

# Directed Fuzzing

---

August  
2019



Amir Abas Kabiri Zamani 97131100

# What our journey contains

- Testing and Categories
- Fuzzing Position
- Directed Fuzzing
- DGF Vs. DSE
- AFLGO
- History of Fuzzing
- References



# Modern Testing Thinking



---

- **Verification + Validation:**
  - **Verification:**
    - “Did we build the product right?”
  - **Validation:**
    - “Did we build the right product?”

# Fuzzing



- **From a quality assurance perspective:**
  - Fuzzing is a branch of testing; testing is a branch of quality control; quality control is a branch of quality assurance.

# Fuzzing



- A highly automated testing technique that covers numerous boundary cases using invalid data as application input to better ensure the absence of exploitable vulnerabilities.
- The name comes from modem applications tendency to fail due to random input caused by line noise on fuzzy telephone lines.

# Fuzzing VS Testing



---

- Fuzzing differs from other testing methods in that it :
  - Tends to focus on input validation errors
  - Tends to focus on actual applications and dynamic testing of a finished product
  - Tends to ignore the responses, or valid behavior
  - Concentrates mostly on testing interfaces that have security implications

# Testing



---

- To a QA person, a test has to have a purpose, or otherwise it is meaningless.
- Without a test purpose, it is difficult to assign a test verdict—that is, did the test pass or fail?
- Various types of testing have different purposes.
- Black-box testing today can be generalized to focus on three different purposes:
  - Features
  - Robustness
  - Performance

# Positive and Negative Testing



---

- **Positive Testing**
  - can be divided into:
    - Feature Testing
    - Performance Testing
  - Test requirements for feature testing consist of a set of valid use cases, which may consist of only few dozens or at most hundreds of tests.
  - Performance testing repeats one of the use cases using various means of test automation.



# Positive and Negative Testing



---

- **Negative Testing**

- Tries to test the robustness of the system through exploring the infinite amount of possible anomalous inputs to find the tests that cause invalid behavior.
- An anomaly can be defined as any unexpected input that deviates from the expected norm, ranging from simple field-level modifications to completely broken message structures or alterations in message sequences.

# Testing Categories



---

- **Feature testing, or conformance testing:**
  - **Verifies that the software functions according to predefined specifications.**
  - **The features can have relevance to security— for example, implementing security mechanisms such as encryption and data verification.**

# Testing Categories



---

- **Performance testing:**
  - Tests the performance limitations of the system.
  - Typically consisting of positive testing only, meaning it will send large amounts of legal traffic to the SUT(System Under Test).
  - Performance testing is often called load testing or stress testing, meaning that a system should tolerate the unfortunate event when everyone tries to use it simultaneously.

# Testing Categories

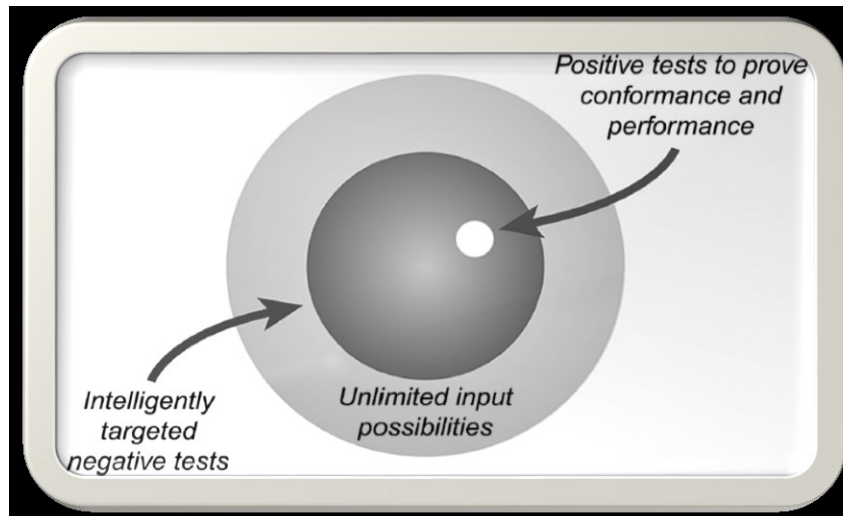


---

- **Robustness testing (including fuzzing):**
  - Is complementary to both feature and performance tests.
  - Robustness can be defined as an ability to tolerate exceptional inputs and stressful environmental conditions. Software is not robust if it fails when facing such circumstances.
  - Most security vulnerabilities reported in the public are caused by robustness weaknesses.
  - Whereas both feature testing and performance testing are still positive tests, based on real-life use cases, robustness testing is strictly negative testing with tests that should never occur in a well-behaving, friendly environment.

# Fuzzing

- Fuzzing is one form of robustness testing, and it tries to fulfill the testing requirements in negative testing with random or semi-random inputs.
- But more often robustness testing is model-based and optimized, resulting in better test results and shorter test execution time due to optimized and intelligently targeted tests selected from the infinity of inputs needed in negative testing.



Limited input space in positive tests and the infinity of tests in negative testing

# Testing Categories



---

- **Functional Testing:**
  - In contrast to structural testing disciplines, fuzzing falls into the category of functional testing, which is more interested in how a system behaves in practice rather than in the components or specifications from which it is built.
  - The system under test during functional testing can be viewed as a black box, with one or more external interfaces available for injecting test cases, but without any other information available on the internals of the tested system.



# Fuzzing

**Barton Miller at the University of Wisconsin in 1990:**

- There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. A program is detected as faulty only if it crashes or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures. While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states.

# Fuzzing



- Delivering input to the software through different communication interfaces with no or very little knowledge of the internal operations of the SUT. Fuzzing is a black-box testing technique in which the SUT is stressed with unexpected inputs and data structures through external interfaces.
- Fuzzing is proactive, whereas, majority of scanners and testing tools are reactive and based on previously found knowledge.
- In short, fuzzing is a negative software (as opposed to feature testing) testing method that feeds malformed and unexpected input data to a program, device, or system. In negative testing, unexpected or semi-valid inputs or sequences of inputs are sent to the tested interfaces, instead of the proper data expected by the processing code.



# Fuzzing



- Sometimes other terms are used to describe tests similar to fuzzing. Some of these terms include:
  - Negative testing
  - Protocol mutation
  - Robustness testing
  - Syntax Testing
  - Fault injection
  - Rainy-day testing
  - Dirty testing
- Traditionally, terms such as negative testing or robustness testing have been used mainly by people involved with software development and QA testing, and the word fuzzing was used in the software security field.

# Fuzzing Process



---

- 1) Tests are first generated and sent to the SUT
- 2) A critical part of the fuzzing process is to monitor the executing code as it processes the unexpected input
- 3) Finally, a pass-fail criteria needs to be defined
- with the ultimate goal being to perceive errors as they occur and store all knowledge for later inspection.
- If all this can be automated, a fuzzer can have an infinite amount of tests
- Only the actual failure events need to be stored, and analyzed manually
- If failures were detected, the reason for the failure is often analyzed manually

# Why Fuzzing? What about others?



---

- One of the main strengths of fuzzing is that if an input crashes an application, a problem definitely exists in the application (no false positives).
- It should be noted that both source code audits and reverse engineering are (traditionally) a purely static method for understanding the operation (and misoperation) of a given application. However, actually executing the target for a few minutes can often yield more understanding than hours of reverse engineering.
- What if no understanding of an application was available and all we could do was supply input?
- What if, for whatever reason, when we supply a malformed input, the target crashes?

# DGF

What is Directed Greybox Fuzzing (DGF)?



# Basic Terms



---

- Coverage is an important term that is used in testing, and the same applies for fuzzing.
- From a vulnerability analysis perspective, coverage typically refers to simple code coverage—that is, how many lines of the existing source code or compiled code have been tested or executed during the test.
- A related term to coverage is attack surface: the amount of code actually exposed to an attacker.
- A trust boundary is any place that data or execution goes from one trust level to another, where a trust level is a set of permissions to resources.

# Directed Greybox Fuzzing



---

- Existing Greybox Fuzzers (GF) cannot be effectively directed, for instance, towards problematic changes or patches, towards critical system calls or dangerous locations, or towards functions in the stacktrace of a reported vulnerability that we wish to reproduce.
- In this paper, we introduce Directed Greybox Fuzzing (DGF) which generates inputs with the objective of reaching a given set of target program locations efficiently. We develop and evaluate a simulated annealing-based power schedule that gradually assigns more energy to seeds that are closer to the target locations while reducing energy for seeds that are further away.
- Experiments with our implementation AFLGo demonstrate that DGF outperforms both directed symbolic-execution-based whitebox fuzzing and undirected greybox fuzzing.

# Directed Greybox Fuzzing



---

- Greybox fuzzing (GF) is considered the state-of-the-art in vulnerability detection. GF uses lightweight instrumentation to determine, with negligible performance overhead, a unique identifier for the path that is exercised by an input.
- New inputs are generated by mutating a provided seed input and added to the fuzzer's queue if they exercise a new and interesting path.
- Unlike undirected fuzzers, a directed fuzzer spends most of its time budget on reaching specific target locations without wasting resources stressing unrelated program components.

# Applications of DGF



- Patch testing: by setting changed statements as targets. When a critical component is changed, we would like to check whether this introduced any vulnerabilities. Figure 1 shows the commit introducing Heartbleed. A fuzzer that focusses on those changes has a higher chance of exposing the regression.
- Crash reproduction
- Static analysis report verification
- Information flow detection



# Symbolic-execution-based Fuzzers



- Most existing directed fuzzers are based on symbolic execution. Symbolic execution is a whitebox fuzzing technique that uses program analysis and constraint solving to synthesize inputs that exercise different program paths.
- Suppose, in the control-flow graph there exists a path  $\pi$  to the target location. A symbolic execution engine can construct a path condition, a firstorder logic formula  $\varphi(\pi)$  that is satisfied by all inputs exercising  $\pi$ . A satisfiability modulo theory (SMT) solver generates an actual input  $t$  as a solution to the path constraint  $\varphi(\pi)$  if the constraint is satisfiable.

# Symbolic-execution-based Fuzzers



---

- Most existing directed fuzzers are based on symbolic execution. Symbolic execution is a whitebox fuzzing technique that uses program analysis and constraint solving to synthesize inputs that exercise different program paths.
- Suppose, in the control-flow graph there exists a path  $\pi$  to the target location. A symbolic execution engine can construct a path condition, a firstorder logic formula  $\varphi(\pi)$  that is satisfied by all inputs exercising  $\pi$ . A satisfiability modulo theory (SMT) solver generates an actual input  $t$  as a solution to the path constraint  $\varphi(\pi)$  if the constraint is satisfiable. Thus, input  $t$  exercises path  $\pi$  which contains the target.

# Directed Symbolic-Execution (DSE)

- Directed Symbolic-Execution (DSE)
- Suppose, we set Line 1480 as target and Katch found a feasible path  $\pi_0$  reaching Line 1465 as intermediate target. Next, Katch passes the constraint  $\varphi(\pi_0) \wedge (hbtype == TLS1\_HB\_REQUEST)$  to the constraint solver to generate an input that actually exercises the target location in Line 1480.

```
1455 + /* Read type and payload length first */
1456 + hbtype = *p++;
1457 + n2s(p, payload);
1458 + pl = p;
...
1465 + if (hbtype == TLS1_HB_REQUEST) {
1477 +     /* Enter response type, length and copy payload */
1478 +     *bp++ = TLS1_HB_RESPONSE;
1479 +     s2n(payload, bp);
1480 +     memcpy(bp, pl, payload);
```

# Disadvantages of DSE



- DSE spends considerable time with heavy-weight program analysis and constraint solving. At each iteration DSE:
- Identifies those branches that need to be negated to get closer to the target using PA
- Constructs the corresponding path conditions from the sequence of instructions along these paths
- Checks the satisfiability of those conditions using a constraint solver
- In the same time that a DSE generates a single input, a greybox fuzzer can execute several orders of magnitude more inputs.

# AFL GO



- Our directed greybox fuzzer AFLGo takes less than 20 minutes to expose Heartbleed while the DSE tool Katch cannot expose Heartbleed even in 24 hours!
- Focussed on reaching a given set of target locations
- DGF casts the reachability of target locations as optimization problem while existing directed (whitebox) fuzzing approaches cast reachability as iterative constraint satisfaction problem.
- Employ a specific meta-heuristic to minimize the distance of the generated seeds to the targets

# AFL GO



- To compute seed distance, we first compute the distance of each basic block to the targets.
- While seed distance is *interprocedural*, our measure requires analysis only once for the call graph and once for each *intraprocedural* CFG
- The meta-heuristic that DGF employs to minimize seed distance is called Simulated Annealing and is implemented as power schedule. A power schedule controls the energy of all seeds. A seed's energy specifies the time spent, fuzzing the seed.

# AFLGO VS. DSE



A brief comparison between these two methods in  
discovering vulnerabilities:

## AFLGO:

- More effective: DGF exposes more vulnerabilities
- More efficient: DGF reaches more targets in the same time
- Naive and simple approach

## DSE:

- Moves the analysis to the compile-time, as a result, it minimizes the overhead at runtime
- The analysis takes long for complex programs and languages



## AFLGO VS. DSE

A comparison between AFLGo and Katch, a DSE engine, on the original Katch benchmark:





# Heartbleed (CVE-2014-0160)



- Heartbleed (CVE-2014-0160) is a vulnerability which compromises the privacy and integrity of the data sent via a secure protocol (SSL/TLS).
- Suppose, Bob has a secret that the attacker Mallory wants to find out. First, Bob reads the message type and payload from Mallory's incoming message. If the message is of a certain type, Bob sets the type and payload of the outgoing message as his response. Finally, Bob copies payload many bytes from the incoming message (pl) to the outgoing message (bp). If less than payload bytes are allocated for pl, Bob reveals his secret.

# How Katch detects it?



---


- First, OpenSSL must be compiled to bytecode. Then, Katch processes one changed basic block at a time. For our example, Katch identifies 11 changed basic blocks as reachable target locations that are not already covered by the existing regression test suite.

# How Katch detects it?

- For each target  $t$ , Katch executes the following greedy search: Katch identifies a seed  $s$  in the regression test suite that is “closest” to  $t$ . For instance, an existing seed might execute the branch  $b_i$  in Line 1465 but contain an incorrect message type. This seed is close in the sense that only one branch needs to be negated to reach the target. Now, Katch uses program analysis: 1) to identify the executed branch  $b_i$  that is closest to the target  $t$ , 2) to construct a path constraint  $\Pi(s) = \varphi(b_0) \wedge \varphi(b_1) \wedge \dots \wedge \varphi(b_i)$  as a conjunction of every branch condition that  $s$  executes, and 3) to identify the specific input bytes in  $s$  it needs to modify to negate  $b_i$ . Then, Katch negates the condition of  $b_i$  in  $\Pi$  to derive  $\Pi' = \varphi(b_0) \wedge \varphi(b_1) \wedge \dots \wedge \neg\varphi(b_i)$ . The constraint  $\Pi'$  is passed to the Z3 SMT solver to compute the specific values of the identified input bytes such that  $b_i$  is indeed negated. In this case, the resulting incoming message would now contain the correct type (Line 1465) and execute the vulnerability at Line 1480.

# How Katch detects it?

- While directed symbolic-execution-based whitebox fuzzing is very effective, it is also extremely expensive. Due to the heavy-weight program analysis, Katch takes a long time to generate an input.
- Katch cannot detect Heartbleed within 24 hours

| CVE   | Fuzzer | Runs | Mean TTE | Median TTE |
|---|--------|------|----------|------------|
|  | AFLGo  | 30   | 19m19s   | 17m04s     |
|   | KATCH  | 1    | > 1 day  | > 1 day    |

# Disadvantages



- The search might be incomplete:
- Since the interpreter might not support every bytecode
- The constraint solver might not support every language feature, such as floating point arithmetic
- The greedy search might get stuck in a local rather than a global optimum and never reach the target
- Due to sequential search, Katch misses an opportunity to inform the search for other targets by the progress of the search for the current target; the search starts anew for every target

# AFL GO



- Requires virtually no program analysis at runtime and only lightweight program analysis at compile/instrumentation time.
- Implements a global search based on Simulated Annealing
- AFLGo implements a parallel search, searching for all targets simultaneously.

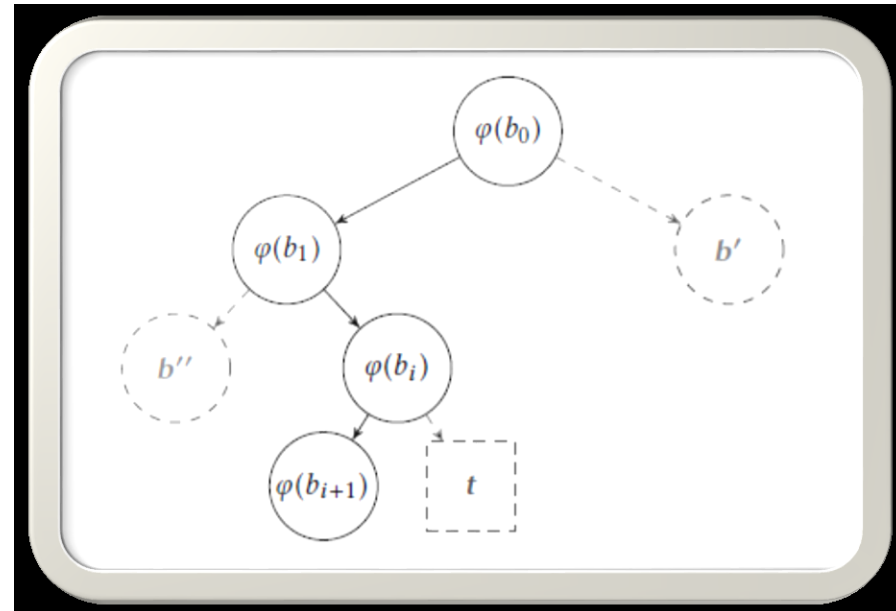
# AFL GO Mechanism



- In the exploration phase, AFLGo randomly mutates the provided seeds to generate many new inputs. If a new input increases the code coverage, it is added to the set of seeds to be fuzzed; otherwise, it is discarded

# AFL GO Mechanism

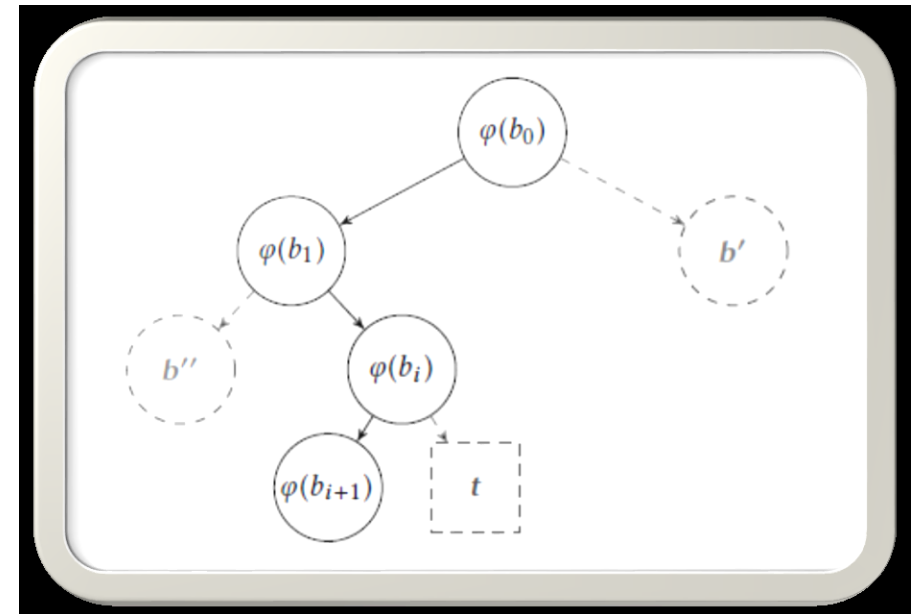
- **Example:**  $s_0$  exercising branches  $\langle b_0, b' \rangle$  and  $s_1$  exercising  $\langle b_0, b_1, b'' \rangle$ . Suppose, there is a direct path from  $b'$  to  $t$  that cannot be exercised by an input. At the beginning, roughly the same number of new inputs would be generated from both seeds. The rationale for the exploration is to explore other paths, even if longer.
- The time when AFLGo enters exploitation is specified by the user.





# AFL GO Mechanism

- AFLGo generates substantially more new inputs from seeds that are closer to the target—essentially not wasting time fuzzing seeds that are too far away.
- Suppose, at this point AFLGo generated a seed  $s_2$  that exercises the branches:  $\langle b_0, b_1, b_i, b_i + 1 \rangle$
- In the exploitation phase, most of the time is spent on fuzzing the seed  $s_2$  since it is closest to the target  $t$ .



# AFL GO Mechanism



---

- In the exploration phase, AFLGo randomly mutates the provided seeds to generate many new inputs. If a new input increases the code coverage, it is added to the set of seeds to be fuzzed; otherwise, it is discarded
- Defines an inter-procedural measure of distance (i.e., seed to target locations) that is fully established at instrumentation-time and can be efficiently computed at runtime
- While the measure is inter-procedural, the program analysis is actually intra-procedural based on the call graph (CG) and intra-procedural control-flow graphs (CFGs).

# Greybox Fuzzing



---

- Today, we distinguish three streams based on the degree of program analysis:
- *Black-box fuzzing* only requires the program to execute.
- *White-box fuzzing* based on symbolic execution requires heavy-weight program analysis and constraint solving
- *Greybox fuzzing* is placed in-between and uses only lightweight instrumentation to extract some program structure.

# Greybox Fuzzing



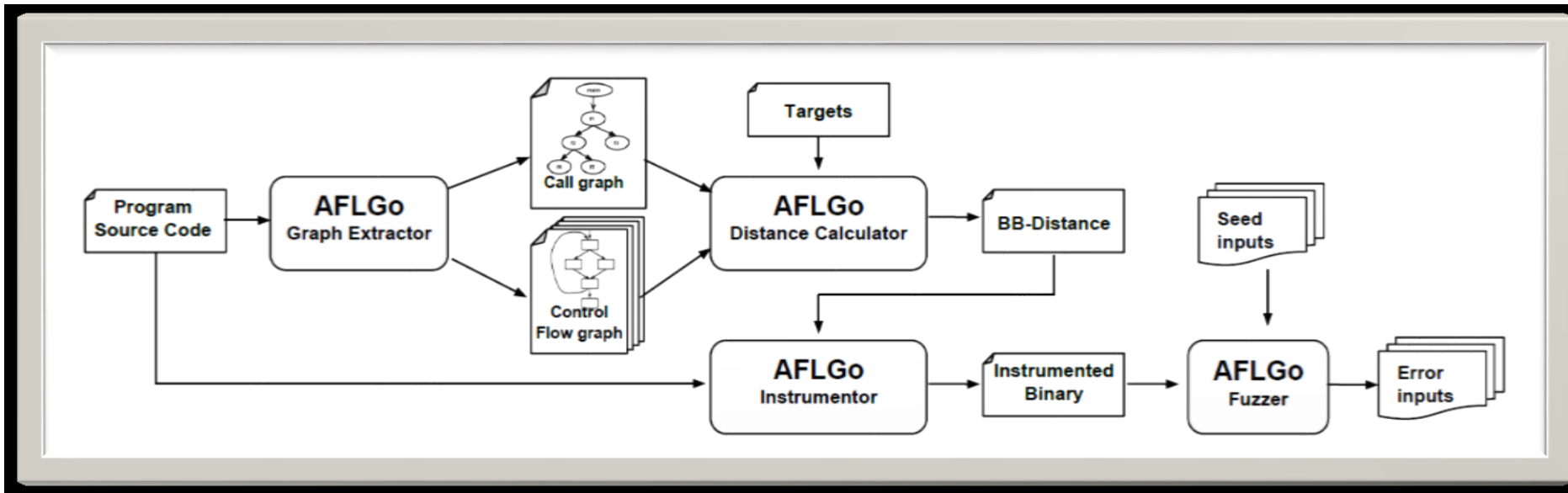
---

- Coverage-based greybox fuzzers like AFL and Lib-Fuzzer use lightweight instrumentation to gain coverage information. CGF uses the coverage information to decide which generated inputs to retain for fuzzing, which input to fuzz next and for how long.
- We extend this instrumentation to also account for the distance of a chosen seed to the given set of target locations. The distance computation requires finding the shortest path to the target nodes in the call graph and the *intra-procedural* control-flow graphs which are readily available in LLVM. The shortest path analysis is implemented as Dijkstra's algorithm.

# Greybox Fuzzing



- The fuzzer is provided with a set of seed inputs  $S$  and chooses inputs  $s$  from  $S$  in a continuous loop until a timeout is reached or the fuzzing is aborted.
- For the selected seed input  $s$ , the CGF determines the number  $p$  of inputs that are generated by fuzzing  $s$  as implemented in *assignEnergy*
- This is also where the (annealing-based) power schedule is implemented



- After the Graph Extractor generates the call and control-flow graphs from the source code, the distance calculator computes the basic-block-level target distance for each basic block which is used by the Instrumentor during instrumentation. The instrumented binary informs the Fuzzer not only about coverage but also about the seed distance.

# References



- Takanen, Ari, et al. Fuzzing for software security testing and quality assurance. Artech House, 2018.
- Böhme, Marcel, et al. "Directed greybox fuzzing." Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017.
- Chen, Hongxu, et al. "Hawkeye: towards a desired directed grey-box fuzzer." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018.
- Li, Jun, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey." Cybersecurity 1.1 (2018): 6.

# THANK YOU!

---

August 2019

Name:

**Amir Abas Kabiri Zamani**

Email:

**KabiriZamani@aut.ac.ir**

