format is extremely complicated
  - PDF files can require complex rendering
    - Flash, Quicktime video, 3-d animation,
  - PDF files can include executable JS
    - Extremely complicated code base

[Mitchell’18]

Mutation-based fuzzing with pdf

- Similar process used for Acrobat, Preview (Apple)
  - Collect a large number of pdf files
    - Aim to exercise all features of pdf readers
    - Found 80,000 PDF's on Internet
  - Reduce to smaller set with apparently equivalent code coverage
    - Used Adobe Reader + Valgrind in Linux to measure code coverage
    - Reduced to 1,515 files of 'equivalent' code coverage
    - Same effect as fuzzing all 80k in 2% of the time
  - Mutate these files and test Acrobat (Preview) for vulnerabilities

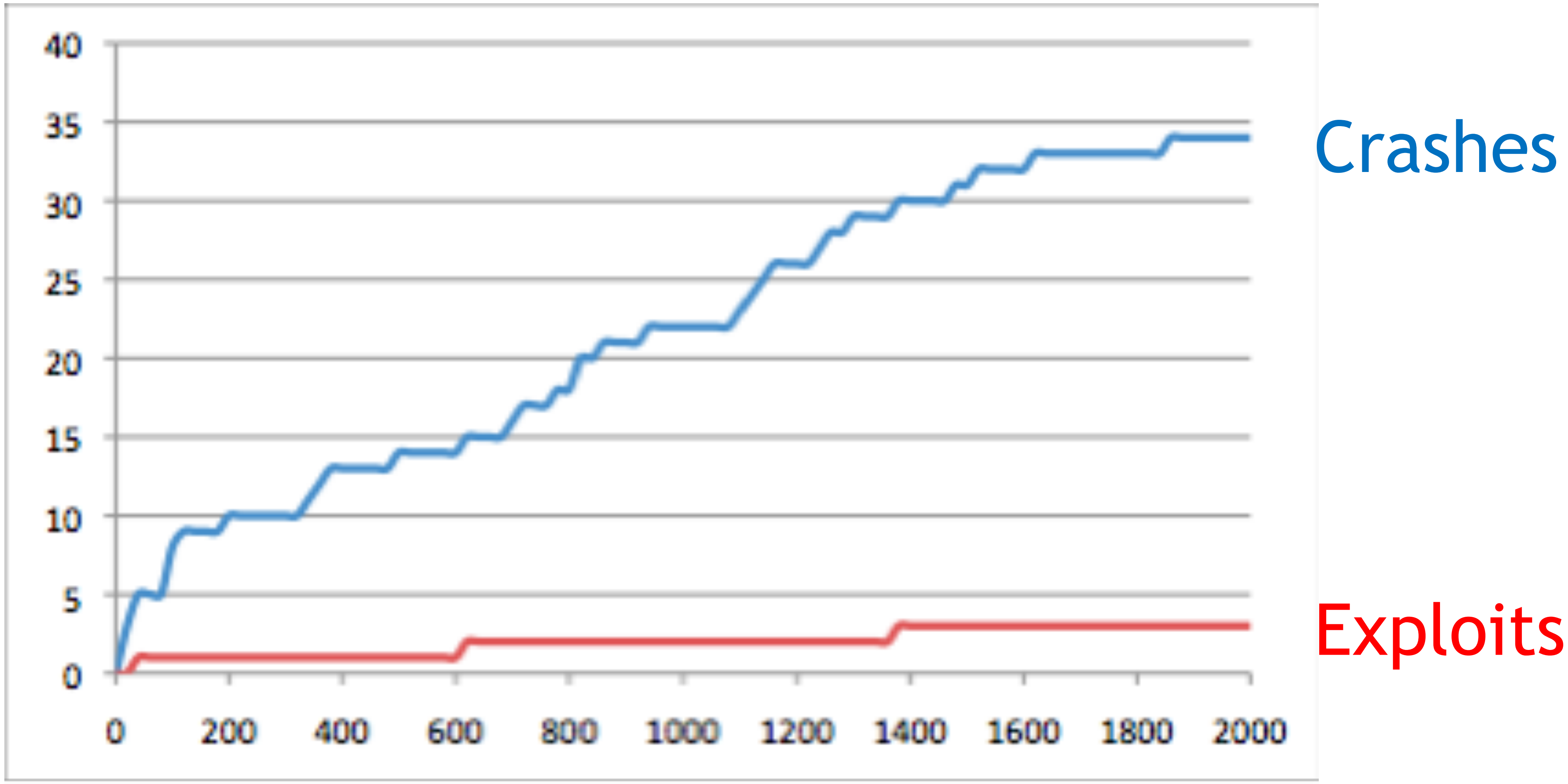
[Mitchell'18]

Mutation

- Modify each file in a number of ways
  - Randomly change selected bytes to random values
  - Produce ~3 million test cases from 1500 files
    - Approximately same numbers for Acrobat and Preview,
    - Even though code and methods for testing code coverage are different
  - Use standard platform-specific tools to determine if crash represents a exploit
    - Acrobat: 100 unique crashes, 4 actual exploits
    - Preview: maybe 250 unique crashes, 60 exploits (tools may over-estimate)

[Mitchell’18]

Was this enough fuzzing?



Shape of curve suggests more possible: keep fuzzing until diminishing returns

Generational fuzzing
aka
Grammar-based fuzzing

CVEs as inspiration for fuzzing file formats

- Microsoft Security Bulletin MS04-028
  Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution
  Impact of Vulnerability: Remote Code Execution
  Maximum Severity Rating: Critical

Root cause: a zero sized comment field, without content.

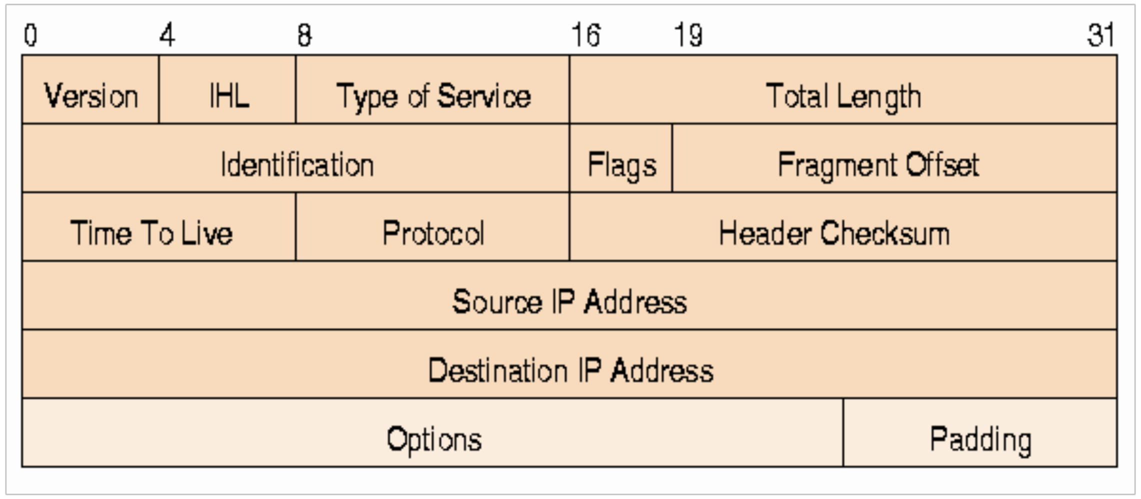
- CVE-2007-0243
  Sun Java JRE GIF Image Processing Buffer Overflow Vulnerability
  Critical: Highly critical   Impact: System access   Where: From remote

Description: A vulnerability has been reported in Sun Java Runtime Environment (JRE). … The vulnerability is caused due to an error when processing GIF images and can be exploited to cause a heap-based buffer overflow via a specially crafted GIF image with an image width of 0. Successful exploitation allows execution of arbitrary code.

Note: a buffer overflow in (native library of) a memory-safe language

Generation-based fuzzing

- For a given file format or communication protocol, a generational fuzzer tries to generate files or data packets that are slightly malformed or hit corner cases in the spec.

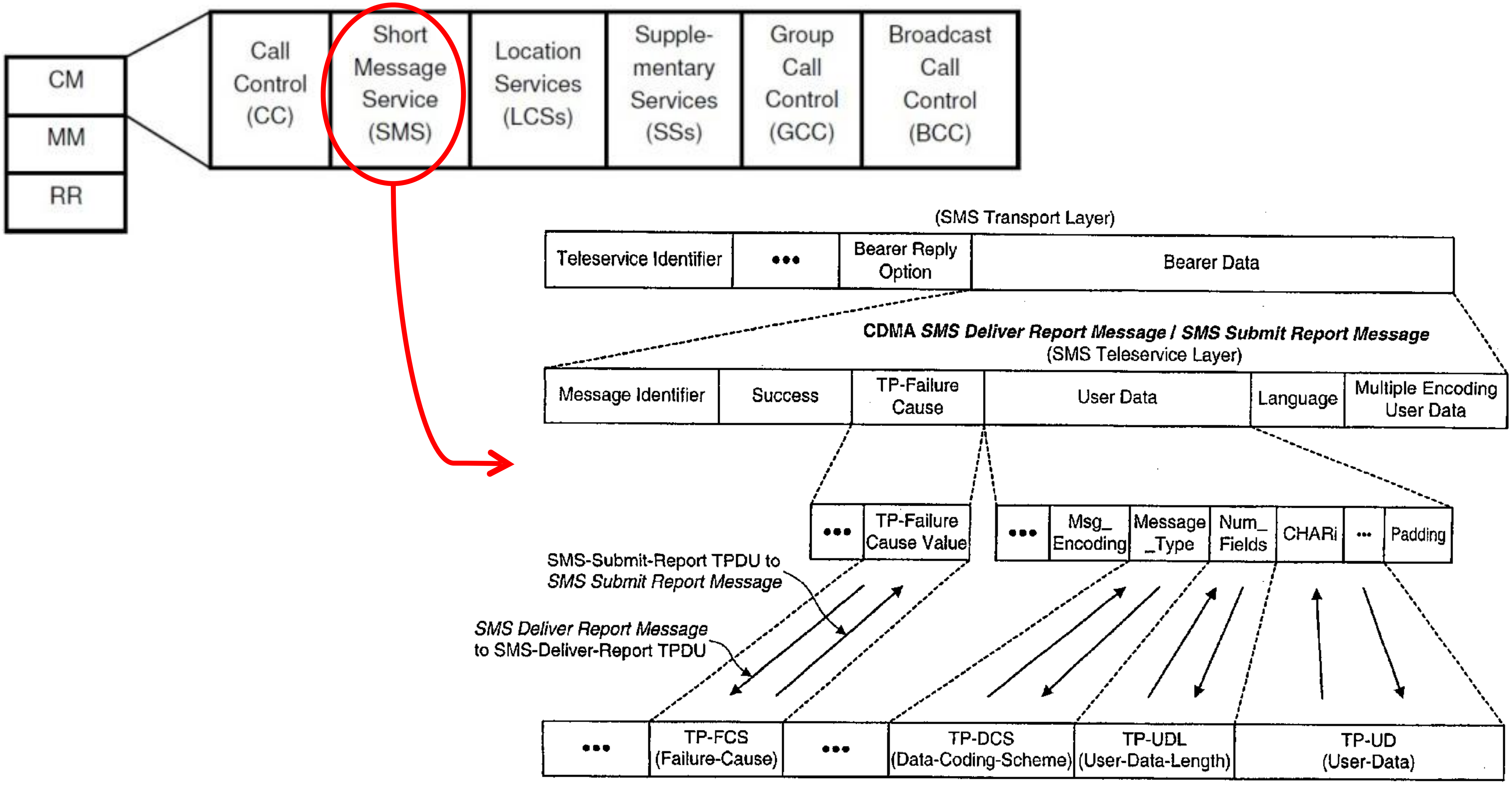


0	4	8	16	19	31
Version	IHL	Type of Service	Total Length		
Identification			Flags	Fragment Offset	
Time To Live		Protocol	Header Checksum		
Source IP Address					
Destination IP Address					
Options					Padding

- Possible starting :
  grammar defining legal inputs,
  or a data format specification
- Typical things to fuzz:
- many/all possible value for specific fields especially undefined values, or values Reserved for Future Use (RFU)
- incorrect lengths, lengths that are zero, or payloads that are too short/long
- Tools for building such fuzzers:
  SNOOZE, SPIKE, Peach, Sulley, antiparser, Netzob, ...

Example : GSM protocol fuzzing [MSc theses of Brinio Hond and Arturo Cedillo Torres]

- GSM is a extremely rich & complicated protocol



[Poll'18]

SMS message fields

Field	size
Message Type Indicator	2 bit
Reject Duplicates	1 bit
Validity Period Format	2 bit
User Data Header Indicator	1 bit
Reply Path	1 bit
Message Reference	integer
Destination Address	2-12 byte
Protocol Identifier	1 byte
Data Coding Scheme (CDS)	1 byte
Validity Period	1 byte/7 bytes
User Data Length (UDL)	integer
User Data	depends on CDS and UDL

[Poll’18]

Example: GSM protocol fuzzing

- Lots of stuff to fuzz!
- We can use a USRP

- with open source cell tower software (OpenBTS)
- to fuzz any phone

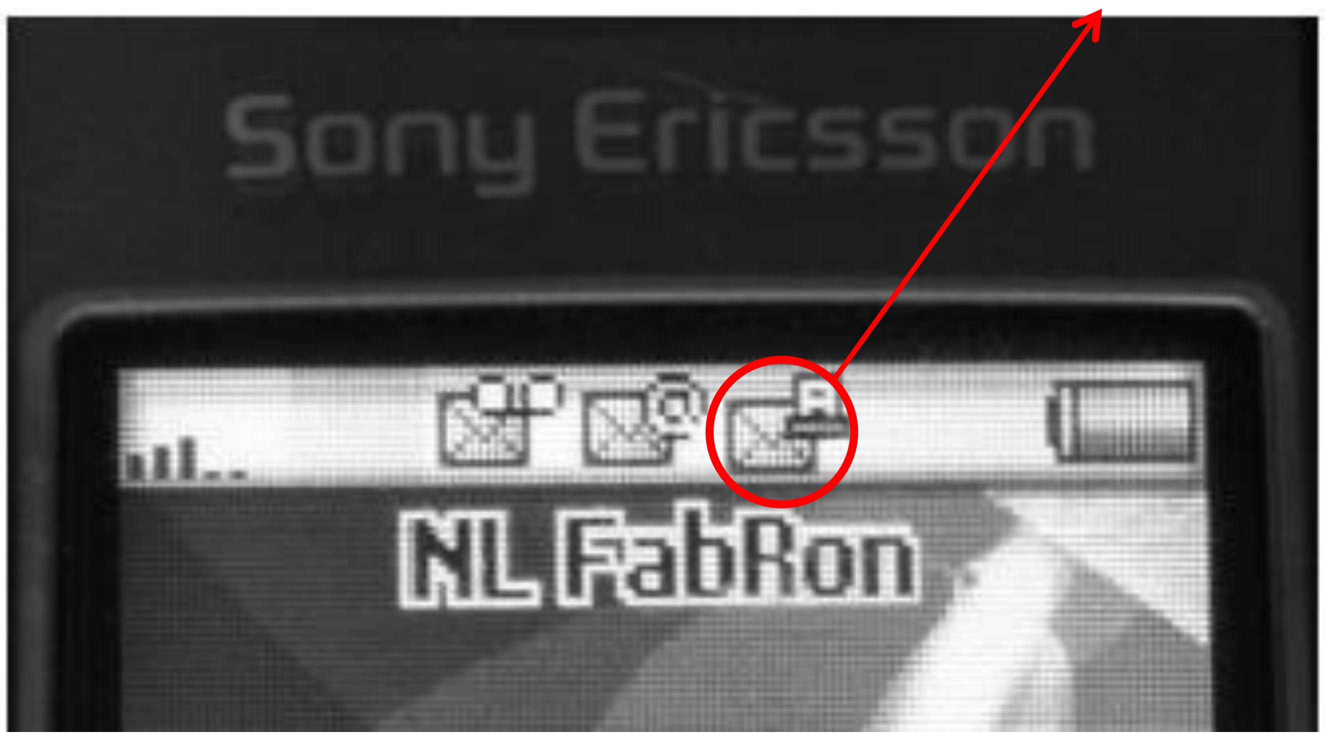
Example: GSM protocol fuzzing

- Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and in phones

Example: GSM protocol fuzzing

- Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and in phones
  - eg possibility to receive faxes (!?)

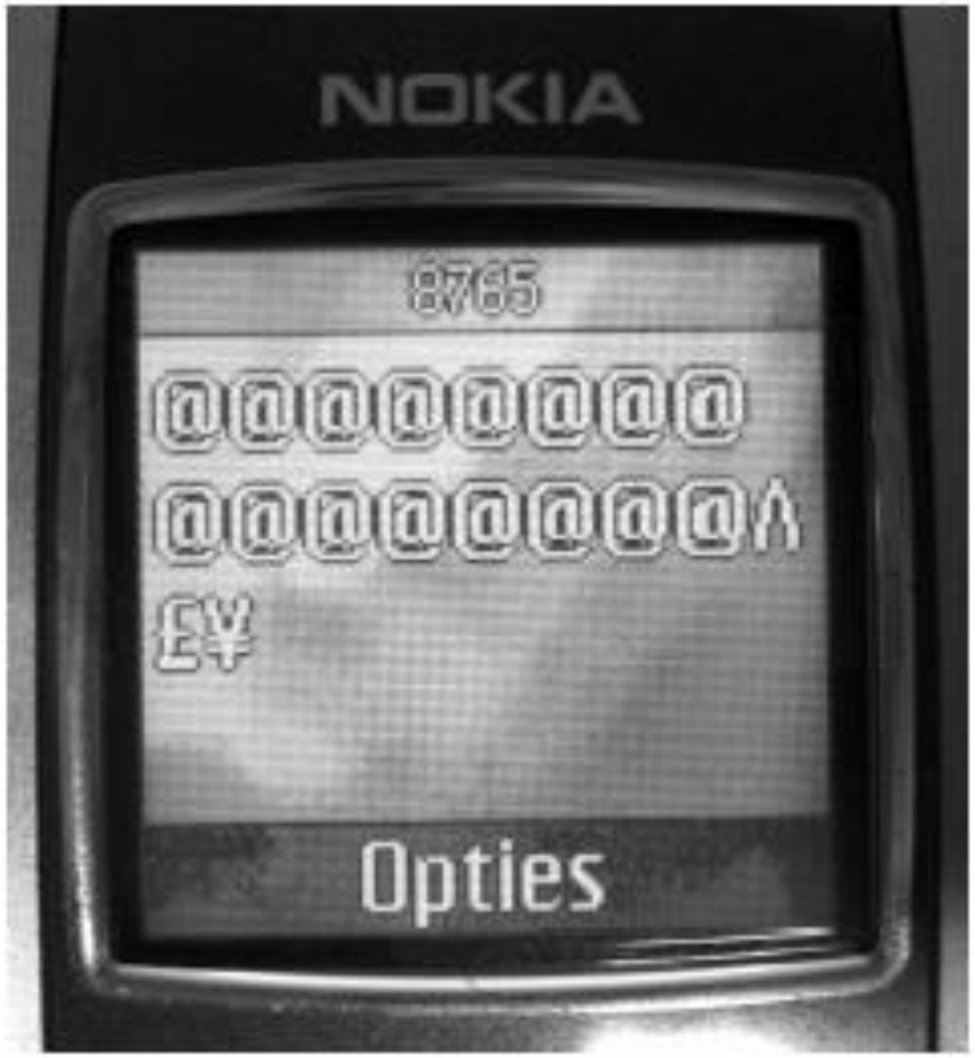
you have a fax!



  - Only way to get rid if this icon; reboot the phone

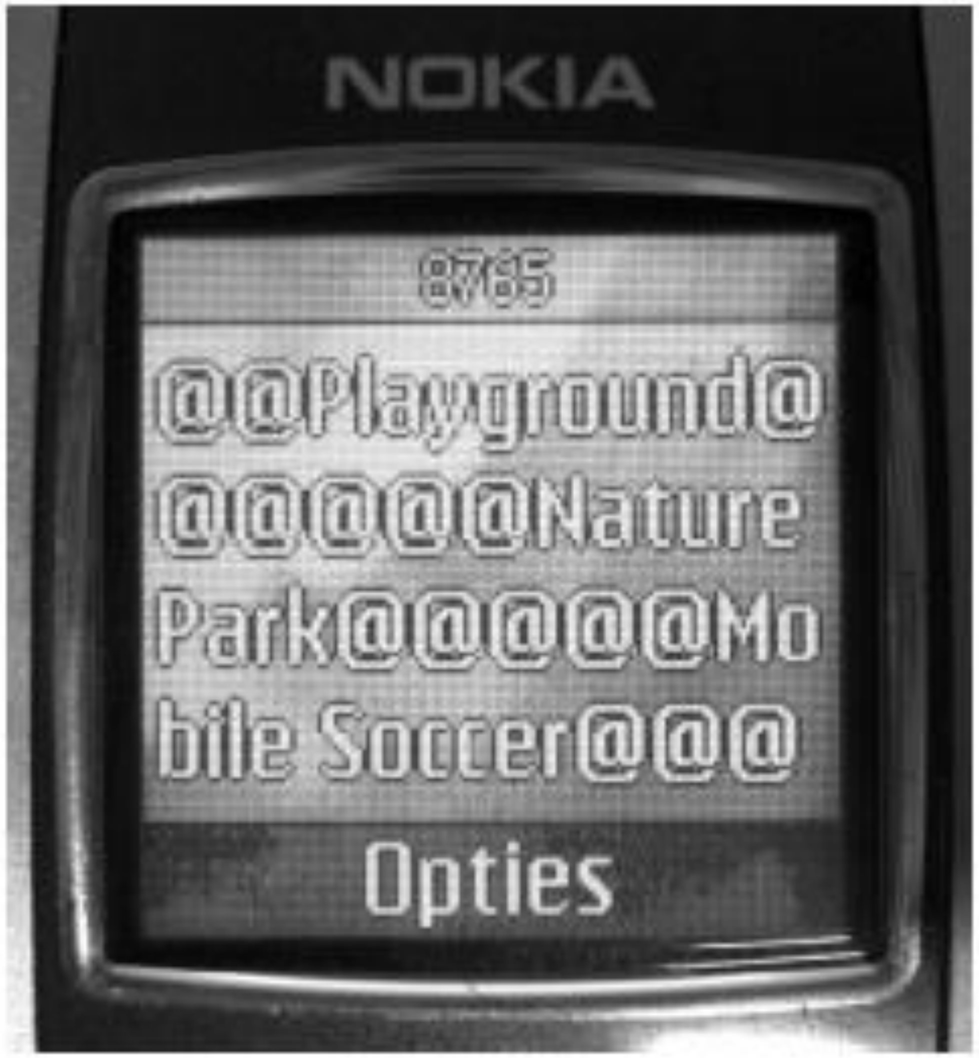
Example: GSM protocol fuzzing

- Malformed SMS text messages showing raw memory contents, rather than content of the text message



@@@@@@@@@
@@@@@@@@@Λ
£¥

(a) Showing garbage



@@Playground@
@@@@@Nature
Park@@@@@Mo
bile Soccer@@@

(b) Showing the name of a wallpaper and two games

[Poll’18]

Results with GSM protocol fuzzing

- Lots of success to DoS phones: phones crash, disconnect from the network, or stop accepting calls
  - requiring reboot or battery removal to restart, to accept calls again, or to remove weird icons
  - after reboot, the network might redeliver the SMS message, if no acknowledgement was sent before crashing, re-crashing phone
    - But: not all these SMS messages could be sent over real network
- There is surprisingly little correlation between problems and phone brands & firmware versions
- The scary part: what would happen if we fuzz base stations?
  - [Fabian van den Broek, Brinio Hond and Arturo Cedillo Torres, Security Testing of GSM Implementations, Essos 2014]
  - [Mulliner et al., SMS of Death, USENIX 2011]

Whitebox fuzzing with SAGE

Whitebox fuzzing using symbolic execution

- The central problem with fuzzing: how can we generate inputs that trigger interesting code executions?
  - fuzzing the procedure below is unlikely to hit the error case

```
int foo(int x) {
   y = x+3;
        if (y==13) abort(); // error
     }
```

- The idea behind whitebox fuzzing: if we know the code, then by analysing the code we can find interesting input values to try.
- SAGE (Scalable Automated Guided Execution) is a tool from Microsoft Research that uses symbolic execution of x86 binaries to generate test cases.

```
m(int x,y){
   x = x + y;
   y = y – x;
   if (2*y > 8) { ...
               }
   else if (3*x < 10){ ...
                     }
   }
```

Can you provide values for *x* and *y* that will trigger execution of the two if-branches?

Symbolic execution

```
m(int x,y){
   x = x + y;
   y = y - x;
   if (2*y > 8) { ...
                }
   else if (3*x < 10){ ...
                     }
   }
```

Suppose `x` = N and `y` = M.

`x` becomes N+M

`y` becomes M-(N+M) = -N

if-branch taken if $-2N > 8$, ie $N < -4$

Aka the path condition

2nd if-branch taken if
$N \geq -4$ & $3(M+N) < 10$

SMT solvers (such as Yikes, Z3, ...) are tools that can simplify such constraints and produce test data that meets them, or prove that they are not satisfiable.

This generates test data (i) automatically and (ii) with good coverage.

Symbolic execution for test generation

- Symbolic execution can be used to automatically generate test cases with good coverage
- Basic idea symbolic execution: instead of giving variables a concrete value (say 42), variables are given a symbolic value (say α), and the program is executed with these symbolic values to see when certain program points are reached
- Downside of symbolic execution:
  - it is very expensive (in time & space)
  - things explode with loops
  - …
  - SAGE mitigates this by using a single symbolic execution to generate many test inputs for many execution paths

SAGE example

- Example Program

```
void top(char input[4]) {
       int cnt = 0;
       if (input[0] == 'b') cnt++;
       if (input[1] == 'a') cnt++;
       if (input[2] == 'd') cnt++;
       if (input[3] == '!') cnt++;
       if (cnt >= 3) crash();
   }
```

What would be interesting test cases? How could you find them?

[Poll’18]

SAGE example

- Example Program

```
void top(char input[4]) {

       int cnt = 0;
       if (input[0] == 'b') cnt++;
       if (input[1] == 'a') cnt++;
       if (input[2] == 'd') cnt++;
       if (input[3] == '!') cnt++;
       if (cnt >= 3) crash();
   }
```

path contraints:

$i_0 \neq$ 'b'

$i_1 \neq$ 'a'

$i_2 \neq$ 'd'

$i_3 \neq$ '!'

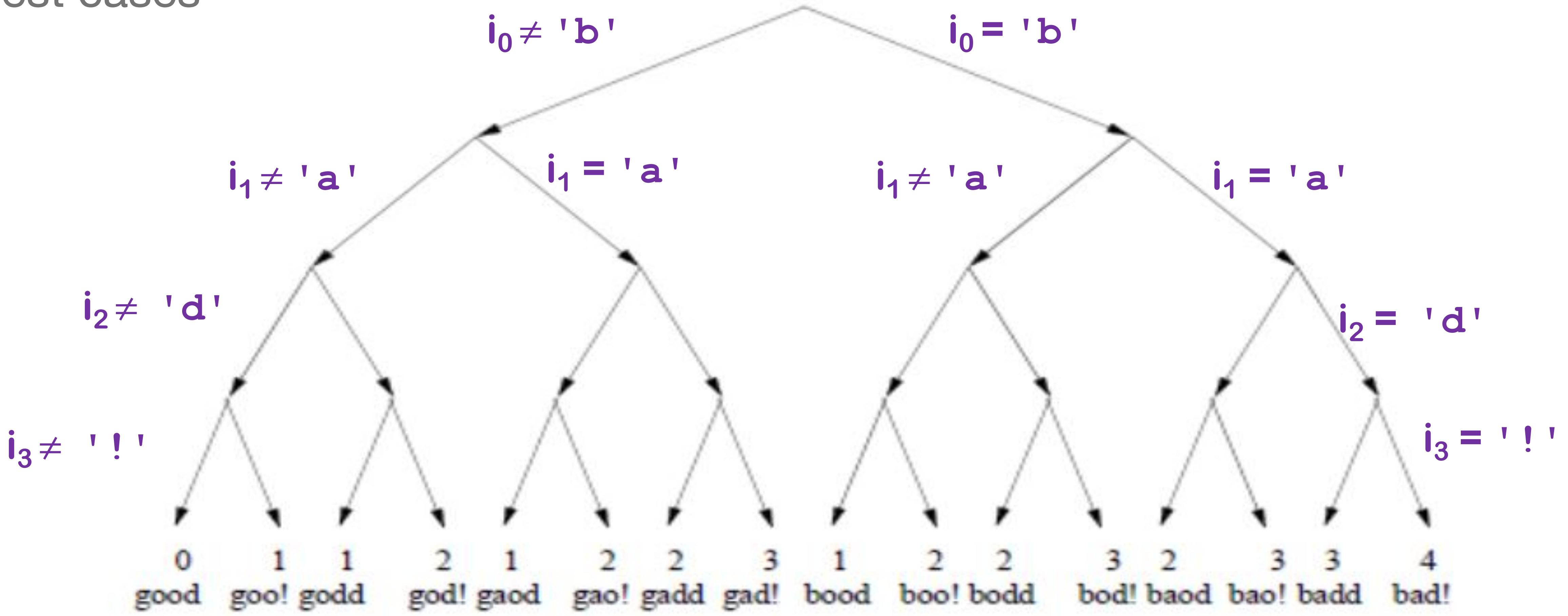
SAGE executes the code for some concrete input, say 'good'

It then collects path constraints for an arbitrary symbolic input, say i0 i1 i2 i3

[Poll'18]

Search space for interesting inputs

- Based on this one execution, combining all these constraints now yields 16 test cases



- Note: the initial execution with the input 'good' was not very interesting, but these others are

SAGE success

- SAGE proved successful at uncovering security bugs:
  - Microsoft Security Bulletin MS07-017 aka CVE-2007-0038: Critical
    - Vulnerabilities in GDI Could Allow Remote Code Execution
    - Stack-based buffer overflow in the animated cursor code in Microsoft Windows 2000 SP4 through Vista allows remote attackers to execute arbitrary code or cause a denial of service (persistent reboot) via a large length value in the second (or later) anih block of a RIFF .ANI, cur, or .ico file, which results in memory corruption when processing cursors, animated cursors, and icons
  - Security vulnerablity in parsing the ANI-format. SAGE generated a well-formed ANI file triggering the bug, without knowing the ANI format.
- First experiments also found bugs in handling a compressed file format, media file formats, and generated 43 test cases to crash Office 2007

[Poll'18]

Evolutionary Fuzzing with afl (American Fuzzy Lop)

Evolutionary Fuzzing with afl

- Downside of generation-based fuzzing:
    - lots of work work to write code to do the fuzzing, even if you use tools to generate this code based on some grammar
- Downside of mutation-based fuzzer:
    - chance that random changes in inputs hits interesting cases is small
- afl (American Fuzzy Lop) takes an evolutionary approach to learn interesting mutations based on measuring code coverage
    - basic idea: if a mutation of the input triggers a new execution path through the code, then it is an interesting mutation & it is kept; if not, the mutation is discarded.
    - by trying random mutations of the input and observering their effect on code coverage, afl can learn what interesting inputs are

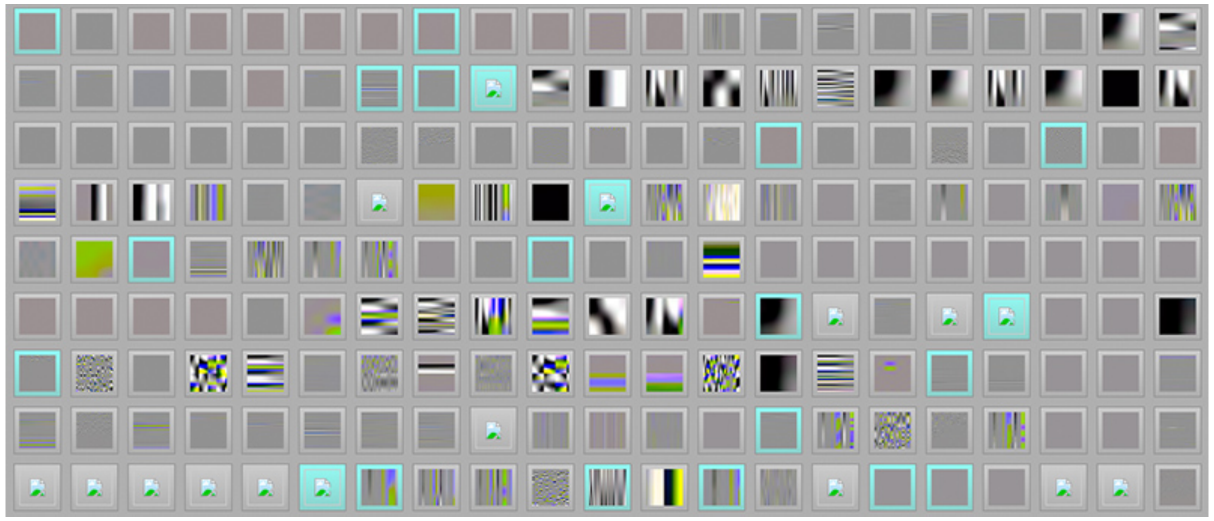
afl [http://lcamtuf.coredump.cx/afl]

- Supports programs written in C/C++/Objective C and variants for Python/Go/Rust/OCaml
- Code instrumented to observe execution paths:
  - if source code is available, by using modified compiler
  - if source code is not available, by running code in an emulator
- Code coverage represented as a 64KB bitmap, where control flow jumps are mapped to changes in this bitmap
  - different executions could result in the same bitmap, but chance is small
- Mutation strategies include: bit flips, incrementing/decrementing integers, using pre-defined interesting integer values (eg. 0, -1, MAX_INT,....), deleting/combining/zeroing input blocks, ...
- The fuzzer forks the SUT to quickly process lots of test cases
- Big win: no need to specify the input format!

[Poll'18]

Cool example: learning the JPG file format

- Fuzzing a program that expects a JPG as input, starting with 'hello world' as initial test input, afl can learn to produce legal JPG files
  - along the way producing/discovering error messages such as
    - Not a JPEG file: starts with 0x68 0x65
    - Not a JPEG file: starts with 0xff 0x65
    - Invalid JPEG file structure: two SOI markers
    - Quantization table 0x0e was not defined
    - Premature end of JPEG file
- and then JPGs like



- [Source http://lcamtuf.blogspot.nl/2014/11/pulling-jpegs-out-of-thin-air.html]

Vulnerabilities found with afl

IJG jpeg 1	libjpeg-turbo 1 2	libpng 1
libtiff 1 2 3 4 5	mozjpeg 1	PHP 1 2 3 4 5
Mozilla Firefox 1 2 3 4	Internet Explorer 1 2 3 4	Apple Safari 1
Adobe Flash / PCRE 1 2 3 4	sqlite 1 2 3 4...	OpenSSL 1 2 3 4 5 6
LibreOffice 1 2 3 4	poppler 1	freetype 1 2
GnuTLS 1	GnuPG 1 2 3 4	OpenSSH 1 2 3
PuTTY 1 2	ntpd 1 2	nginx 1 2 3
bash (post-Shellshock) 1 2	tcpdump 1 2 3 4 5 6 7 8 9	JavaScriptCore 1 2 3
pdfium 1 2	ffmpeg 1 2 3 4 5	libmatroska 1
libarchive 1 2 3 4 5 6 ...	wireshark 1 2 3	ImageMagick 1 2 3 4 5 7 8 9 ...
BIND 1 2 3 ...	QEMU 1 2	lcms 1
Oracle BerkeleyDB 1 2	Android / libstagefright 1 2	iOS / ImageIO 1
FLAC audio library 1 2	libsndfile 1 2 3 4	less / lesspipe 1 2 3
strings (+ related tools) 1		

VUzzer: Application-aware Evolutionary Fuzzing,
Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar,
Cristiano Giuffrida, Herbert Bos, NDSS'17

Vuzzer

Your takeaway message of today

Smart fuzzing without symbolic execution

Extract application features for meaningful mutation

VUzzer 30k inputs: 403 crashes

AFL 30,000K inputs: 238 crashes

[Veen'17]

AFL

```
….
read( fd, buf, size);
if (buf [5] == 0xDB && bud [4] == 0xFF)
    // interesting code here
else
    pr_exit (“invalid file”)
```

AFL will run for hours on this

- Has to figure out that offset 4 and 5 are of interest (where)
- Needs to guess 0xFFDB (what)

[Veen’17]

More Problems

- Handling ‘Complex’ code structure
  - Magic Bytes
    - certain values should be places at pre-determined offsets
  - Deeper Execution
    - Many inputs will end up in less interesting error-handling code
  - (multibyte) Markers
    - Not at fixed offsets: if (strstr (&buf, “MAZE”)) ….

[Veen’17]

Vuzzer

Our mutation-, coverage-based, greybox fuzzer

[Veen’17]

VUzzer

- where to Mutate, what to insert
    - Evolutionary fuzzing
        - Mutate/select most promising paths
    - Magic byte detection
        - Find possible magic byte values to reach deeper into the binary
    - (limited) Input type detection
        - Aid mutation by detecting input bytes of certain types (integers)

- Avoid non-scalable techniques
    - No symbolic execution
    - Limited use of Dynamic Taint Analysis

[Veen’17]

Feature Extraction

- Data-flow features
  - Information about relationship between input data and program computations
  - Extracted using static analysis / dynamic taint analysis
  - example: *cmp* instruction on x86
    - magic values: immediate operands for *cmp* (static analysis)
    - Offsets: which input bytes are compared against? (taint analysis)
  - example: *lea* instructions
    - integer types: is *index* operand tainted?

[Veen'17]

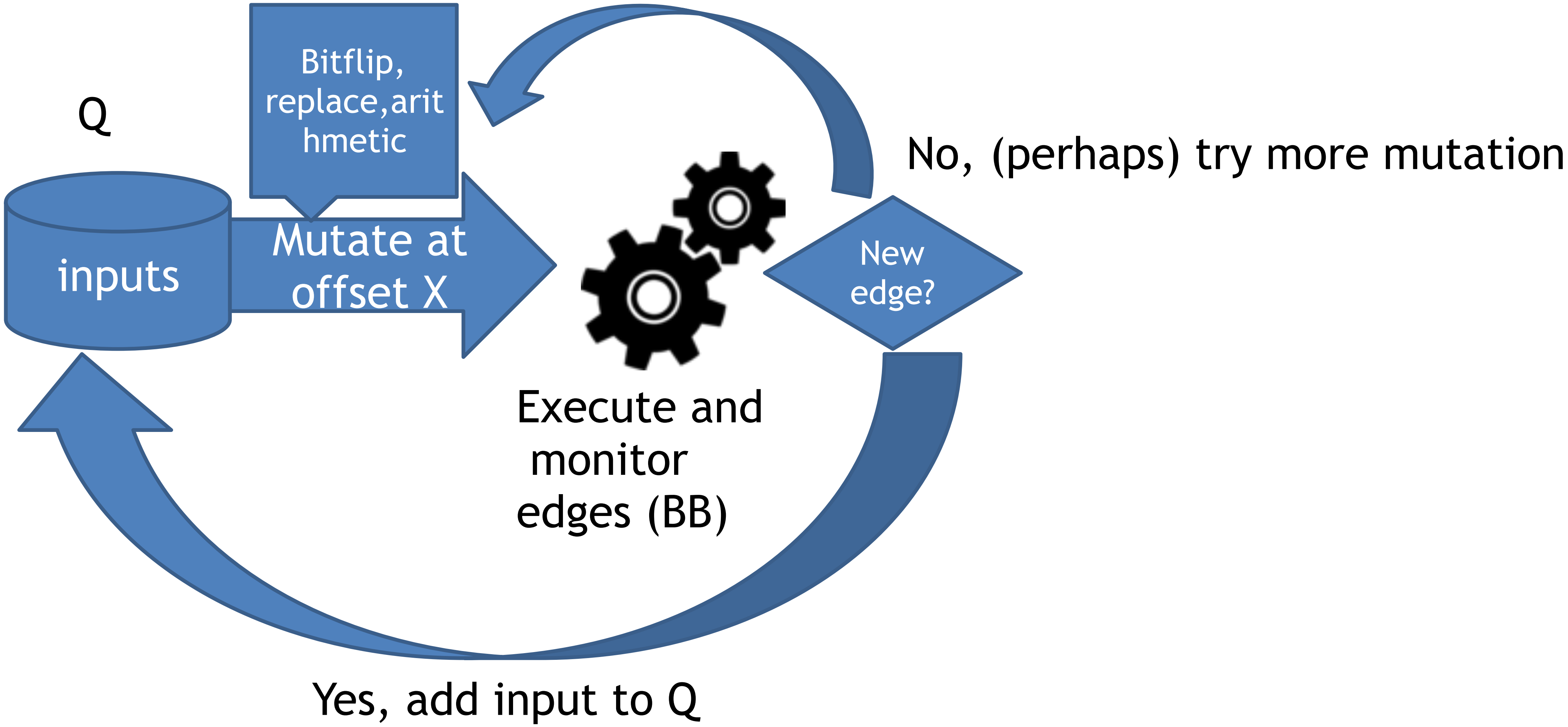
Feature extraction

- Control-flow features
  - Information about the importance of certain executed paths
  - Identifying error-handling blocks (heuristics based)
  - Rank basic blocks to prioritize hard-to-reach code
    - Each basic block gets a weight depending on how deep it is nested
    - Error-handling blocks get a negative weight

[Veen’17]

Evolving Our Solution- VUzzer

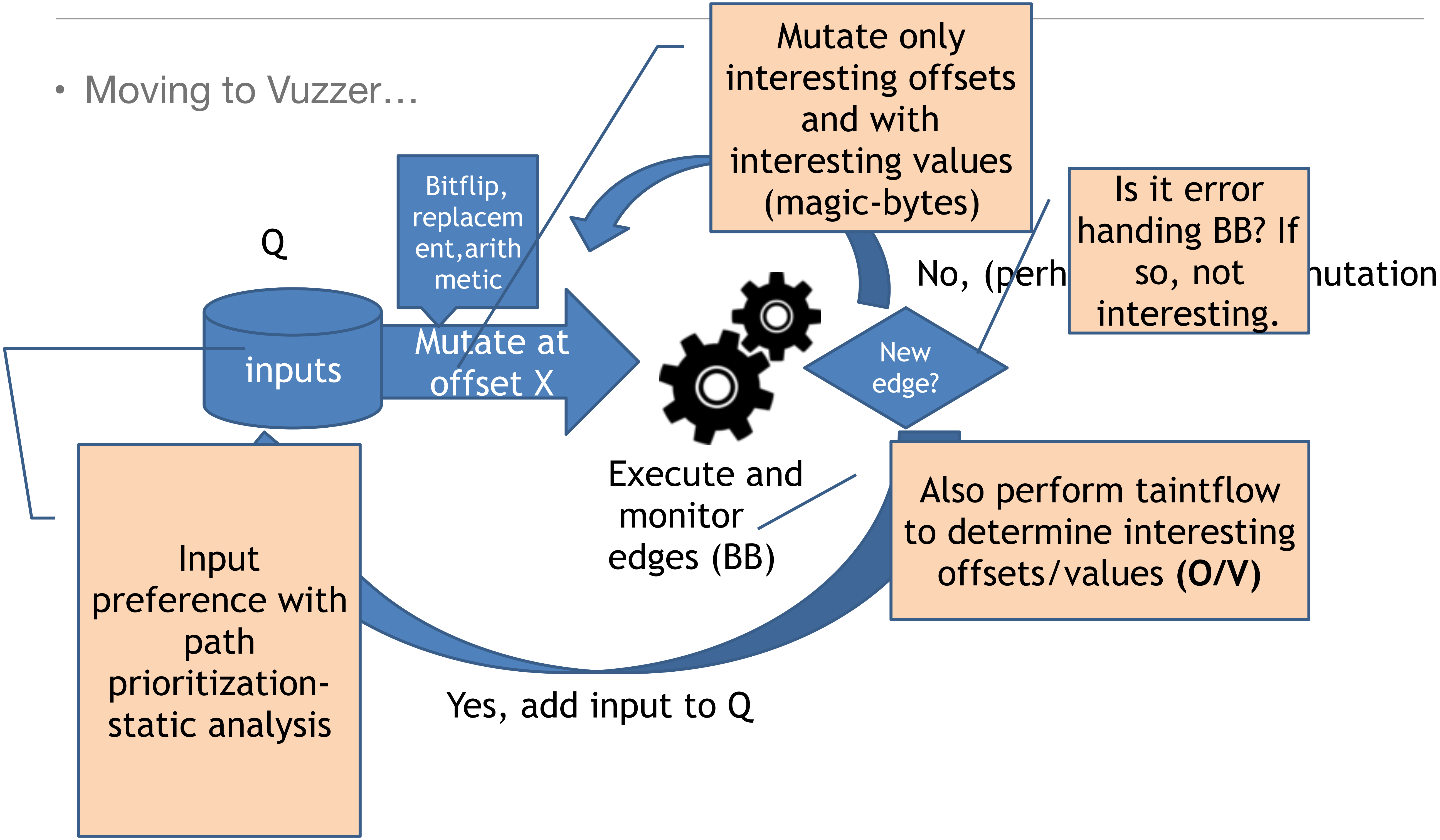
- Lets start with something we know- AFL


Q
inputs
Bitflip, replace,arithmetic
Mutate at offset X
Execute and monitor edges (BB)
New edge?
No, (perhaps) try more mutation
Yes, add input to Q

[Giuffrida'17]

Evolving Our Solution- VUzzer

- Moving to Vuzzer…



Mutate only interesting offsets and with interesting values (magic-bytes)

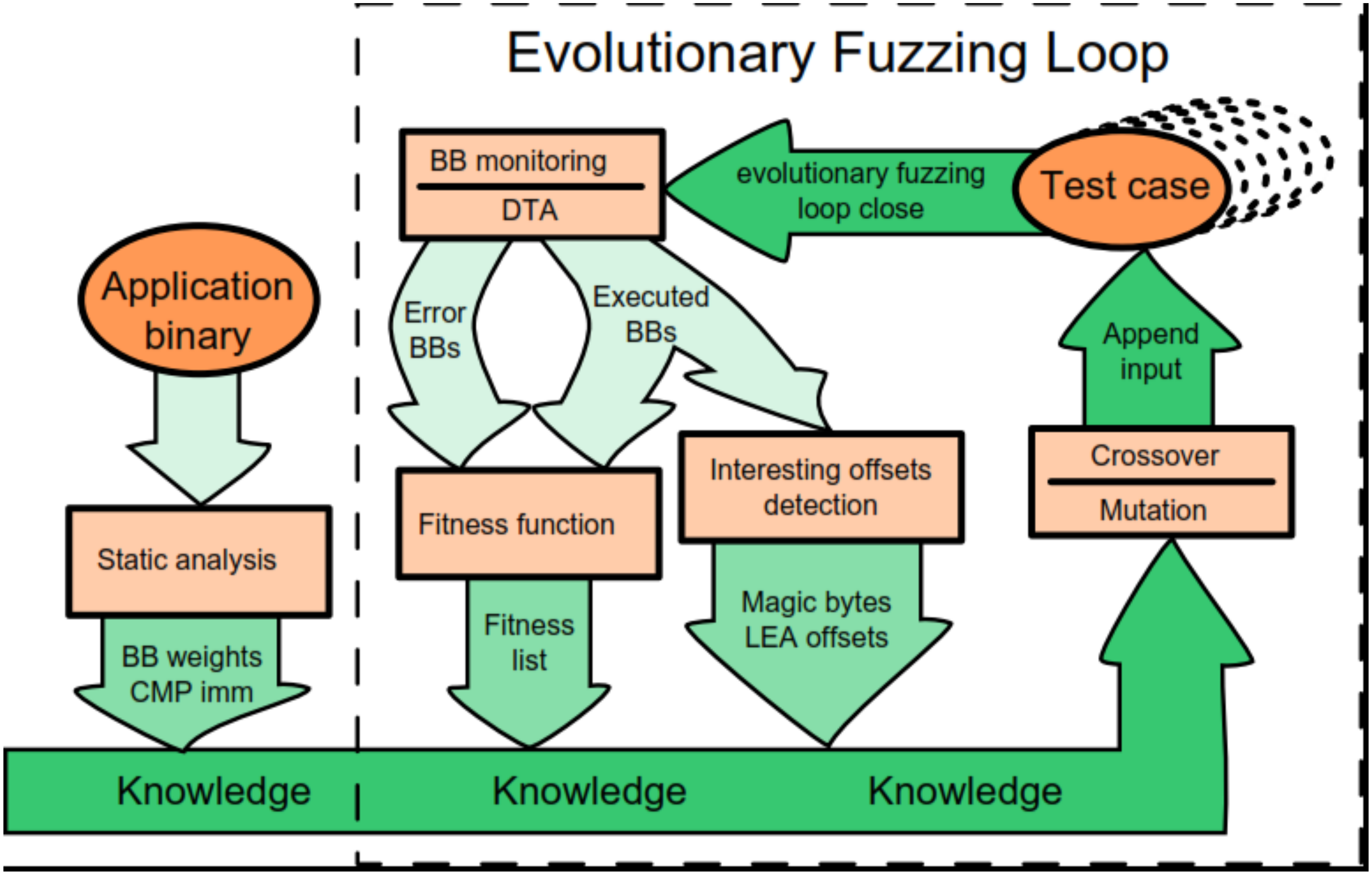
Is it error handing BB? If so, not interesting.

Also perform taintflow to determine interesting offsets/values **(O/V)**

Input preference with path prioritization- static analysis

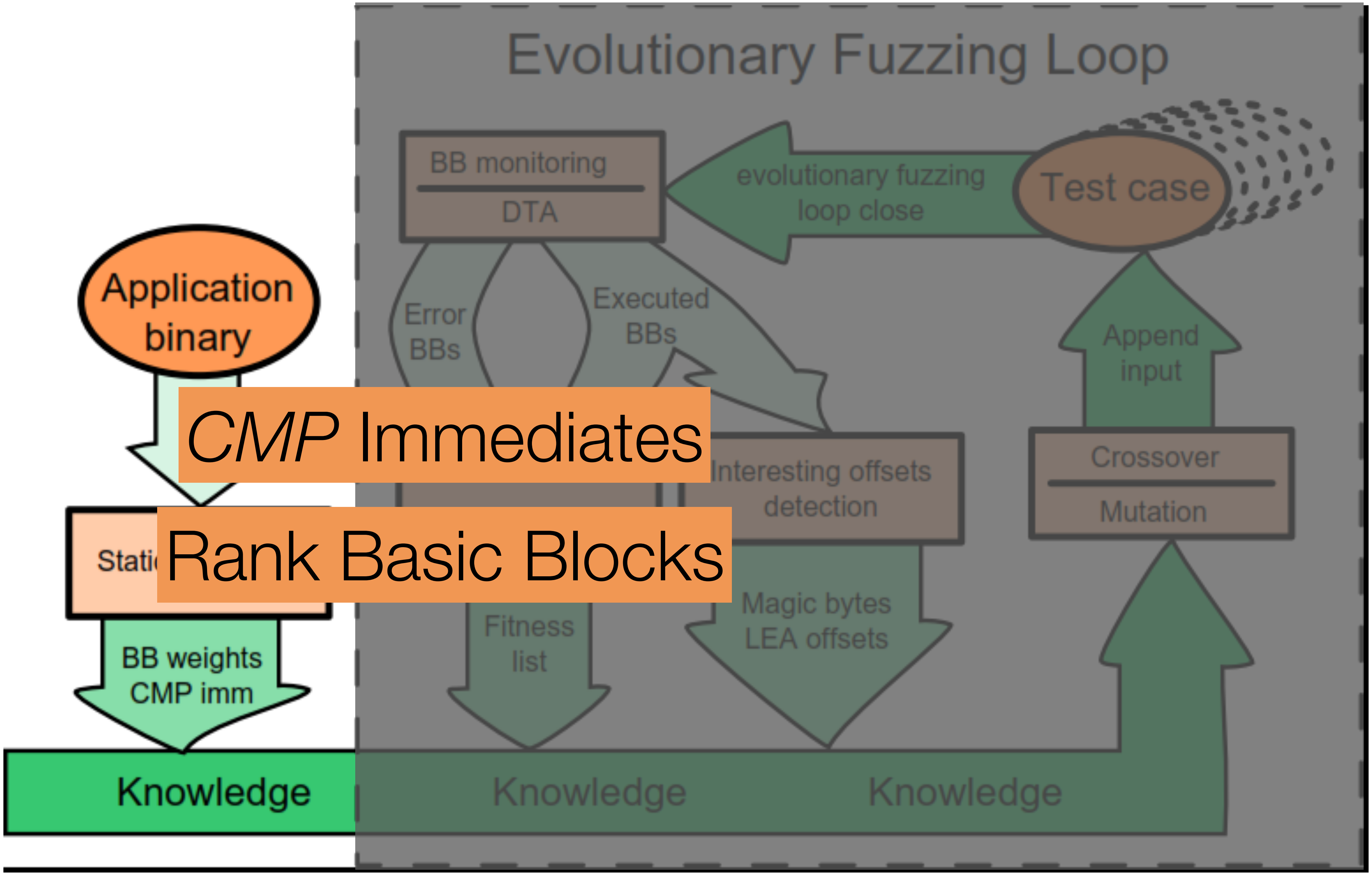
[Giuffrida'17]

Our Solution: VUzzer



[Veen'17]

Our Solution: VUzzer

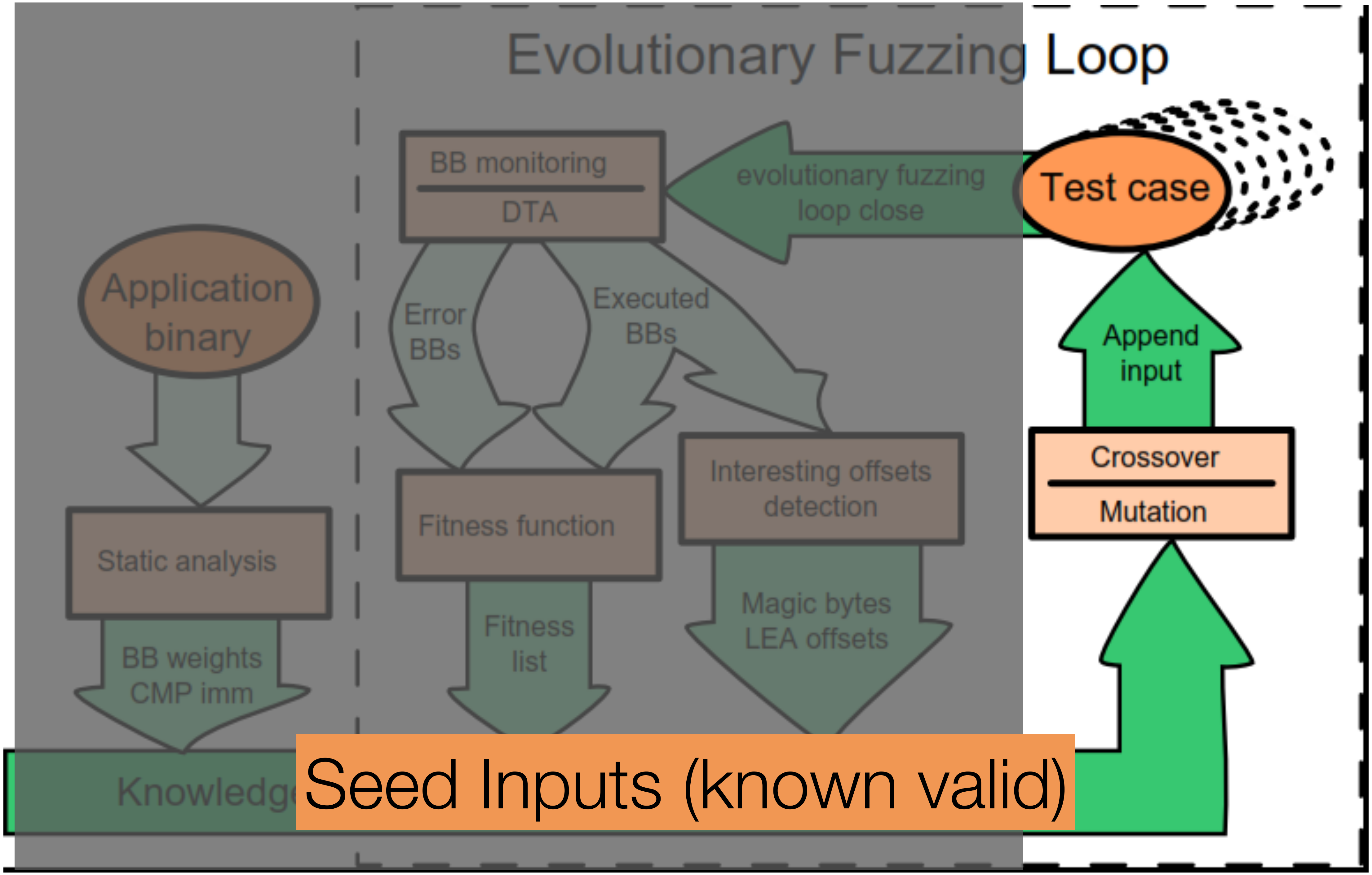


CMP Immediates

Rank Basic Blocks

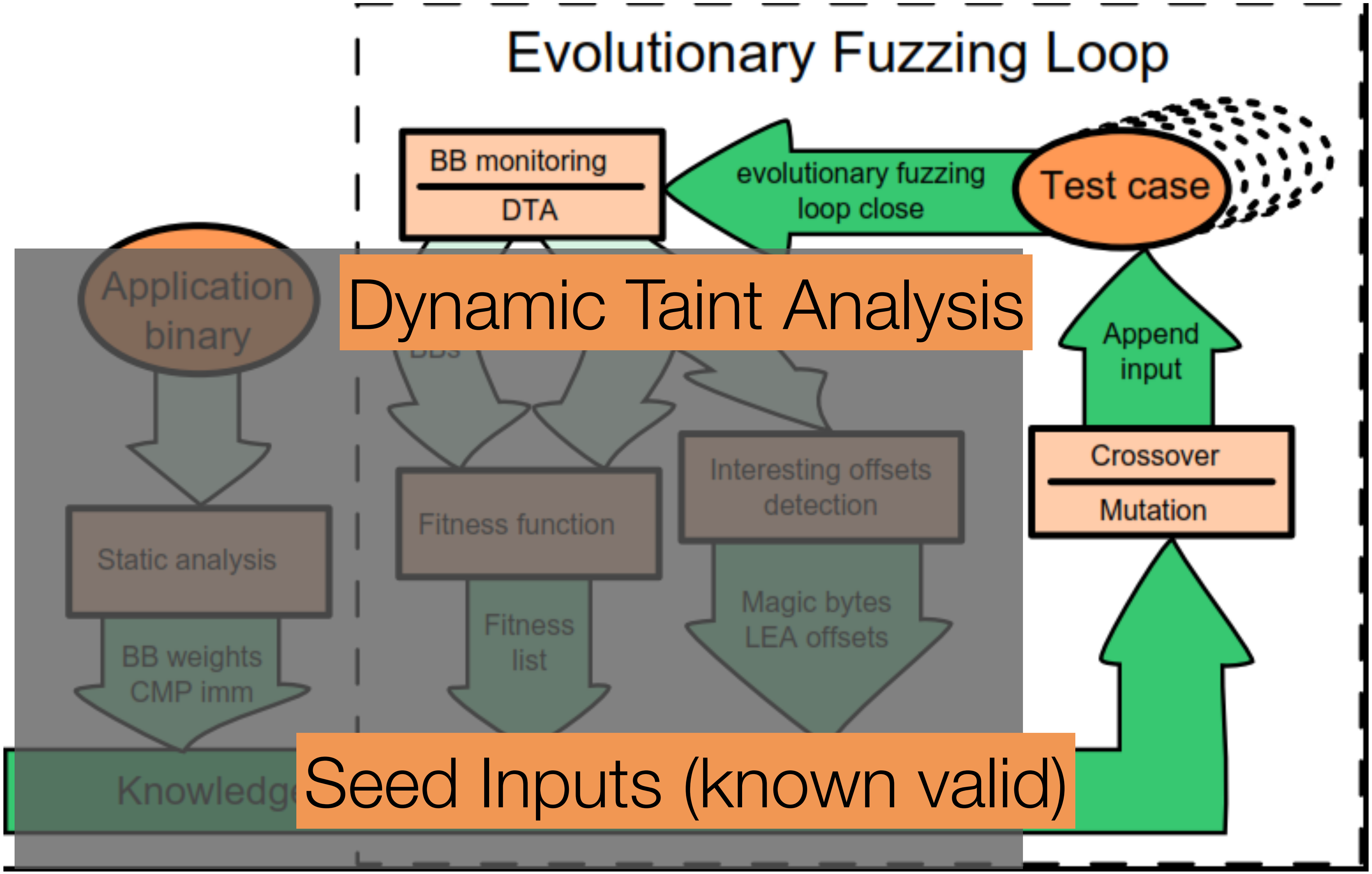
[Veen'17]

Our Solution: VUzzer



Seed Inputs (known valid)

Our Solution: VUzzer

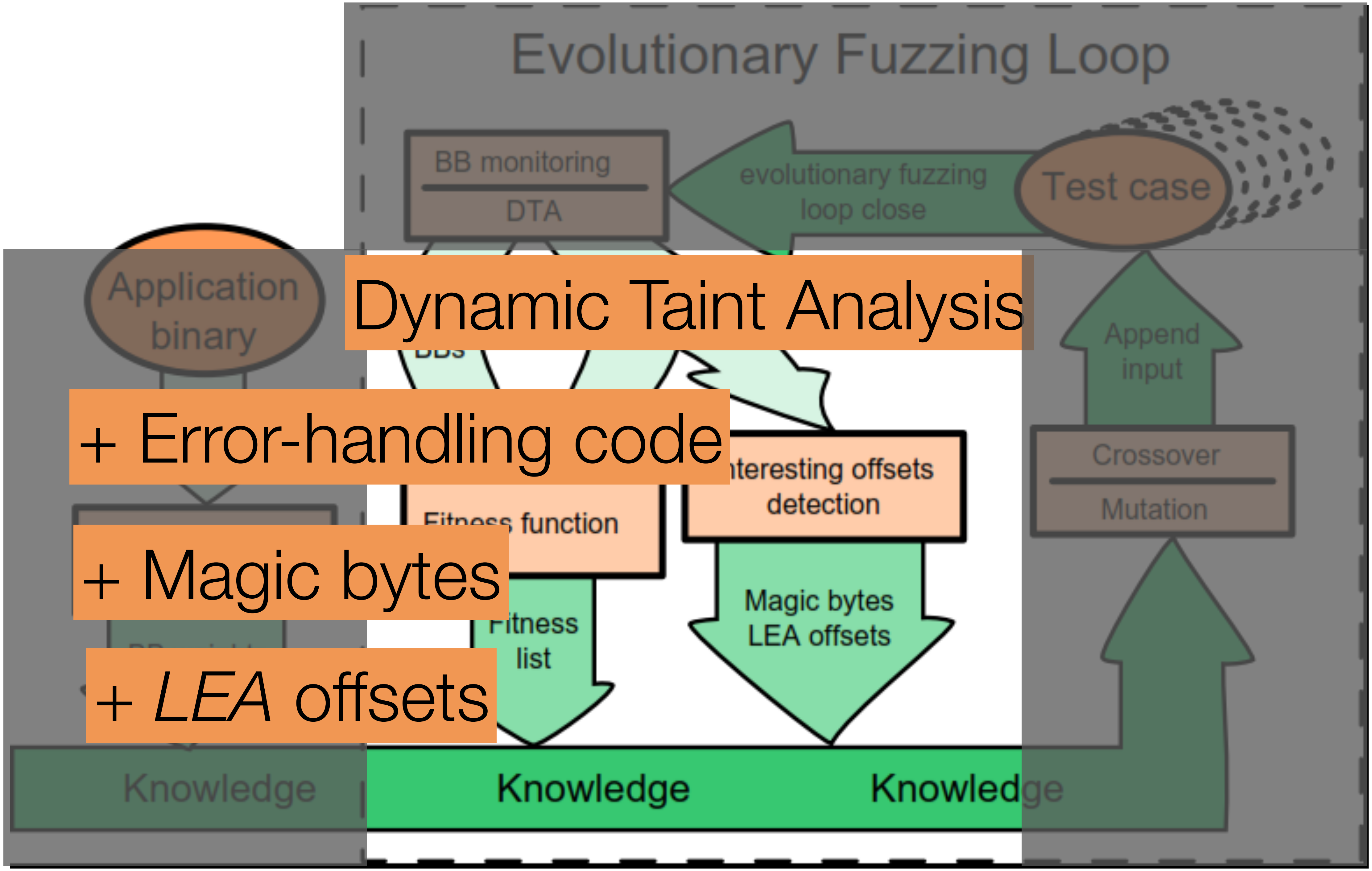


Dynamic Taint Analysis

Seed Inputs (known valid)

[Veen'17]

Our Solution: VUzzer



Dynamic Taint Analysis

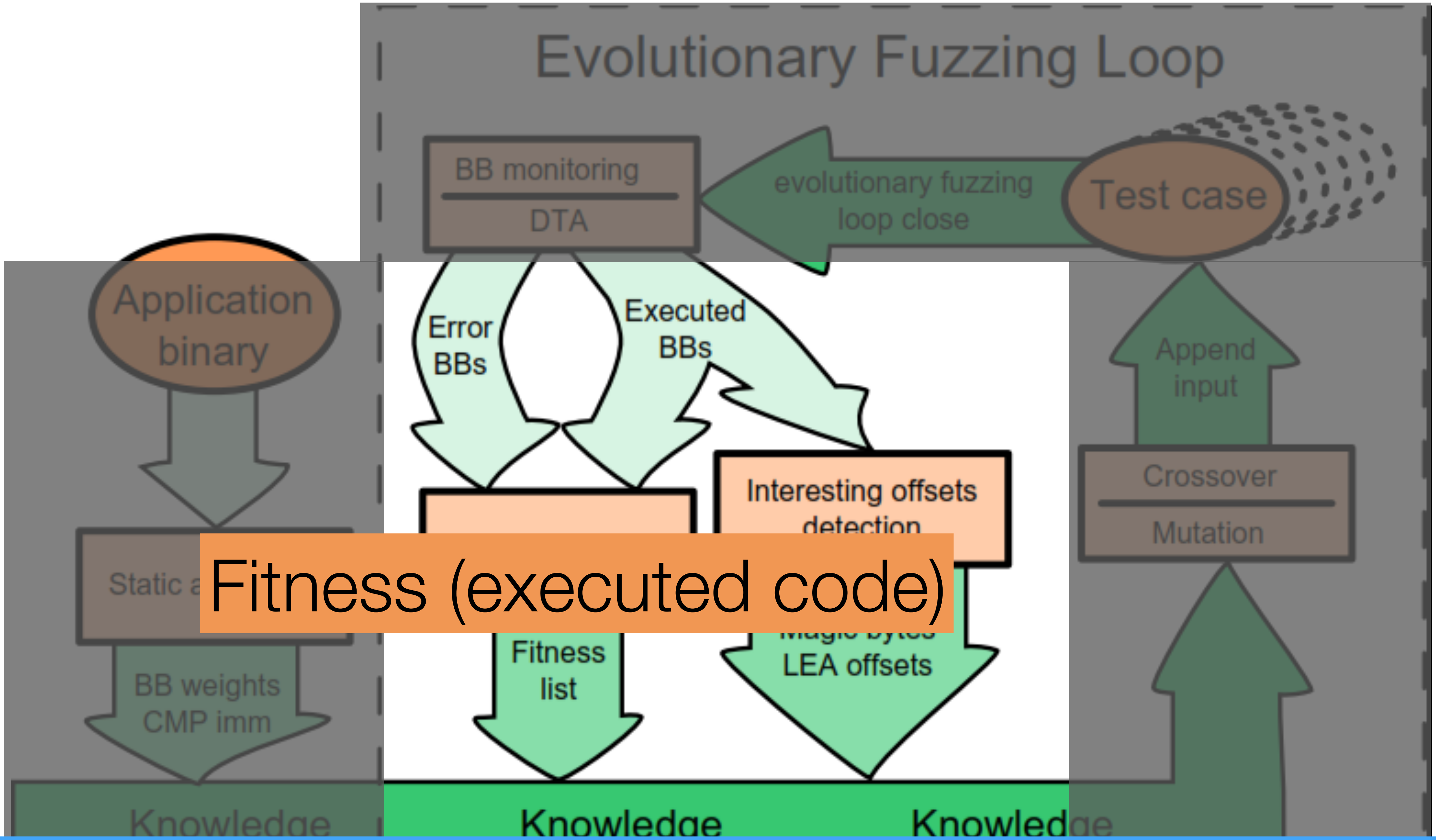
+ Error-handling code

+ Magic bytes

+ *LEA* offsets

[Veen’17]

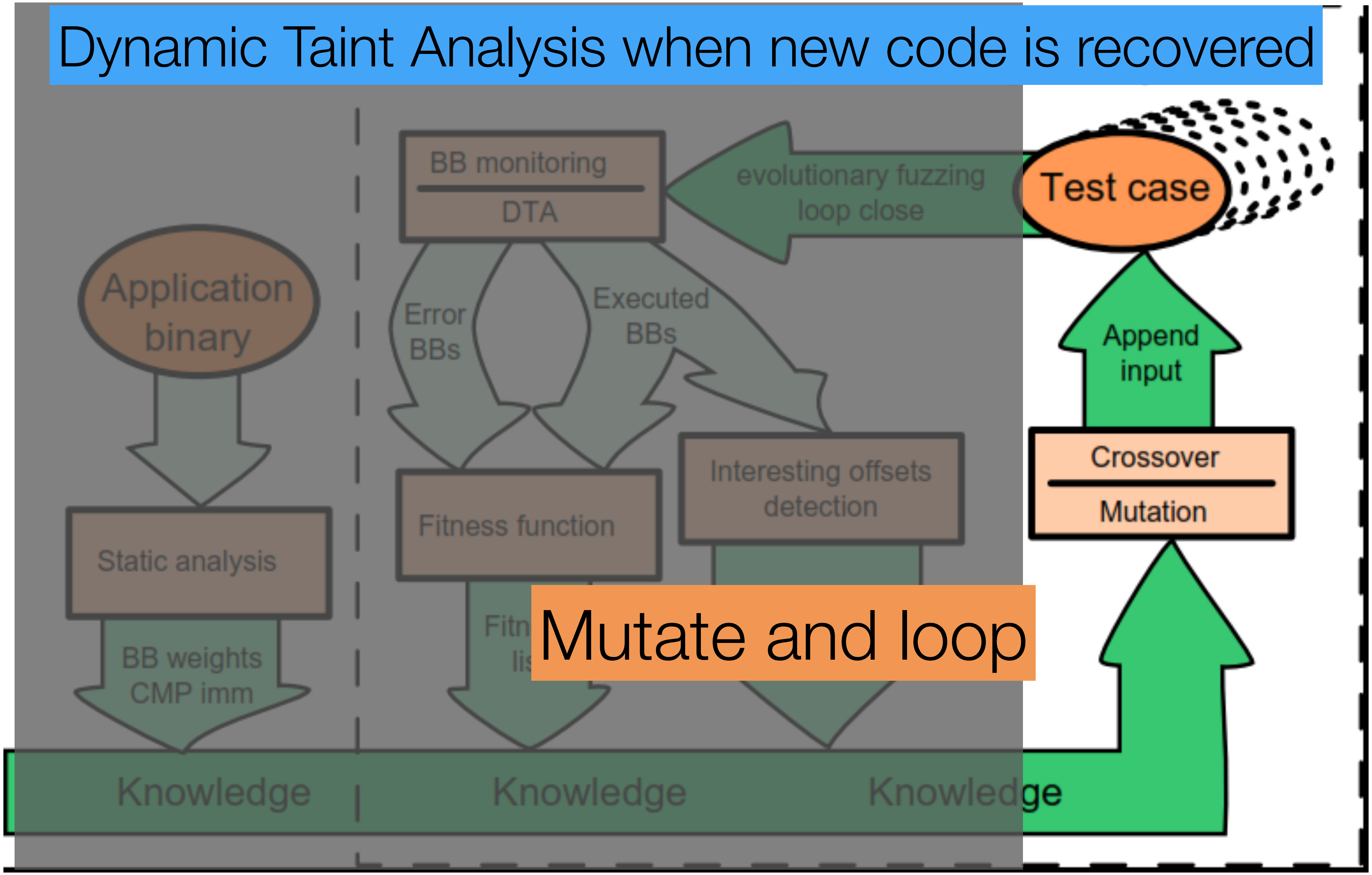
Our Solution: VUzzer



Fitness (executed code)

High scores for inputs that execute highly ranked basic blocks

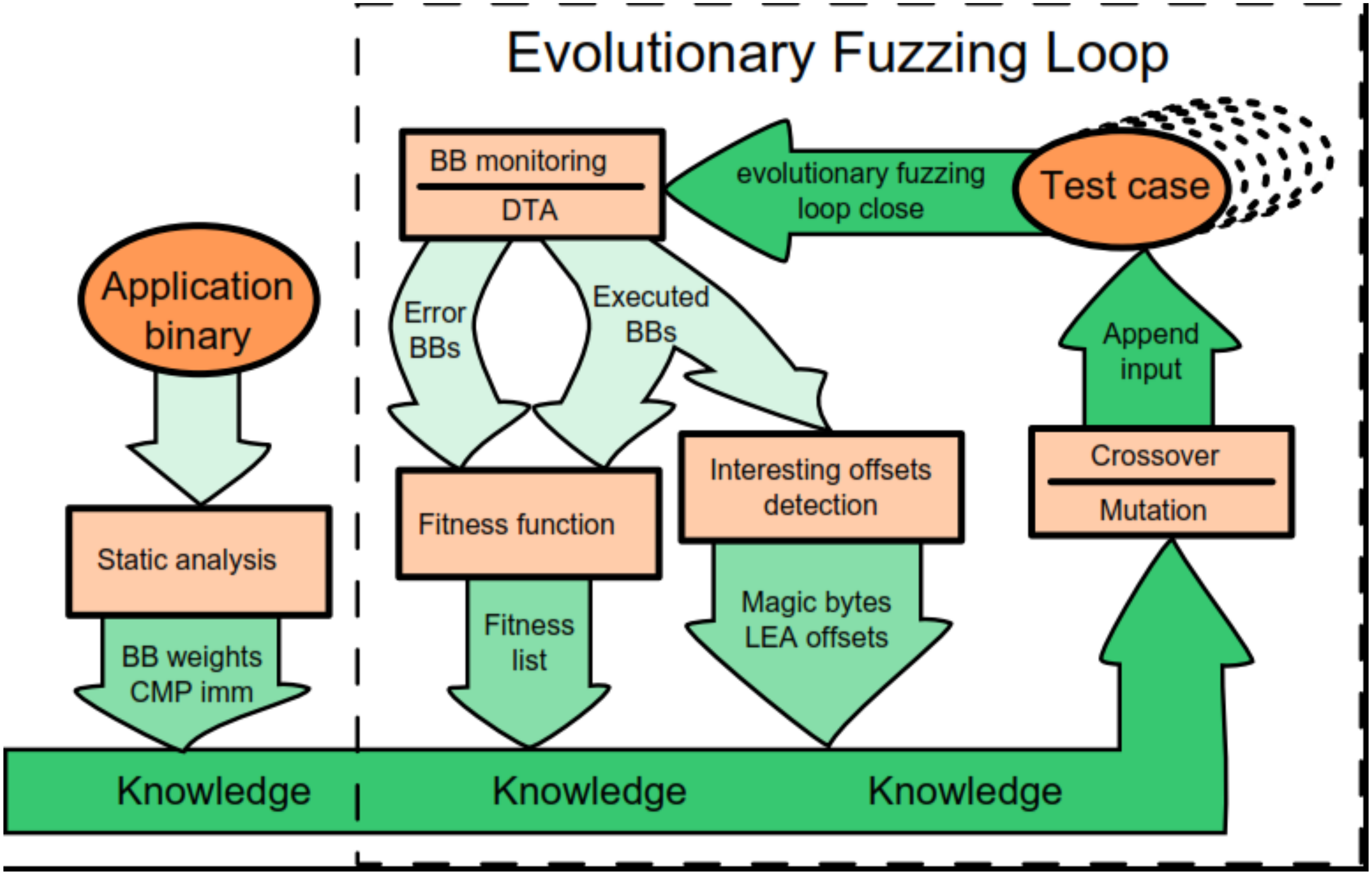
Our Solution: VUzzer



Dynamic Taint Analysis when new code is recovered

Mutate and loop

VUzzer



[Veen’17]

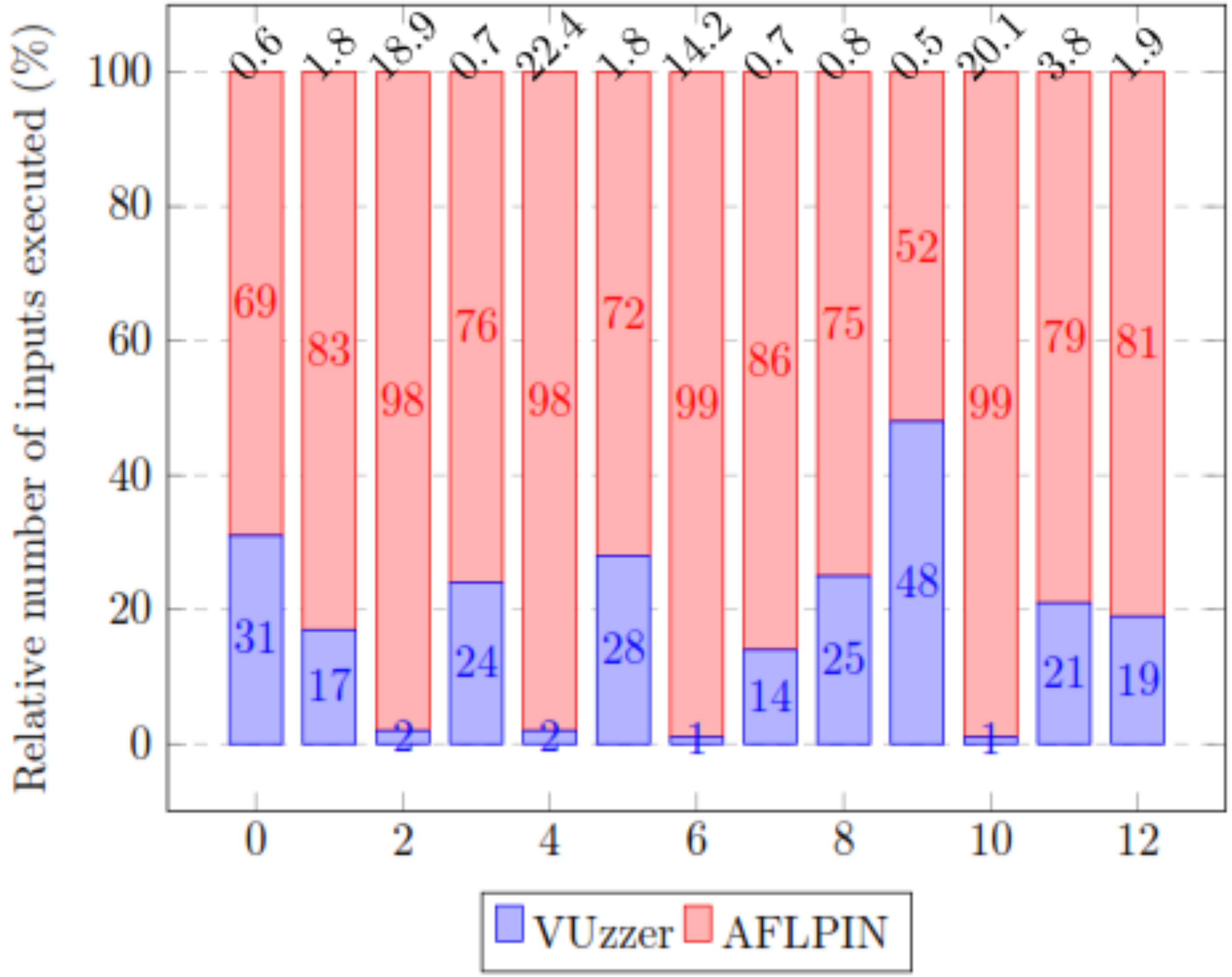
Evaluation

- DARPA CGC binaries
- Various applications with binary input format as used in other work (VA)
- A set of buggy binaries recently proposed in LAVA

[Giuffrida'17]

Evaluation

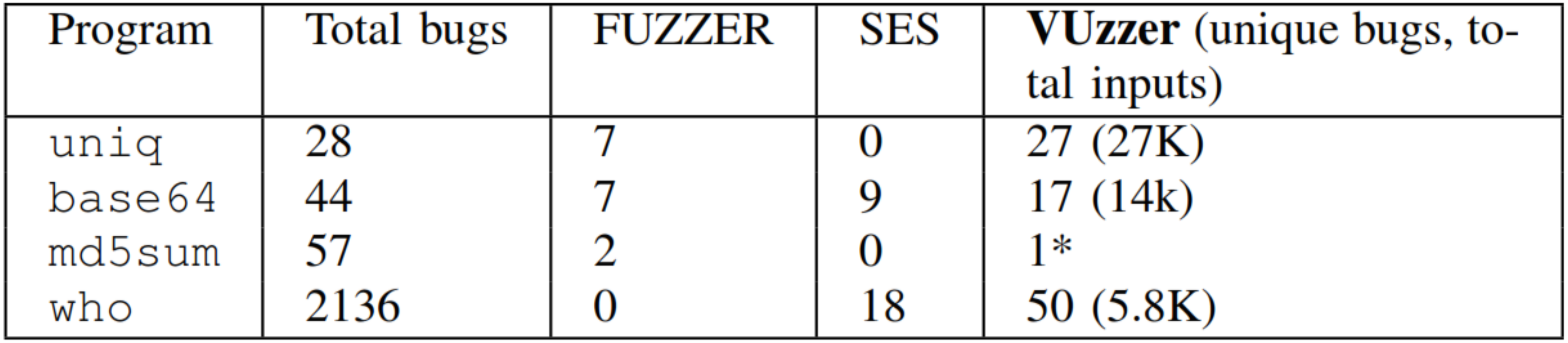
- DARPA CGC binaries
  - VUzzer / AFL
  - 13 binaries, 6 hours per binary
  - Far fewer VUzzer inputs



[Veen'17]

Evaluation

- LAVA-M Dataset
  - LAVA: inject hard-to-reach faults to evaluate buzzers



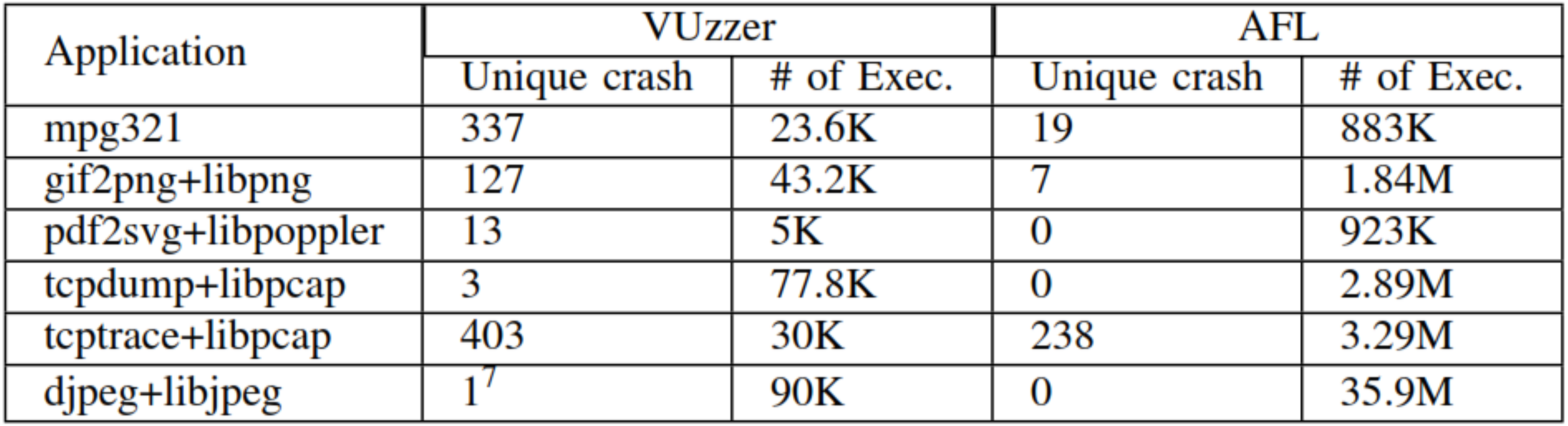
Program	Total bugs	FUZZER	SES	**VUzzer** (unique bugs, total inputs)
`uniq`	28	7	0	27 (27K)
`base64`	44	7	9	17 (14k)
`md5sum`	57	2	0	1*
`who`	2136	0	18	50 (5.8K)

- VUzzer hits significantly more bugs that
  - FUZZER: coverage based
  - SES: symbolic execution / SAT-based

[Veen’17]

Evaluation

- Various Applications
  - Comparison against AFL on vanilla Ununtu 14.04



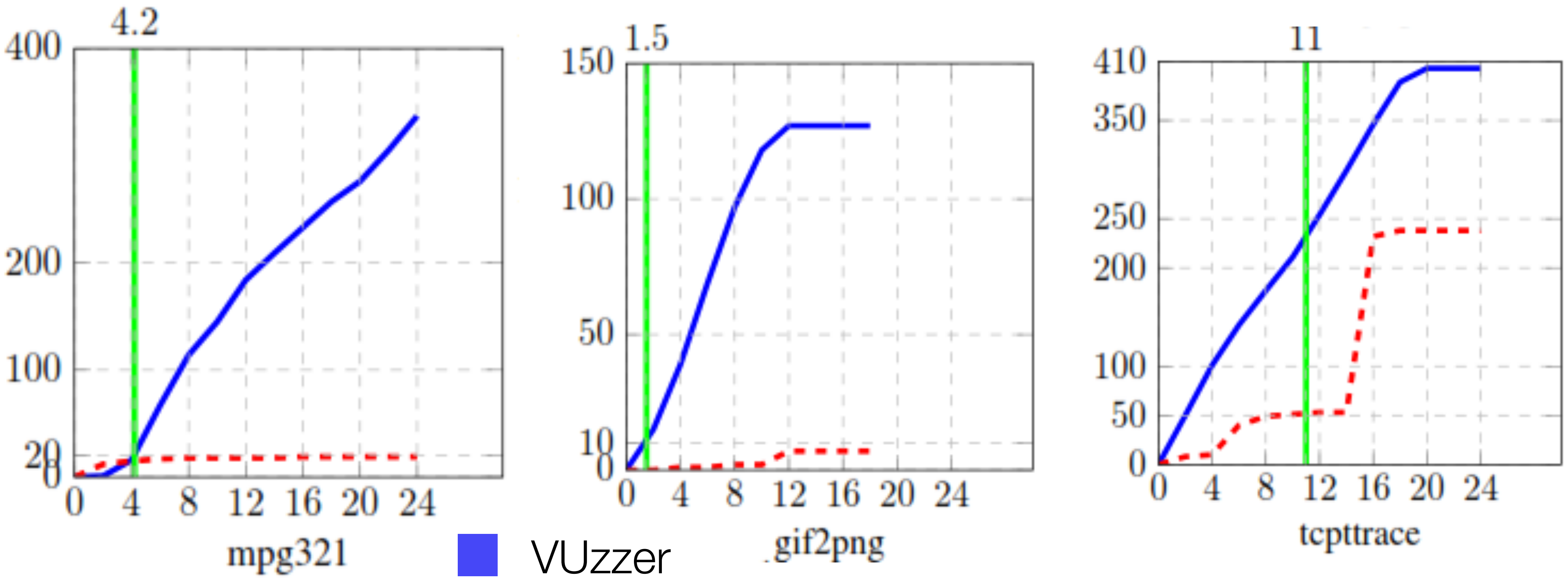
Application	VUzzer		AFL	
	Unique crash	# of Exec.	Unique crash	# of Exec.
mpg321	337	23.6K	19	883K
gif2png+libpng	127	43.2K	7	1.84M
pdf2svg+libpoppler	13	5K	0	923K
tcpdump+libpcap	3	77.8K	0	2.89M
tcptrace+libpcap	403	30K	238	3.29M
djpeg+libjpeg	1[7]	90K	0	35.9M

  - More bugs with fewer inputs

[Veen’17]

Evaluation

- Various applications
  - Comparison against AFL on vanilla Ununtu 14.04
  - Crash faster
  - Consistent progress



🟦 VUzzer

🟥 AFL

🟩 Time taken by VUzzer to find all crashed found by AFL

[Veen’17]

Conclusion

- VUzzer
  - Novel fuzzing technique based on evolutionary approach
  - Application-aware buzzer by exploiting data-flow and control-flow features
  - Prioritize hard-to-reach code paths
  - Deprioritize error-handling code
  - Significantly more bugs with order of magnitude fewer inputs in less time

[Veen’17]

Acknowledgments/References

- [Sutton’06] Fuzzing - Brute Force Vulnerability Discovery, Michael Sutton, Recon 2006
- [Payer’19] The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes, Mathias Payer, blog post, April, 2019
- [Heasman’09] Fuzzing for Security Flaws, John Heasman, Stanford University, April 2009
- [Poll’18] Software Security, Autumn, Erik Poll, Radboud University Nijmegen, Autumn 2018
- [Mitchell'18] CS155 Computer and Network Security, John Mitchell, Stanford University, Spring 2018
- [Veen’17] VUzzer: Application-aware Evolutionary Fuzzing, Victor van der Even, Slides NDSS 2017
- [Giuffrida’17] VUzzer: Application-aware Evolutionary Fuzzing, Cristiano Giuffrida, Slides from VUSec, 2017