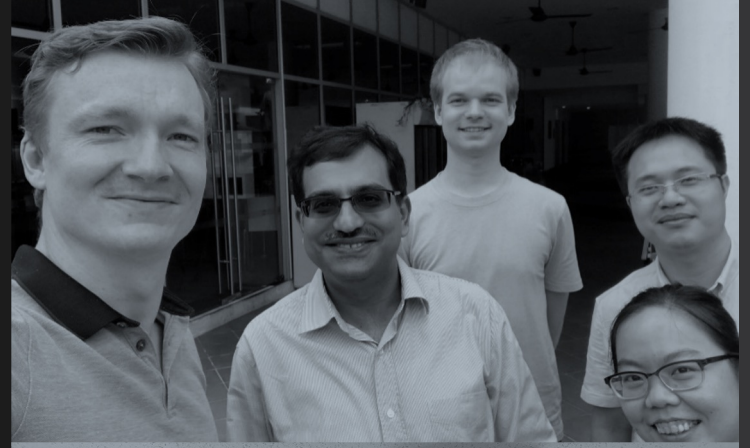
Fuzz Testing

Abhik Roychoudhury

National University of Singapore

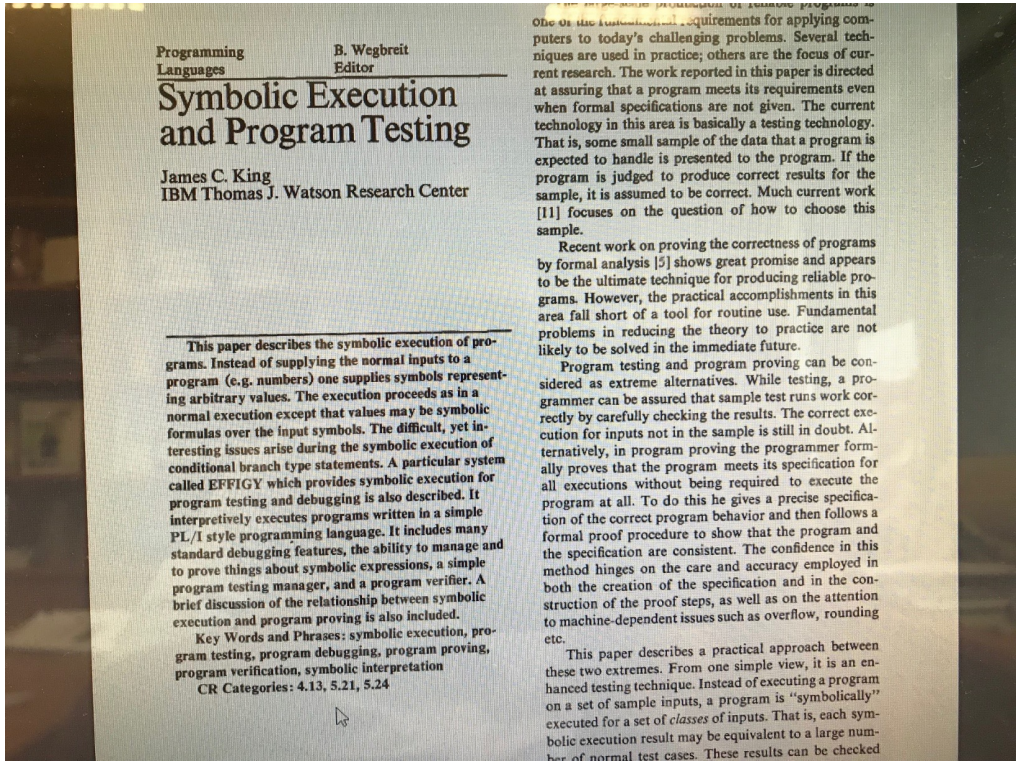


Background / interests ~ 2013-14

"*Program testing and program proving can be considered as extreme alternatives. ….*

This paper describes a practical approach between these two extremes …

Each symbolic execution result may be equivalent to a large number of normal tests"



Programming Languages — B. Wegbreit Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

This paper describes the symbolic execution of programs. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. The difficult, yet interesting issues arise during the symbolic execution of conditional branch type statements. A particular system called EFFIGY which provides symbolic execution for program testing and debugging is also described. It interpretively executes programs written in a simple PL/I style programming language. It includes many standard debugging features, the ability to manage and to prove things about symbolic expressions, a simple program testing manager, and a program verifier. A brief discussion of the relationship between symbolic execution and program proving is also included.

Key Words and Phrases: symbolic execution, program testing, program debugging, program proving, program verification, symbolic interpretation
CR Categories: 4.13, 5.21, 5.24

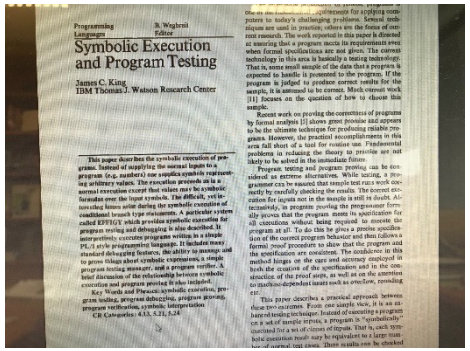
…quirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

Recent work on proving the correctness of programs by formal analysis [5] shows great promise and appears to be the ultimate technique for producing reliable programs. However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved in the immediate future.

Program testing and program proving can be considered as extreme alternatives. While testing, a programmer can be assured that sample test runs work correctly by carefully checking the results. The correct execution for inputs not in the sample is still in doubt. Alternatively, in program proving the programmer formally proves that the program meets its specification for all executions without being required to execute the program at all. To do this he gives a precise specification of the correct program behavior and then follows a formal proof procedure to show that the program and the specification are consistent. The confidence in this method hinges on the care and accuracy employed in both the creation of the specification and in the construction of the proof steps, as well as on the attention to machine-dependent issues such as overflow, rounding etc.

This paper describes a practical approach between these two extremes. From one simple view, it is an enhanced testing technique. Instead of executing a program on a set of sample inputs, a program is "symbolically" executed for a set of *classes* of inputs. That is, each symbolic execution result may be equivalent to a large number of normal test cases. These results can be checked

1976 paper on Symbolic Execution



Programming Languages — B. Wegbreit, Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

Symbolic Execution

```
SEARCH( A, L, U, X, found, j){

int j, found = 0;
while (L  <= U  && found == 0){
        j = (L+U)/2;
        if (X == A[j]){  found  = 1;}
        else if (X < A[j]){ U = j -1; }
        else{ L = j +1; }
}
if (found == 0){ j =  L - 1;}

}
```

SEARCH(A, 1, 5, X, found, j)

X == A[3]	found == 1	j == 3
X == A[1] && X < A[3]	found == 1	j == 1
X < A[1] && X <A[3]	found == 0	j == 0
X = A[2] && X > A[1] && X <A[3]	found == 1	j == 2

….

USABILITY rather than SCALABILITY

Systematic Testing ?
Comprehension??
Verification ???

Fuzz testing

Fuzz testing is a simple technique for feeding random input to applications to expose bugs and vulnerabilities. The approach has three characteristics.

- *The input is **random**. We do not use any model of program behavior, application type, or system description. This is sometimes called **black box testing**.*
- *The reliability criteria is **simple**: if the application **crashes or hangs**, it is considered to fail the test, otherwise it passes. Note that the application does not have to respond in a sensible manner to the input, and it can even quietly exit.*
- *As a result of the first two characteristics, fuzz testing can be **automated** to a high degree and results can be compared across applications, operating systems, and vendors.*

TRUE STORY…

Part of the story starts with teaching in 2015.

NUS decided to start a Bachelors in Information Security.

Fuzzing was an established tech., but I had little exposure.

Lot of work in 2014-15 on <u>using</u> fuzzing to find vulnerabilities.

TRUE STORY…

- May 4, 2015
  - Abhik was preparing lecture notes on fuzzing for the to-be-newly-offered CS4239 Software Security course at National University of Singapore (taught Aug –Dec 2015).
  - 11:00 AM – finished deciding on structure and trying to decide on a motivating example for fuzzing to interest the students, there are so many of them!
  - 11:11 AM – I get email update about a latest incident – an integer overflow in Boeing – a classic case where an automated method for sending out mal-formed or boundary inputs can reveal errors.

Little or no research on developing newer fuzzing technologies at that time.
AFL existed as a tool from Google.

No understanding of why it worked, when it worked
Got keen about getting <u>inside</u> fuzzers to improve the fuzzing algorithm!

Why fuzz – the true story

Boeing 787 Dreamliners contain a potentially catastrophic software bug

Beware of integer overflow-like bug in aircraft's electrical system, FAA warns.

by Dan Goodin - May 2, 2015 1:55am CST

A software vulnerability in Boeing's new 787 Dreamliner jet has the potential to cause pilots to lose control of the aircraft, possibly in mid-flight, Federal Aviation Administration officials warned airlines recently.

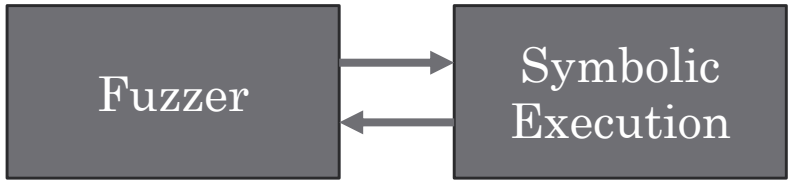
The bug—which is either a classic integer overflow or one very much resembling it—resides in one of the electrical systems responsible for generating power, according to memo the FAA issued last week. The vulnerability, which Boeing reported to the FAA, is triggered when a generator has been running continuously for a little more than eight months. As a result, FAA officials have adopted a new airworthiness directive (AD) that airlines will be required to follow, at least until the underlying flaw is fixed.

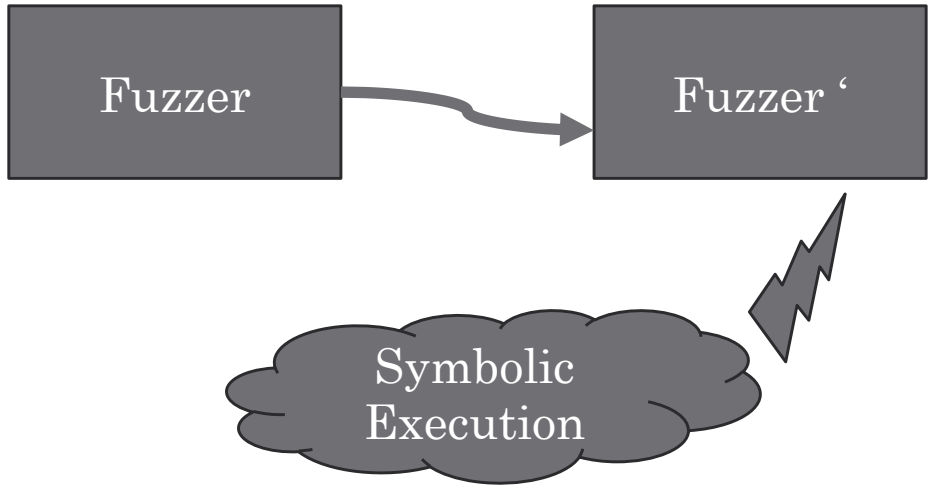
"This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode," the memo stated. "This condition is caused by a software counter internal to the GCUs that will overflow after 248 days of continuous power. We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane."

The memo went on to say that Dreamliners have four main GCUs associated with the engine mounted generators. If all of them were powered up at the same time, "after 248 days of continuous power, all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase." Boeing is in the process of developing a GCU software upgrade that will remedy the unsafe condition. The new model plane previously experienced a battery problem that caused a fire while one aircraft was parked on a runway.

The memo doesn't provide additional details about the underlying software bug. Informed speculation suggests it's a signed 32-bit integer overflow that is triggered after 231 centiseconds (i.e. 248.55 days) of continuous operation.

Judgement call made at the time

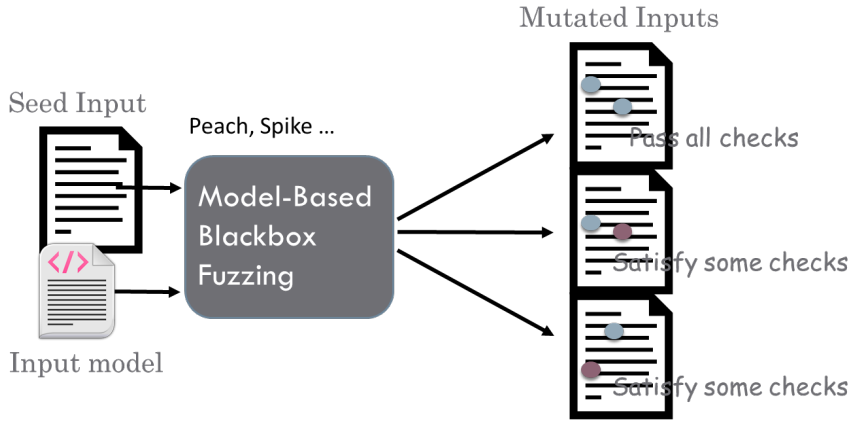




Did not take this approach
Established Approach

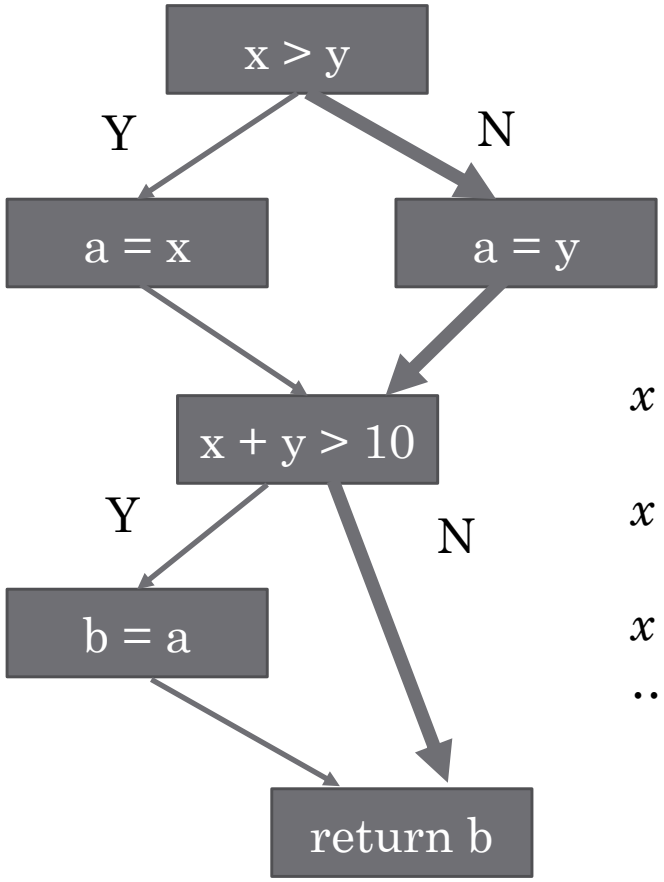
- Learn existing software assurance techniques – what works, and what does not work.
- Whenever possible keep the discussion **rigorous and formal**, but only when possible
- Keep a **pragmatic** outlook, if the rigorous approach is leading to unusable techniques.
- Remember the **developer**: produce techniques which can be integrated into developer workflows.

Black-box Fuzzing



- Inputs
  - Program P
  - Seed input x0
  - Mutation ratio $0 < m \leq 1$
- Next step
  - Obtain an input x1 by randomly flipping $m*|x0|$ bits
  - Run x1 and check if P crashes or terminates properly.
  - In either case document the outcome, and generate next input.
- End of fuzz campaign
  - When time bound is reached, or N inputs are explored for some N.
  - Always make sure that bit flipping does not run same input twice.

White-box Fuzzing

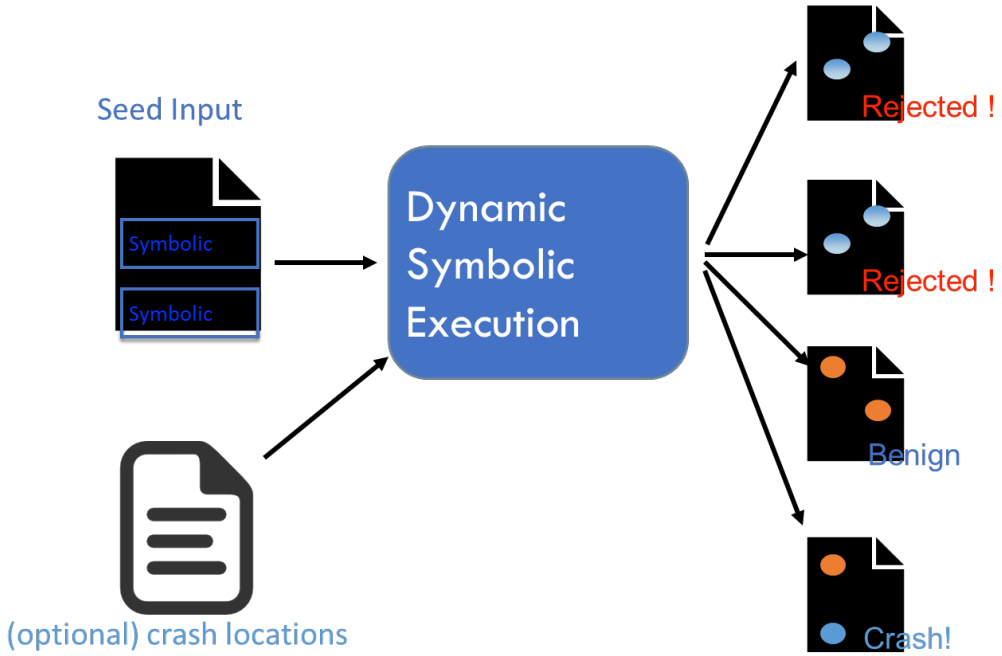


$x \leq y \wedge x+y \leq 10$

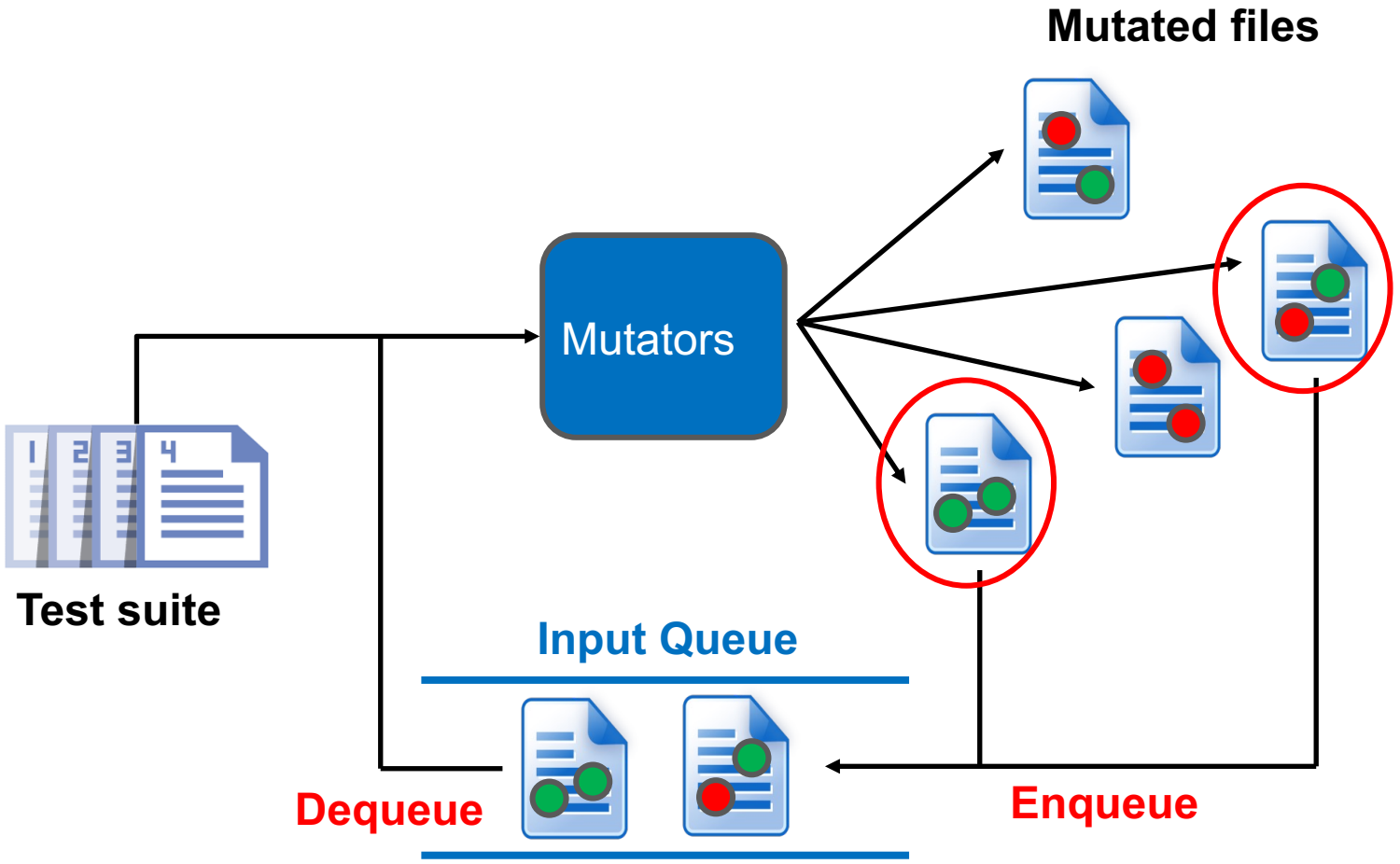
$x \leq y \wedge x+y > 10$

$x > y$

…

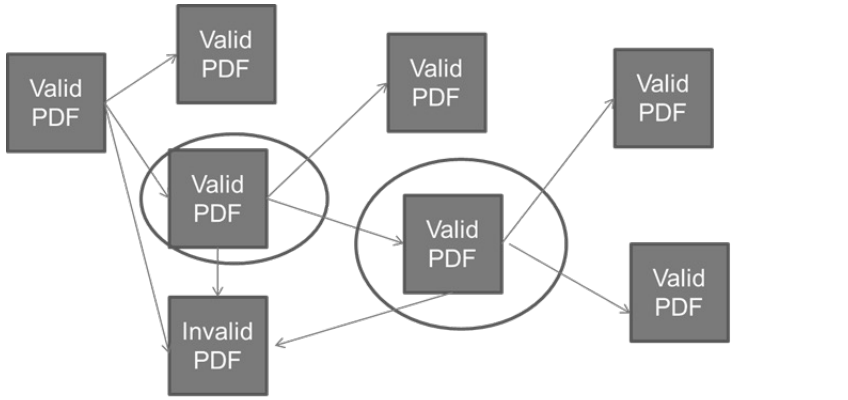


Grey-box Fuzzing



Grey-box Fuzzing Algorithm

- Input: Seed Inputs S
- 1: $T_X = \emptyset$
- 2: T = S
- 3: if T = ∅ then
- 4: add empty file to T
- 5: end if
- 6: repeat
- 7: t = **chooseNext**(T)
- 8: p = **assignEnergy**(t)
- 9: for i from 1 to p do
- 10: t0 = **mutate_input**(t)
- 11: if t0 crashes then
- 12: add t0 to T_X
- 13: else if **isInteresting**(t0) then
- 14: add t0 to T
- 15: end if
- 16: end for
- 17: until timeout reached or abort-signal
- Output: Crashing Inputs T_X



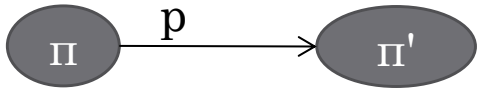
Programming by experienced people

Schematic

```
if (condition1)
        return    // short path, frequented by many inputs
else if  (condition2)
        exit      // short paths, frequented by many inputs
else ....
```

Prioritize low probability paths

- ✓ Use grey-box fuzzer which keeps track of path id for a test.
- ✓ Find probabilities that fuzzing a test t which exercises π leads to an input which exercises π’



- ✓ Higher weightage to low probability paths discovered, to gravitate to those -> discover new paths with minimal effort.

```
1    void crashme (char* s) {
2          if (s[0] == ’b’)
3                if (s[1] == ’a’)
4                      if (s[2] == ’d’)
5                            if (s[3] == ’!’)
6                                  abort ();
7    }
```

Power-Schedules

- Constant: $p(i) = \alpha(i)$
  - AFL uses this schedule (fuzzing ~1 minute)
  - $\alpha(i)$ .. how AFL judges fuzzing time for the test exercising path i

- Cut-off Exponential:

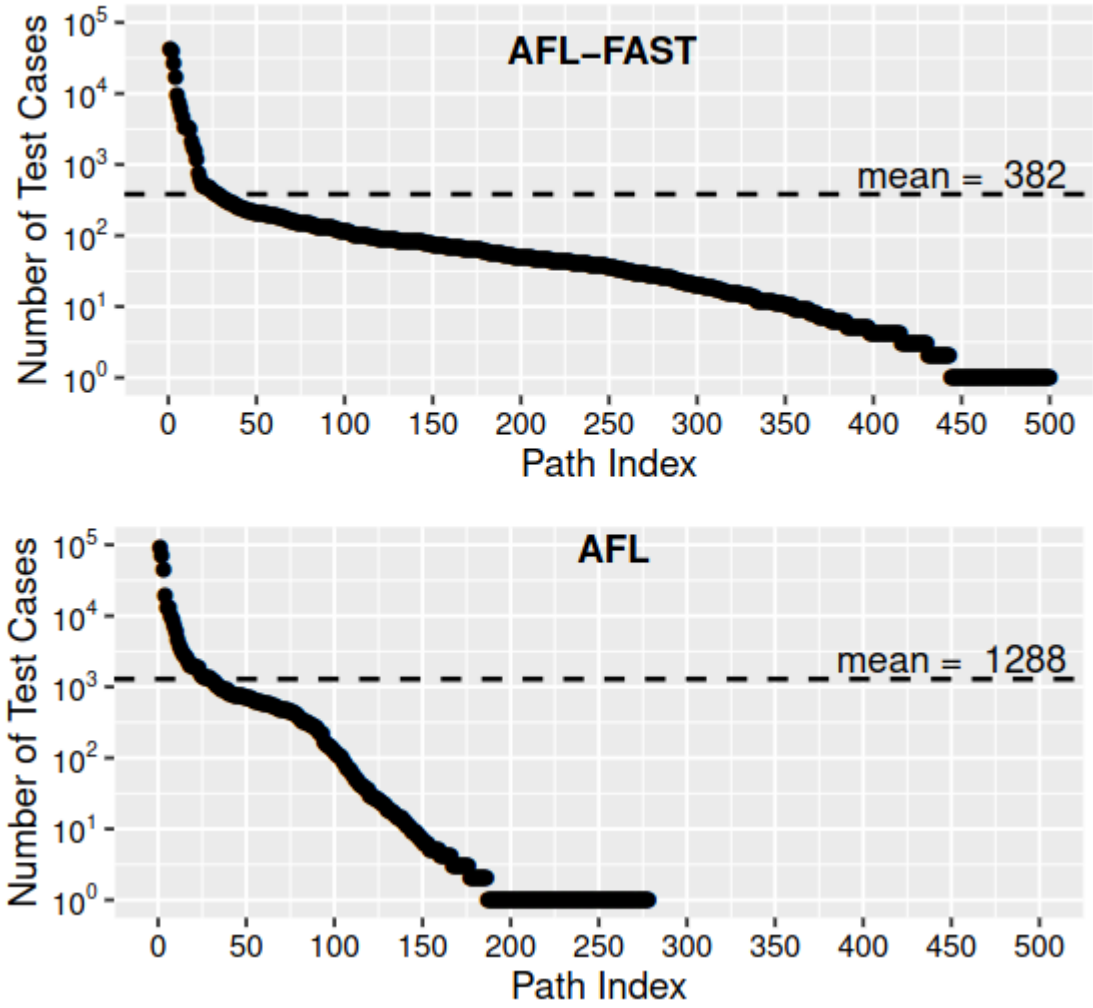
$$p(i) = \begin{cases} 0, & \text{if } f(i) > \mu \\ \min\left((\alpha(i)/\beta)*2^{s(i)}, M \right) & \text{otherwise} \end{cases}$$

β is a constant
s(i) #times the input exercising path i has been chosen from queue
f(i) # generated inputs exercising path i (path-frequency)
μ mean #fuzz exercising a discovered path (avg. path-frequency)
M maximum energy expendable on a state

$$\mu = \left(\sum_{i \in S} f(i) \right) / |S|$$

where S is the set of discovered paths

Showing the idea



GNU
Binutils, nm

Independent Evaluation and Deployment

- An independent evaluation by team Codejitsu found that AFLFast exposes errors in the benchmark binaries of the DARPA Cyber Grand Challenge 19x faster than AFL.
- Picked up by AFL user group, with following observations, paraphrased
  - *AFLFAST assigns substantially less energy in the beginning of the fuzzing campaign.*
  - *Most of the cycles that AFLFAST carries out, are in fact very short. This causes the queue to be cycled very rapidly, which in turn causes new retained inputs to be fuzzed almost immediately. In other words, because AFLFAST assigns less energy, it can process the complete queue substantially faster. We say it starts by exploration rather than by exploitation*
- **Implemented inside AFL and distributed within one year of publication (CCS'16 paper).**

There remain differences between the two in terms of path discovered. More experiments may be needed.

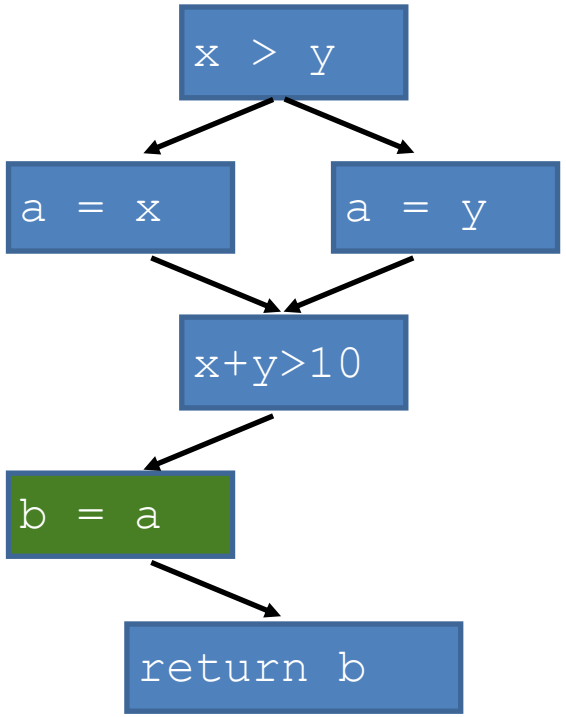
In this talk …

- **Greybox Fuzzing** is frequently used, daily in corporations
  - **State-of-the-art** in automated vulnerability detection
  - **Extremely efficient** coverage-based input generation
    - All program analysis before/at **instrumentation time**.
    - Start with a seed corpus, choose a seed file, **fuzz it**.
    - Add to corpus **only if new input increases coverage**.
  - **Cannot be directed, unlike symbolic execution!**

- Enhance the effectiveness of search techniques, with symbolic execution & model checking as inspiration
  - **Enhance coverage, how to make it directed?**

(Earlier) View-point

- **Directed Fuzzing: classical constraint satisfaction prob.**
  - **Program analysis to identify program paths that reach given program locations.**
  - **Symbolic Execution to derive path conditions for any of the identified paths.**
  - **Constraint Solving to find an input that**
    - **satisfies the path condition and thus**
    - **reaches a program location that was given.**

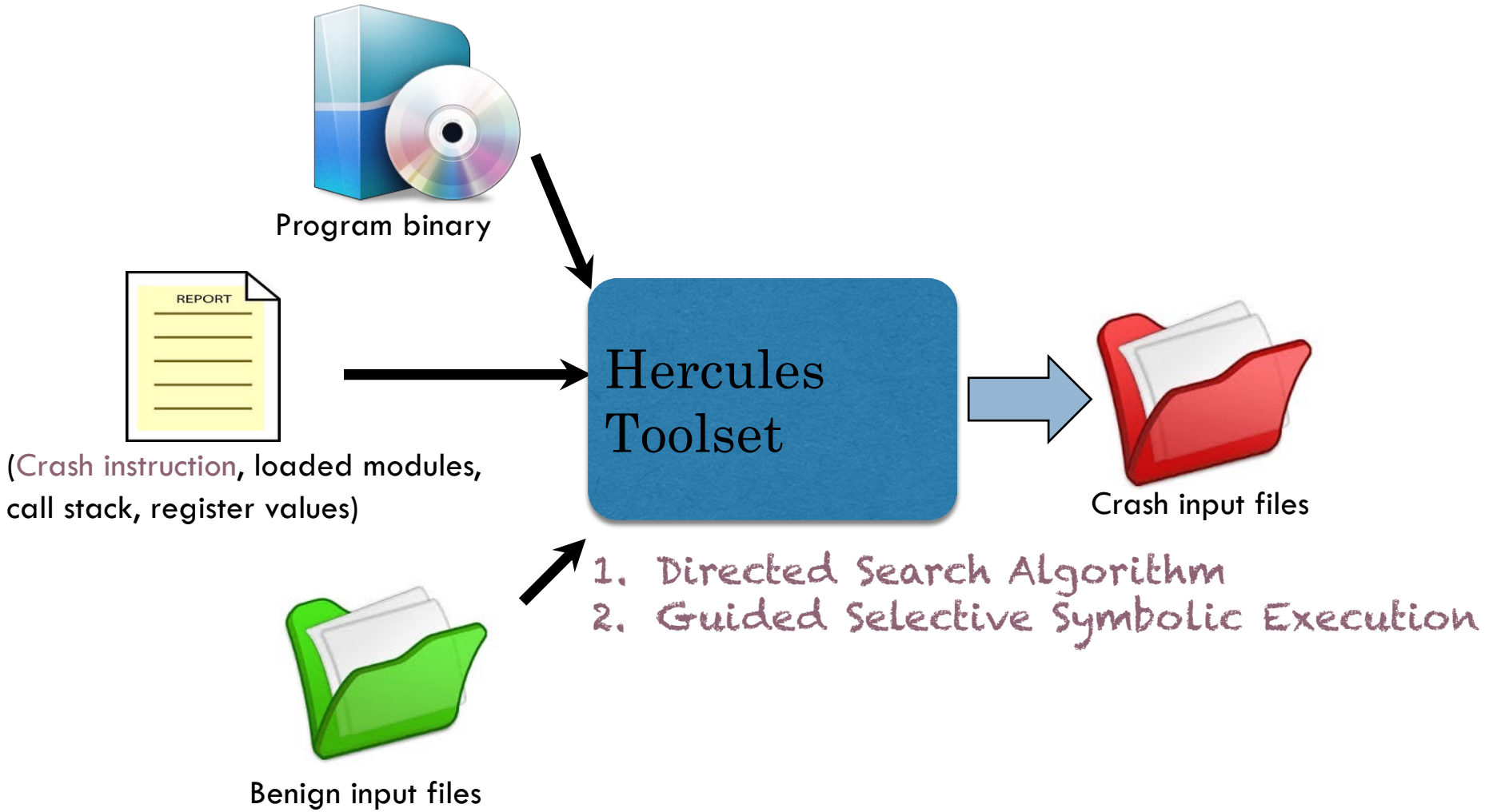


$\varphi_1 = (x>y) \wedge (x+y>10)$

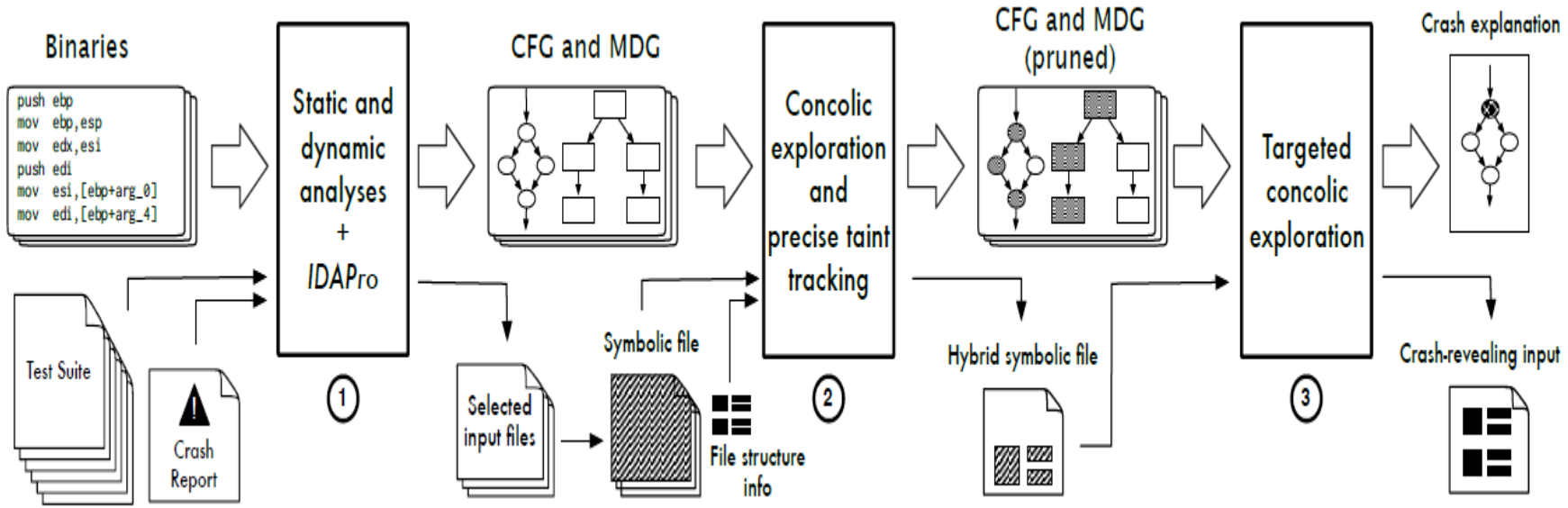
$\varphi_2 = \neg(x>y) \wedge (x+y>10)$

Using symbolic execution

Reproduced vulnerabilities in Acrobat Reader, Media Player with 24 hour time bound

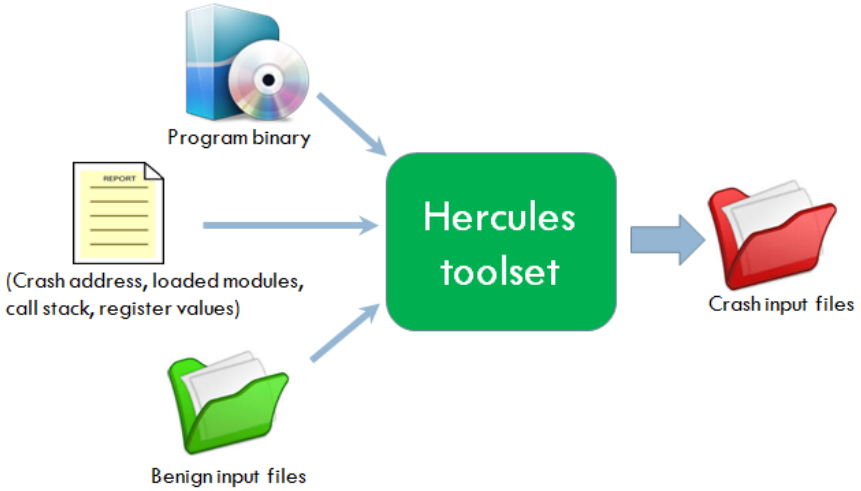


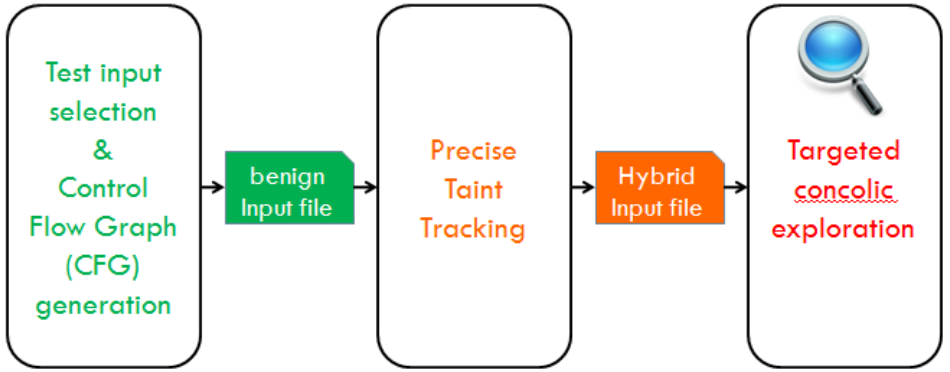
Symbolic Analyzer

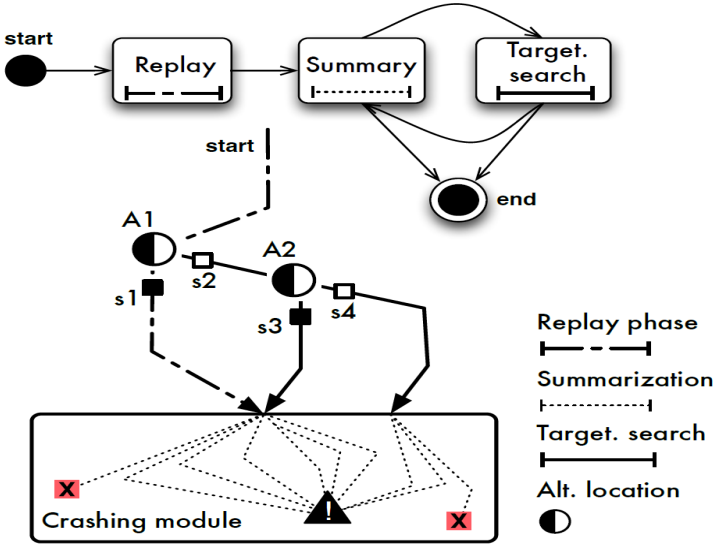


Reproduced vulnerabilities in Acrobat Reader, Media Player with 24 hour time bound [*ICSE15 work, took close to 2 years of effort*]

Hercules!







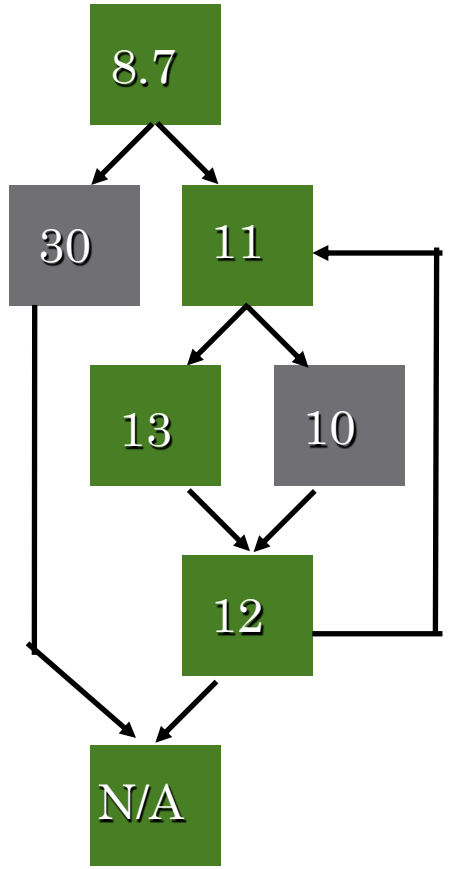
Case studies	Selected modules	S2E (DFS and Random)	Peach Fuzzer - mutation	Hercules
CVE-2010-2204 (Adobe Reader) Memory Access Violation	2 / 78	✗	✗	(*) ✓
CVE-2010-3000 (Real Player) Integer Overflow	2 / 129	✗	✗	✓
CVE-2014-2671 (Windows Media Player) Division by Zero	4 / 84	✗	✗	✓
CVE-2010-0718 (Windows Media Player) Buffer overflow	3 / 86	✗	✗	✓
CVE-2010-0688 (Orbital Viewer) Buffer overflow	2 / 49	✗	✓	✓
CVE-2011-0502 (MAM player) Null pointer reference	1 / 51	✓	✓	(*) ✓

(Later) View-point

- **Directed Fuzzing** as **optimization problem**!
  1. **Instrumentation Time:**
     - Instrument program to **aggregate distance values**.
  2. Runtime, **for each input**
     - decide **how long to be fuzzed** based on distance.
       - If input is **closer** to the targets, it is fuzzed for **longer**.
       - If input is **further away** from the targets, it is fuzzed for **shorter**.

Instrumentation

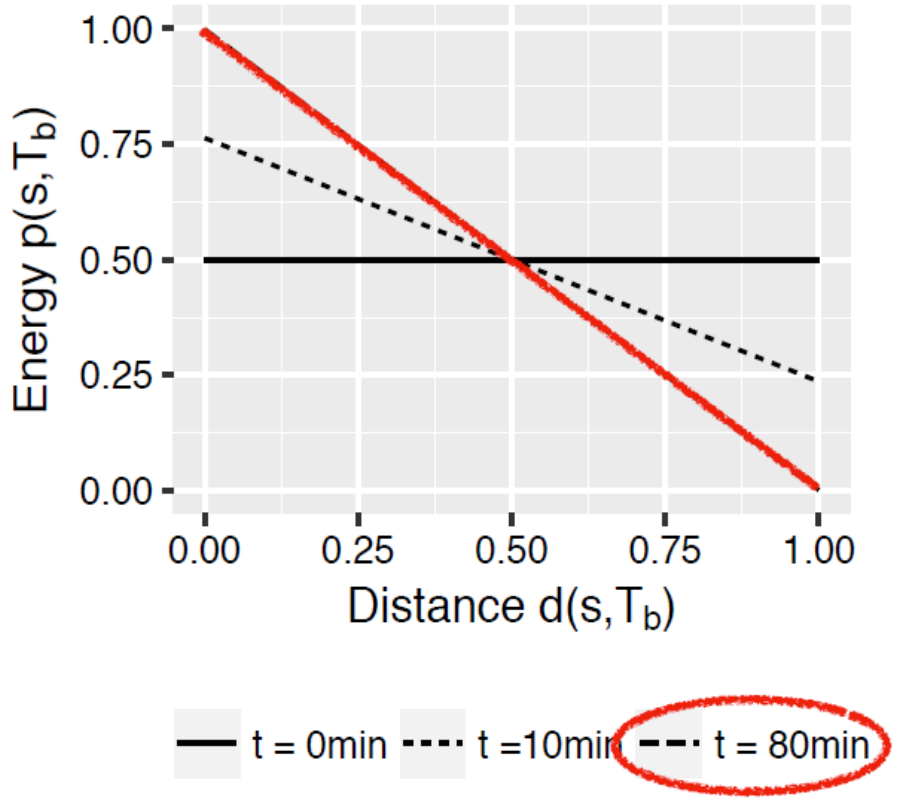
- **Function-level target distance** using call graph (CG)
- **BB-level target distance** using control-flow graph (CFG)
  1. Identify **target BBs** and assign **distance 0**
  2. Identify BBs that call **functions** and assign **10*FLTD**
  3. For **each BB**, compute harmonic mean of (length of shortest path to any function-calling BB + 10*FLTD).



CFG for function b

Directed fuzzing as optimization

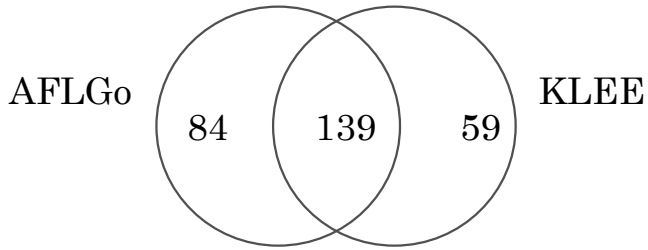
- Integrating Simulated Annealing as power schedule
  - In the beginning (t = 0min), assign the **same energy to all seeds**.
  - Later (t=10min), assign a **bit more energy** to seeds that are **closer**.
  - At exploitation (t=80min), assign **maximal energy** to seeds that are **closest**.



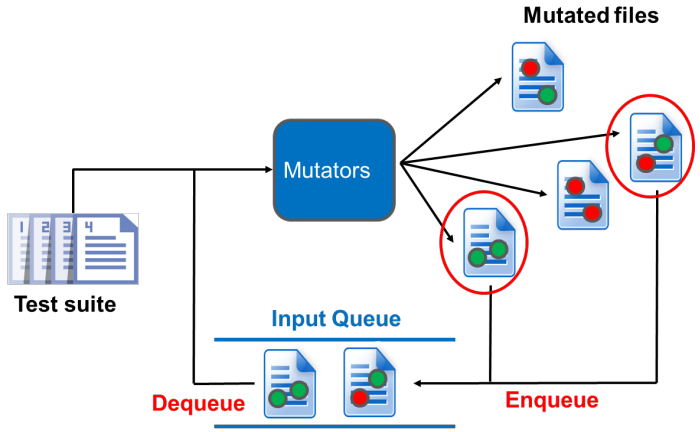
Outcomes

- **Directed greybox fuzzer** (AFLGo) **outperforms symbolic execution-based directed fuzzers** (KATCH & BugRedux)
  - in terms of **reaching more target locations** and
  - in terms of **detecting more vulnerabilities**,
  - on their own, original benchmark sets.
- **Integrated as OSS-Fuzz fork** (AFLGo for Continuous Fuzzing)

- *Tool AFLGo publicly available, follow-up works, survey by community.* **[CCS17 work, less engineering effort]**

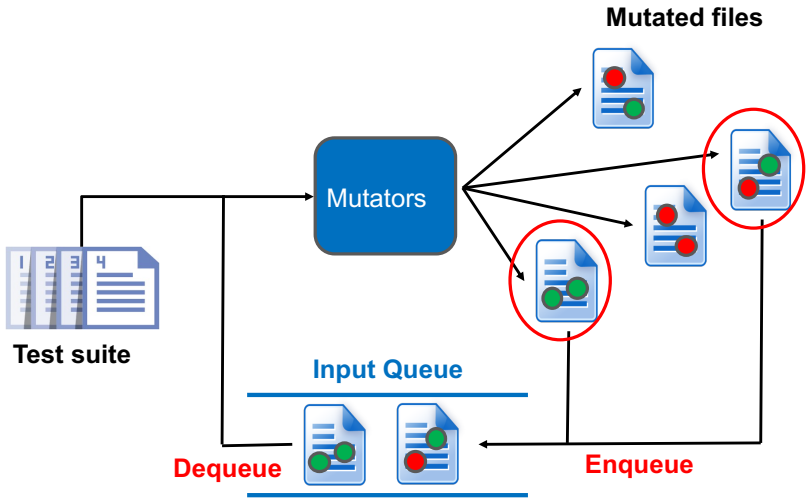


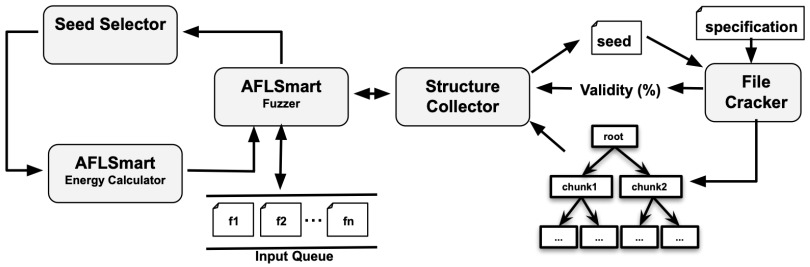
Handling Structured Data?



Make Greybox Fuzzing input-structure aware by

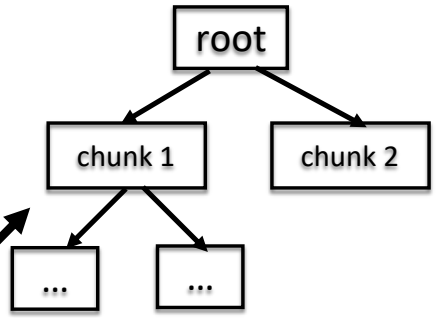
1. Changing input *representation (structured files)*
   - Use tree-like representation instead of bit string
2. Adding new *mutation operators*
   - working at chunk level (e.g., chunk deletion, insertion and splicing)
3. Prioritizing *more valid seed* inputs
   - More valid seeds are assigned higher fuzzing "energy"
4. Applying *optimizations* to retain fuzzing efficiency





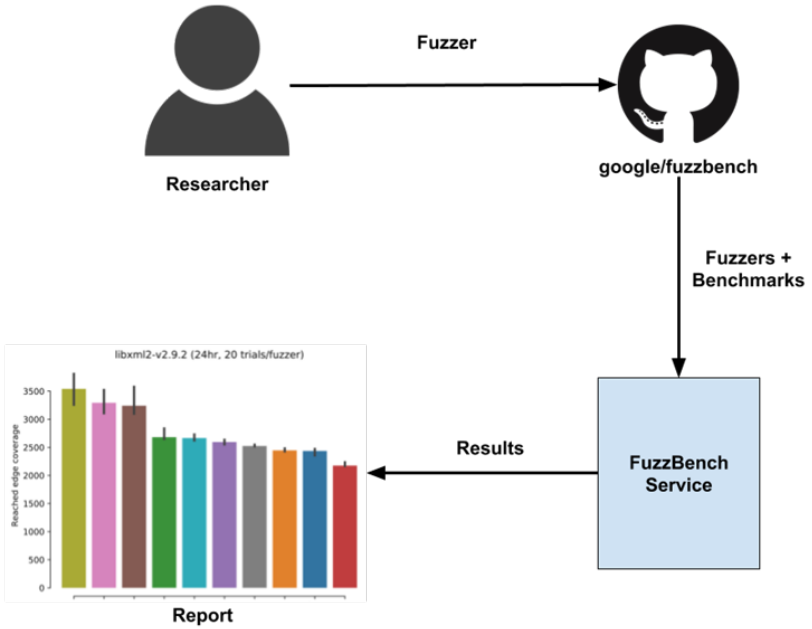

```
<DataModel name="Chunk">
  <String name="ckID" length="4"/>
  <Number name="cksize" size="32" >
    <Relation type="size" of="Data"/>
  </Number>
  <Blob name="Data"/>
  <Padding alignment="16"/>
</DataModel>
<DataModel name="ChunkFmt" ref="Chunk">
   <String name="ckID" value="fmt "/>
   <Block name="Data">
      <Number name="wFormatTag" size="16"/>
      <Number name="nChannels" size="16"/>
      <Number name="nSampleRate" size="32"/>
      <Number name="nAvgBytesPerSec" size="32"/>
      <Number name="nBlockAlign" size="16" />
      <Number name="nBitsPerSample" size="16"/>
   </Block>
</DataModel>
...
<DataModel name="Wav" ref="Chunk">
  <String name="ckID" value="RIFF"/>
  <String name="WAVE" value="WAVE"/>
  <Choice name="Chunks" maxOccurs="30000">
    <Block name="FmtChunk" ref="ChunkFmt"/>
    ...
    <Block name="DataChunk" ref="ChunkData"/>
  </Choice>
</DataModel>
```

XML-based input model.
One input model for each file format.
(e.g., Peach pits)



Seed input

validity score (0->100)
100: the whole seed is fully cracked/parsed



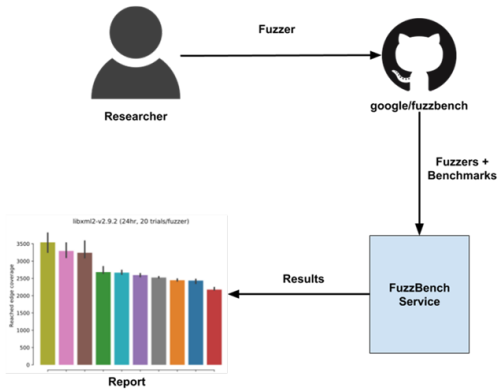
Hot fuzz: Bug detectives whip up smarter version of classic ...
https://www.theregister.co.uk › 2018/11/28 › better_fuzzer_aflsmart
Nov 28, 2018 - Known as **AFLSmart**, this fuzzing software is built on the powerful American ... We're told **AFLSmart** is pretty good at testing applications for common .... The Register - Independent **news** and views for the tech community.

AFLSmart | Latest AFLSmart News, Articles and Updates
https://cyware.com › tags › aflsmart
AFLSmart - Check out latest **news** and articles about **AFLSmart** on Cyware.com. We provide machine learning based curation engine brings you the top and ...

Researchers Introduce Smart Greybox Fuzzing | SecurityWeek ...
https://www.securityweek.com › researchers-introduce-smart-greybox-fuzz...
Nov 29, 2018 - Information Security **News**, IT Security **News** and Cybersecurity Insights: ... According to the experts, **AFLsmart** is highly efficient in analyzing ...

Comparisons in FuzzBench

Community Efforts



Fuzzbench

(follow-up of discussions in Shonan)

Shonan Meeting 2019

(Boehme, Cadar, Roychoudhury)

NII SHONAN MEETING

(more) POINTERS

Coverage-based Greybox Fuzzing as Markov Chain, CCS16
Directed Greybox Fuzzing, CCS17.
Smart Greybox Fuzzing, TSE21.

Linear-time Temporal Logic guided Greybox Fuzzing,
Ruijie Meng et al, ICSE 2022 (Now).

The Fuzzing Book
Andreas Zeller et al

Fuzzing: Challenges and Reflections
IEEE Software, 38(3), pages 79-86, 2021,
Outcome from a 2019 Shonan Meeting.


Fuzzing: Challenges and Reflections

Marcel Böhme, Monash University

Cristian Cadar, Imperial College London

Abhik Roychoudhury, National University of Singapore

// We summarize the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a Shonan Meeting and that were validated in a subsequent survey. //

ACKNOWLEDGEMENT: National Cyber Security Research program from NRF Singapore

Quote on …

"He continues to teach because it provides him with a livelihood; also because it teaches him …

…

The irony does not escape him: that the one who comes to teach learns the keenest of lessons,

while those who come to learn learn nothing."

J. M. Coetzee

Using fuzzing for complex oracles?

Search

- Enhance the effectiveness of search techniques, with symbolic execution as inspiration
  - **Enhance coverage**
  - **Achieve directed search**
  - **More advanced properties than crashes! Get close to the effect of verification as in model checking**

$$M \models \phi$$

Model Checking via Dir. Fuzzing

Testing reactive systems

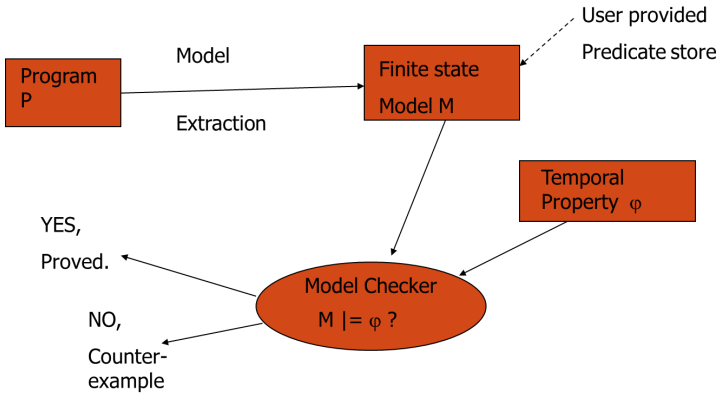
Most uses of **Software Model Checking** are for bug finding

Restricted set of properties for software model checking	Mostly restricted to proving / disproving of invariants due to nature of state abstractions	Unnecessary state savings and state explosion problem.

Bug finding search in model checking via directed fuzzing

Cover the whole specification language of properties for a well-known and popular temporal logic – LTL	No state explosion problem as in model checking. **Fuzzing for more advanced oracles than simply crashes and hangs!**

Software Model Checking



Used in SPIN Model checker

Consider $\neg\varphi$. None of the exec. traces of M should satisfy $\neg\varphi$.

Construct a finite-state automata $A_{\neg\varphi}$ such that

- Language($A_{\neg\varphi}$) = Traces satisfying $\neg\varphi$

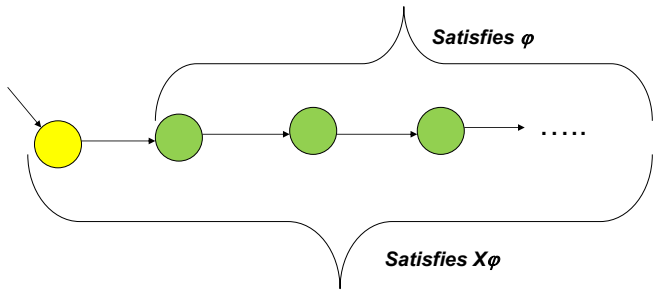
Construct the synch product $M \times A_{\neg\varphi}$

Check whether any exec trace σ of M is an exec trace of the product $M \times A_{\neg\varphi}$ i.e. check Language($M \times A_{\neg\varphi}$) = empty-set?

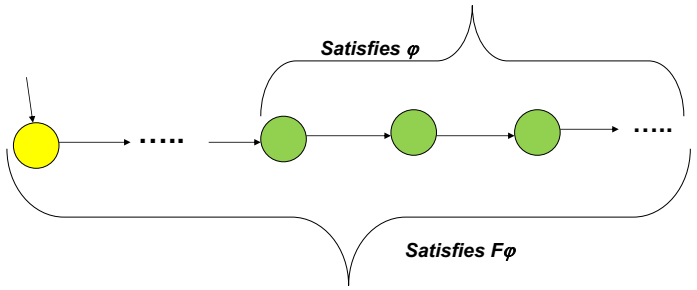
- Yes: Violation of φ found, report counterexample σ
- No: Property φ holds for all exec traces of M.

Linear-time Temporal Logic

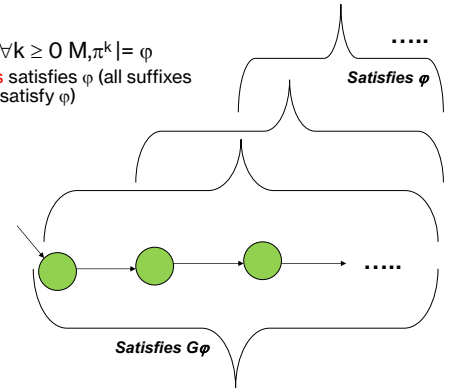
- $M,\pi \models X\varphi$ iff $M,\pi^1 \models \varphi$
  - Path starting from next state satisfies φ

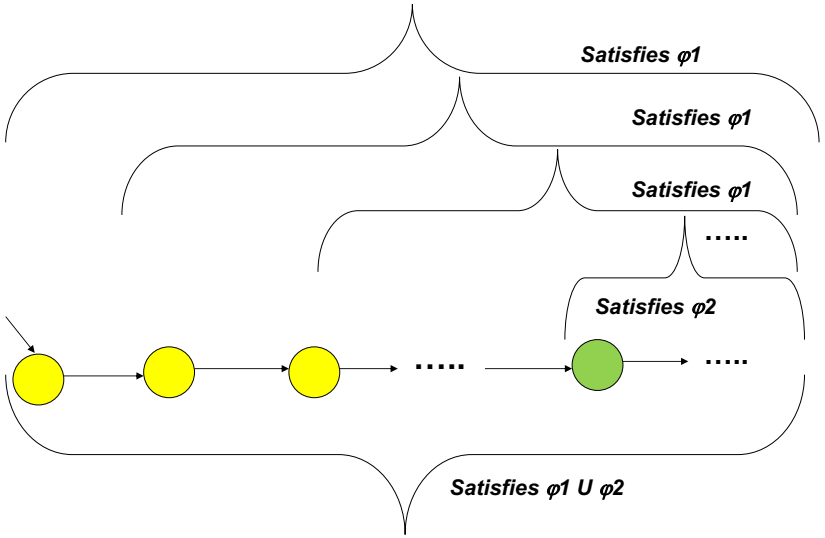


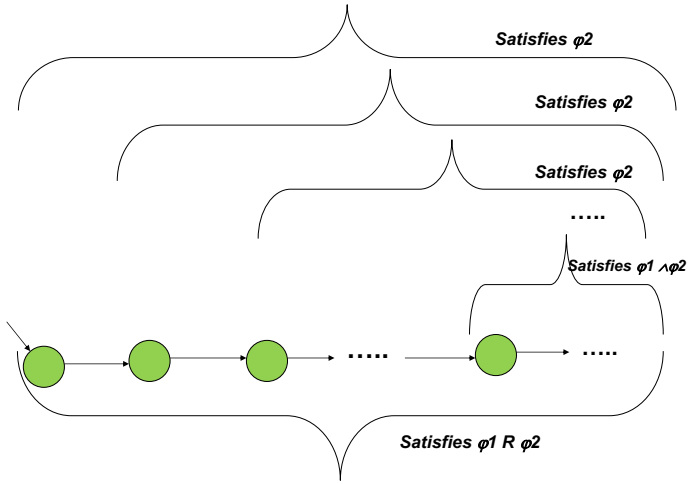
- $M,\pi \models F\varphi$ iff $\exists k \geq 0\ M,\pi^k \models \varphi$
  - Path starting from an eventually reached state satisfies φ

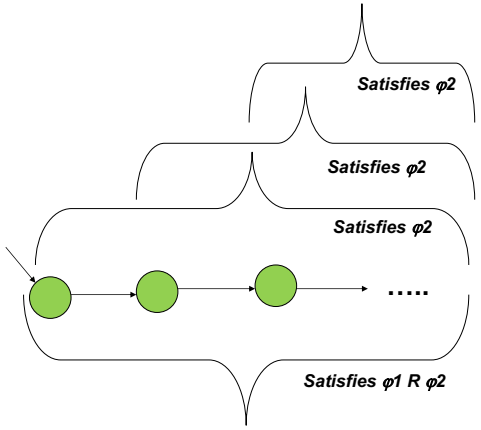


- $M,\pi \models G\varphi$ iff $\forall k \geq 0\ M,\pi^k \models \varphi$
  - Path always satisfies φ (all suffixes of the path satisfy φ)



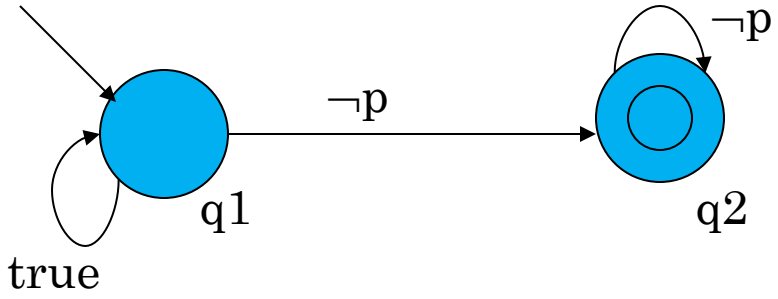






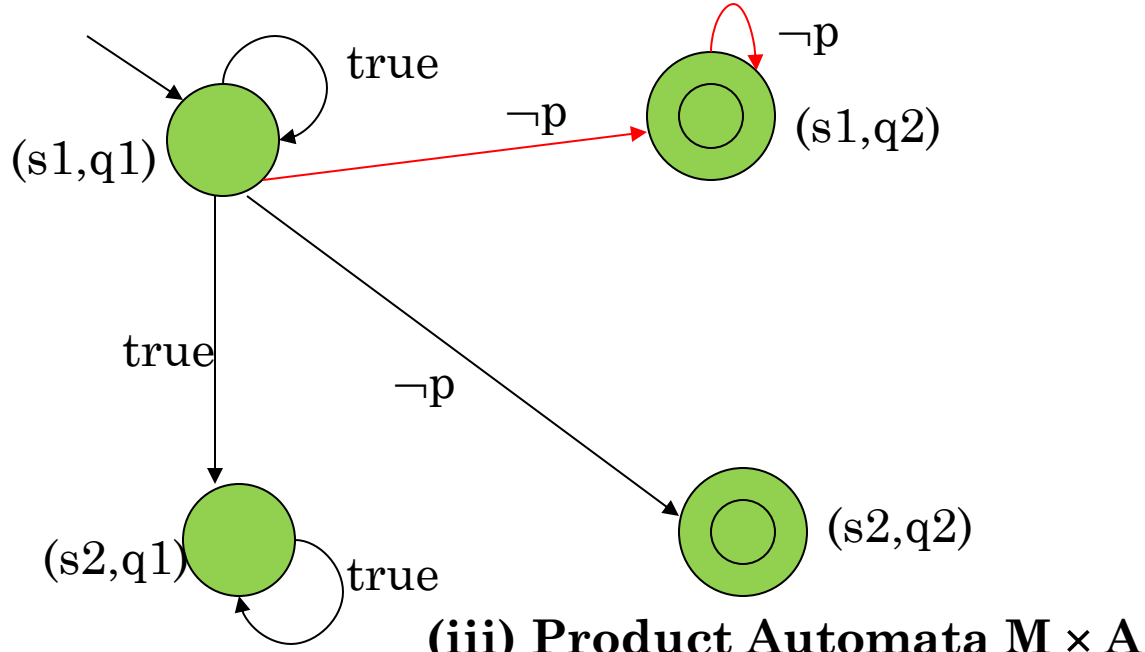
Product Automata

(i) System Model M



(ii) Property Automata A

M ⊭ GF p



(iii) Product Automata M × A

NO	PID	Property Description	LTL Notation
1	$Pr\,F_5$	If receiving invalid username or invalid password, the server must always show the same message to the user.	G(((request = InvalidUsername) ∨ (request = InvalidPassword)) → X(G(sameResponse)))
2	$Pr\,F_6$	If receiving the CWD request without login, the server must not give the CommandOkay response.	G((¬(state = LogIN) ∧ (request = CWD)) → X(G(¬(response = CommandOkay))))
3	$Pr\,F_7$	After a connection is constructed successfully, there should be a successful login and after that without failed login.	G(((request = ValidUserName&ValidPasswd) → X(response = LoginSuccess)) → X(G(¬(response = LoginFailed))))
4	$Pr\,F_8$	After the connection is lost after a long time, responses should be always timeout.	G(LostConnection → X(G(response = Timeout)))
5	LV_6	If the server is in the Play state and receives a Pause request, should go into the Ready state.	G(((state = Play) ∧ (request = Pause)) → X(state = Ready))

Simple (sample) properties

Temporal Logic guided Fuzzing

1

Find

- Find acceptance states reachable from initial states (DFS).
- **Conduct fuzzing to reach an accepting state s from initial state**

2

Find

- Find all such acceptance states which are reachable from itself (DFS).
- **Conduct fuzzing to reach state s back from state s**

3

Counter-example

- Counter-example evidence (if any) obtained by simply concatenating the two DFS stacks.
- **Construct violating input from the two fuzzing runs**

Büchi Automata Guided Fuzzing

Input: P': The transformation of program under test
Input: $\mathcal{A}_{\neg\phi}$: Automata of negation of property under test
Input: *map*: Map between propositions and program locations
Input: *flag*: True for liveness properties
Input: *total_time*: Time budget for fuzzing
Input: *target_time*: Time budget for reaching a program location

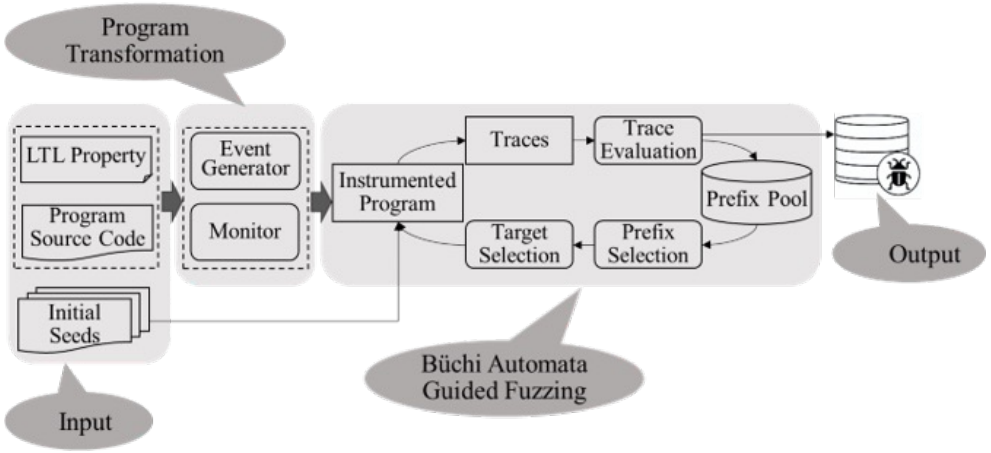
1 **Procedure** Fuzz(P', $\mathcal{A}_{\neg\phi}$, *map*, *flag*, *total_time*, *target_time*)
2 $s_0 \leftarrow$ getInitState($\mathcal{A}_{\neg\phi}$) ;
3 $\mathcal{X} \leftarrow \{\langle \emptyset, s_0 \rangle\}$; // Starting with init state of $\mathcal{A}_{\neg\phi}$
4 **for** *time* < *total_time* **do**
5 $\langle x_t^i, x_t^s \rangle \leftarrow$ selectPrefix($\mathcal{X}$) ;
6 $p \leftarrow$ selectTargetAtomicProposition($\mathcal{A}_{\neg\phi}, x_t^s$) ;
7 $l \leftarrow$ selectProgramLocationTarget(*map*, p)
8 **for** *time'* < *target_time* **do**
 // $\mathcal{D}$: Feedback of CFG distance
 // S_{power}: Power schedule algorithm
9 $I \leftarrow$ generateInput($\mathcal{D}, S_{power}$) ;
10 $I' \leftarrow$ replacePrefix(I, x_t^i) ;
11 $d, \langle x^i, x^s \rangle \leftarrow$ evaluate($P', I', flag$);
12 $\mathcal{D} \leftarrow \mathcal{D} \cup \{d\}$;
13 $\mathcal{X} \leftarrow \mathcal{X} \cup \{\langle x^i, x^s \rangle\}$
14 **end**
15 **end**

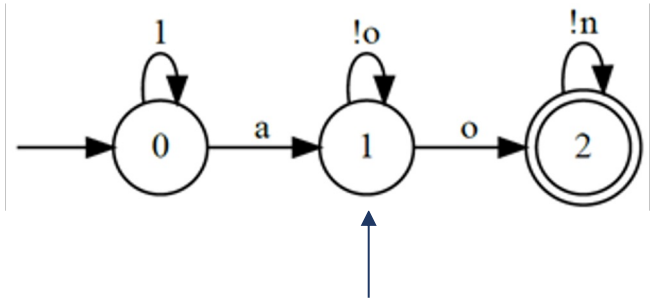
Prefix Selection

Target Selection

Trace Evaluation

Progress Saving





Save Progress

Table 4: Zero-day Bugs found by LTL-Fuzzer; for several of them CVEs have been assigned.

Prop	Program	Description of violated properties	Bug Status
TD_1	TinyDTLS0.9	If the server is in the WAIT_CLIENTHELLO state and receives a ClientHello request with valid cookie and the epoch value 0, must finally give ServerHello responses.	CVE-2021-42143, fixed
TD_2	TinyDTLS0.9	If the server is in WAIT_CLIENTHELLO state and receives a ClientHello request with valid cookie but not 0 epoch value, must not give ServerHello responses before receiving ClientHello with 0 epoch value.	CVE-2021-42142, fixed
TD_3	TinyDTLS0.9	If the server is in the WAIT_CLIENTHELLO state and receives a ClientHello request with an invalid cookie, must reply HelloVerifyRequest.	CVE-2021-42147, fixed
TD_5	TinyDTLS0.9	If the server is in the DTLS_HT_CERTIFICATE_REQUEST state and receives a Certificate request, must give a DTLS_ALERT_HANDSHAKE_FAILURE or DTLS_ALERT_DECODE_ERROR response, or set Client_Auth to be verified.	CVE-2021-42145, fixed
TD_{11}	TinyDTLS0.9	After the server receives a ClientHello request without renegotiation extension and gives a ServerHello response, then receives a ClientHello again, must refuse the renegotiation with an Alert.	Confirmed
TD_{12}	TinyDTLS0.9	After the server receives a ClientHello request and gives a ServerHello response, then receives a ClientKeyExchange request with a different epoch value than that of ClientHello, server must not give ChangeCipherSpec responses.	CVE-2021-42141, fixed
TD_{13}	TinyDTLS0.9	After the server receives a ClientHello request and gives a ServerHello response, then receives a ClientHello request with the same epoch value as that of the first one, server must not give ServerHello.	CVE-2021-42146
TD_{14}	TinyDTLS0.9	If the server receives a ClientHello request and gives a HelloVerifyRequest response, and then receives a over-large packet even with valid cookies, the server must refuse it with an Alert.	CVE-2021-42144, fixed
CT_1	Contiki-Telnet3.0	After WILL request is received and the corresponding option is disabled, must send DO or DONT responses.	CVE-2021-40523
CT_2	Contiki-Telnet3.0	After DO request is received and the corresponding option is disabled, must send WILL or WONT responses.	Confirmed
CT_7	Contiki-Telnet3.0	After WONT request is received and the corresponding option is disabled, must not give responses.	CVE-2021-38311
CT_8	Contiki-Telnet3.0	After DONT request is received and the corresponding option is disabled, must not give responses.	Confirmed
CT_{10}	Contiki-Telnet3.0	Before Disconnection, must send an Alert to disconnect with clients.	CVE-2021-38387
CT_{11}	Contiki-Telnet3.0	If conducting COMMAND without AbortOutput, the response must be same as the real execution results.	CVE-2021-38386
PuF_5	Pure-FTPd1.0.4	When quota mechanism is activated and user quota is exceeded, must finally reply a quota exceed message.	CVE-2021-40524, fixed

Finding Zero-day Bugs

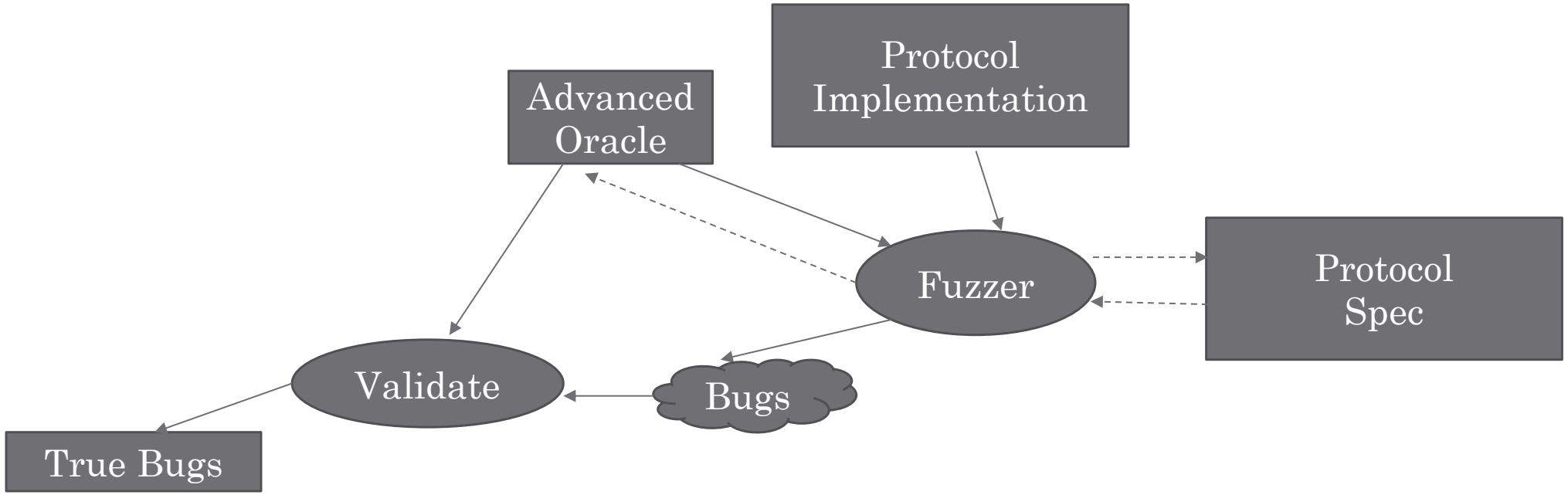
NUS Internal Talk, July 2023

Looking forward

Lot of past works on fuzzing have focused on parsers or file format processors.

Stateful system fuzzing could be the next step – internet facing protocols.

Model Checking efficacy, without guarantees, and *without* state caching.



Research Motivation

1. What is most effective : find more bugs (Utilitarian)

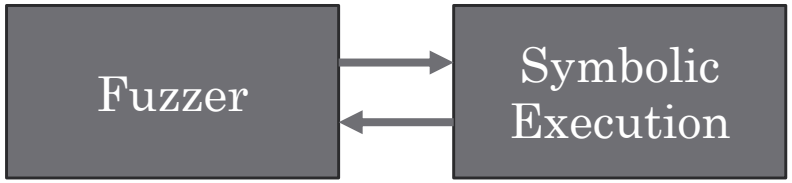
2. The fuzzing search is less sophisticated – how to make it more efficient - or more effective ? May be combine fuzzing and symbolic execution via tools being integrated synergistically (Technical)

3. *How can you achieve a fuzzing search which will look and feel like fuzzing but in effect achieves symbolic execution ?*

   What is the smallest change in the fuzzing algorithm which will achieve this?

   [Imaginative]

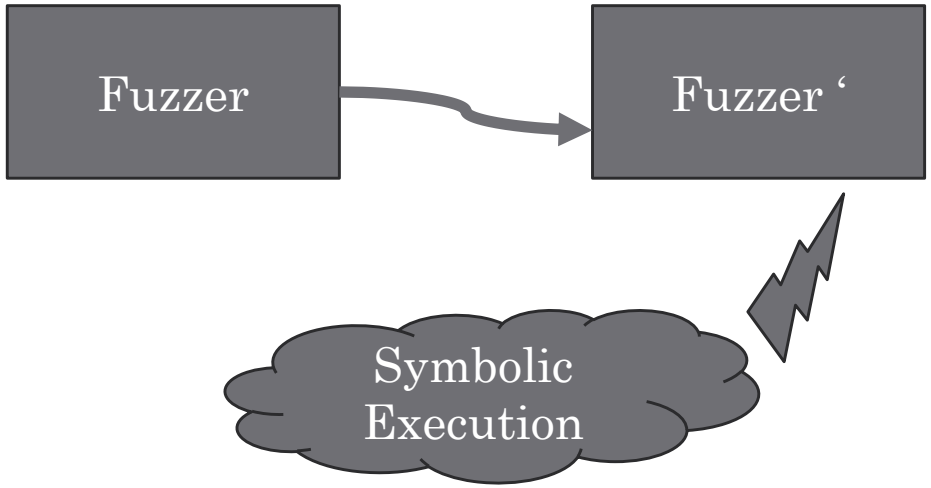
Judgement call to enable translation



Did not take this approach

Established Approach

Usability Concerns



Developed this approach

Helped achieve translation despite limited outreach ability at that time.

Is research translated?...

Most research today is only getting used by other research groups …

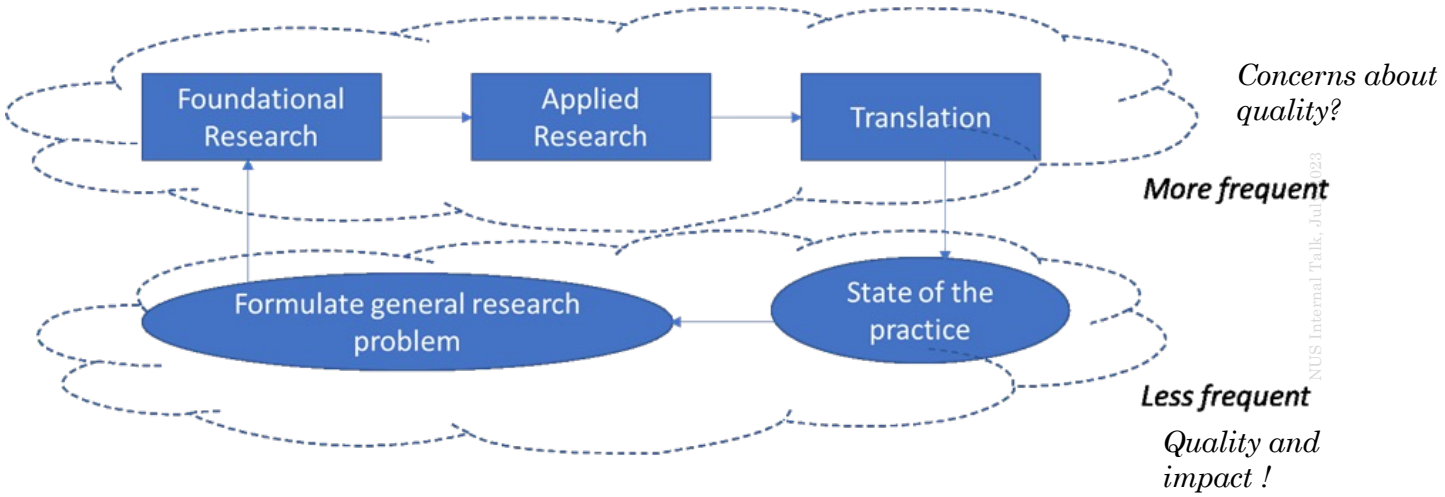
Hence the extra focus on citations in today's research.

Genuine concern about innovation in research not making its way to deployment!

At the same time, genuine concerns about focus on translation-oriented work affecting research quality.

Translation is typically not of your tool.
Companies have concerns and will re-implement.

Ack: National Research Foundation FRC report ---
Foundational Research Capability study 2021,
Foundations of Security and Data Privacy.



Roundtrip Engineering for growing mature R&D ecosystem

NATIONAL RESEARCH FOUNDATION
PRIME MINISTER'S OFFICE
SINGAPORE
Research . Innovation . Enterprise

Translation: a fresh look

Marcel Boehme

Max Planck Institute

Van-Thuan Pham

University of Melbourne

Students

Collaboration with Students

Each person can bring in their own perspective.

As a result, it can take place as a real **collaboration** among us.

This is what everyone would like to achieve, BUT

- there was a very significant learning period before this.

What is most effective : find more bugs [Utilitarian]

The fuzzing search is less sophisticated – how to make it more efficient - or more effective ? [Technical]

How can you achieve a fuzzing search which will look and feel like fuzzing but in effect achieves symbolic execution ? What is the smallest change in the fuzzing algorithm which will achieve this? [Imaginative]

Fostering such collaborations

Diversity of research team

- Not only explicit diversity e.g. geographical
- Also implicit diversity e.g. training and thought

→

By a corollary, beyond mastery of one technology

- The fuzzing works would not have been possible otherwise.

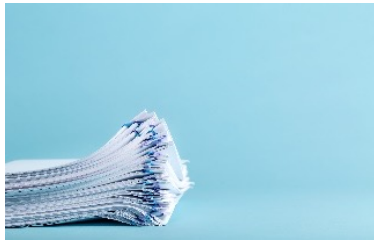
↓

Qualitative outlook instead of excessive focus on results

→

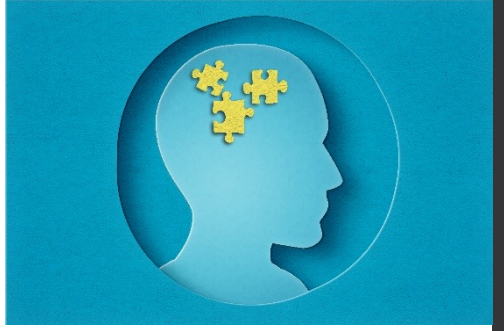
- Convince each other on how much a proposed idea is new …

Student Qualities?



Papers < Placement < Discussions

Output < Outcome < Experience



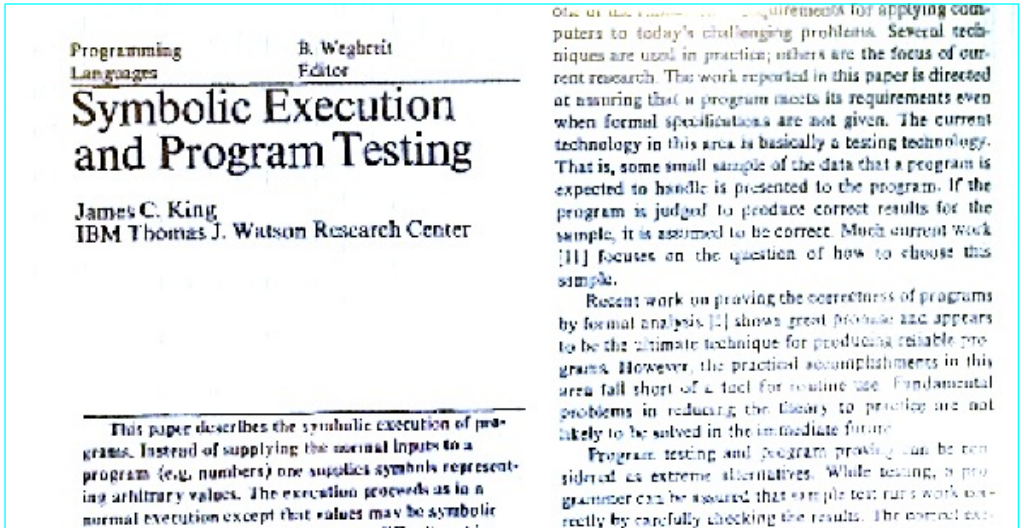
Part of bigger agenda …

Fuzzing

- Enhance the effectiveness of search techniques, with symbolic execution as inspiration
  - **Enhance coverage**
  - **Achieve directed search**
  - **Find temporal logic violations without MC overheads**

Symbolic Execution

- Explore capabilities of symbolic execution *beyond testing which has been studied since 1976 (see below)*
- **Specification inference: Program repair**



Programming Languages — B. Wegbreit, Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

This paper describes the symbolic execution of programs. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic

computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

Recent work on proving the correctness of programs by formal analysis [2] shows great promise and appears to be the ultimate technique for producing reliable programs. However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved in the immediate future.

Program testing and program proving can be considered as extreme alternatives. While testing, a programmer can be assured that sample test runs work correctly by carefully checking the results. The normal ex-

Research Program Announcement

New research program on fuzzing stateful systems at NUS (July 23 – 2027)

PI: Abhik Roychoudhury
Co-PI: Zhenkai Liang, Umang Mathur, Manuel Rigger.

Chronological Evolution of Capabilities

Point Projects MINDEF, MoE... e.g. "Symbolic Taint Analysis" DSTA DIRP [2009-12, 2011-14, 2013-15]

Targeted Capability NCR 1 TSUNAMi(2015 –19) Securify(2015 - 20)

Satellite Center of Excellence Managed by NUS+NTU (2019-23)

Future plan

Fuzz testing – NUS – this proposal (2023-26)

Other efforts from NTU (2023-26)

TSS - Space of Problems

- **Fuzz Testing**
  - Feed semi-random inputs to find hangs and crashes
- **Continuous fuzzing**
  - Incrementally find new "problems" in software
- **Crash reproduction**
  - Re-construct a reported crash, crashing input not included due to privacy
- **Reaching nooks and corners**
- **Localizing reported observable errors**
- **Patching reported errors from input-output examples**

ALL of these problems benefit from solutions in fuzz testing.

Structure of proposal

- 1. Reactive Systems fuzzing
  - Stateful behavior
  - Interaction
- 2. Concurrent System fuzzing
  - Class of concurrency bugs
  - Schedule space
- 3. Data intensive systems
  - Often distributed in nature
- 4. Fuzzer evaluation and usage
  - Binary analysis support
  - Tool collaboration and working with industry

Impact

1. **For entire fuzzing community:** Sample starting corpora from a large pool of seeds with varied characteristics and origins, and broad target programs

2. **For fuzzing researchers and fuzzer builders:** Use explainable benchmarking techniques to account for the effects of covariates during benchmarking

3. **For practitioners or fuzzer users:** There is a dichotomy between LibFuzzer and the other fuzzers w.r.t. corpus and program properties

Applications welcome

https://nus-tss.github.io/fuzzing

- Post-doc
- PhD Student
- Research Assistant

Singapore Fuzzing Summer School 24

- Annually to be held during the 4-year program.
- First installment May 2024.
- The inaugural Singapore Fuzzing Summer School will debut on the week of 27 – 31 May 2024 at the National University of Singapore in Singapore. The school will focus on recent advances in fuzzing technology and the practical application of fuzz testing tools. The school invites both postgraduate students and researchers with a relevant interest in software testing. The school also invites industry professionals who wish to gain practical hands-on knowledge on fuzz testing tools and technologies