M. Meier, D. Reinhardt, S. Wendzel (Hrsg.): Sicherheit 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 23

Distributed Evolutionary Fuzzing with Evofuzz

Fabian Beterke[1]

Abstract: This paper describes the design of a tool (called *Evofuzz*) that implements the technique of evolutionary (or coverage-guided) fuzzing in a scalable, distributed manner. The architecture, design-choices and implementation specifics of this tool are examined, explained and criticized. After outlining possible improvements and future work that is not yet completed, the paper finishes by presenting the results from fuzzing real-world programs and explains how to recreate them using the provided tool.

Keywords: Fuzzing, Fuzzer, Evolutionary, Distributed Computing, Evofuzz, Security Research, Memory Corruption

1 Motivation

After the concept of fuzzing [Mi88] was introduced by Barton Miller (University of Wisconsin) et al. in 1988, the techniques and its variations have led to the discovery of innumerable issues, particularly in the field of memory-corruption vulnerabilities. Since then, the technique has evolved drastically into numerous branches like the original random fuzzing, generative fuzzing and mutation-based fuzzing. While those techniques have been implemented, analyzed and evaluated numerous times before, the idea of guiding the fuzzing process (i. e. the search through the target-program's state-space for unexpected/undefined behavior) by code-coverage is relatively new, the first public mention being with the goal of exhaustively testing the program without a brute-force approach of simply generating every possible input. While evolutionary fuzzing has been proposed in 2007 already by DeMott et al. [DEP07], no public and general-purpose implementation has been available until in February 2014, M. Zalewski published American Fuzzy Lop [Za15] (subsequently referred to as AFL). AFL has since showed a tremendous story of success, uncovering hundreds of bugs in otherwise well researched and widespread software which is due to its careful balance between a capable concept of mutating data in an evolutionary fashion and maintaining a sense of pragmatism, e. g. in terms of always keeping performance (measured in runs of the target program per second) the major focus because more runs always mean a higher likelihood of discovering crashes. A restricting factor for this metric however is the single-threaded design of AFL: Concurrently running it on a few dozen machines can be a very cumbersome task since it can only be achieved by copying over temporary files from one node to another and running each process independently, and since the target programs are run on a single core only, no performance gains can be observed from the use of multicore systems. A customizable, distributed-by-design, evolutionary general-purpose fuzzer would solve those problems which motivated the author to start working on a scalable, coverage-guided fuzzing architecture.

[1] Universität Hamburg, SVS, Vogt-Kölln-Straße 30, 22527 Hamburg, 2beterke@informatik.uni-hamburg.de

2 Background: Evolutionary Fuzzing

While of course the ultimate measurement for a fuzzer is the number of unique crashes it found, there are a few other metrics to orientate at when designing a new fuzzer, the most important one being code-coverage. Since the likelihood of finding issues increases with the amount of actual code that is being tested, covering as much functionality as possible is a valid (and measurable) metric for the success of a fuzzer. One of the most prevalent questions when designing a new fuzzer is whether the testcases for the target program should be generated entirely from scratch or as mutations of already existing inputs – the so-called *corpus* [MP07] – to achieve a high coverage and find numerous bugs. For a general purpose fuzzer that needs no further description of the input data than the samples in the corpus, many different mutation strategies (like e. g. flipping bits, inserting or removing bytes and performing arithmetic operations on chunks of the input) are needed in order to be flexible enough to handle various formats well. If however no big and coverage-rich corpus is available, the options can become sparse very quickly and hand-writing a new grammar for each format to be (generatively) fuzzed is a very time-consuming task.

Another option emerges if the problem of fuzzing a program for memory-corruption bugs is seen from another angle: essentially, fuzzing is nothing else than a search through the state-space of the program for crashing states. The formerly mentioned metric of code-coverage can be seen as a distance function: The more functionality (coverage) a testcase triggers, the higher the likelihood of discovering vulnerabilities when mutating it. The term *evolutionary fuzzing* (coined by DeMott et al. with their talk at Defcon 2007 [DEP07]) refers to this technique which can be summarized by the following algorithm:

1. Seed input queue with corpus
2. Pop a single testcase from the queue, generate mutations and execute
3. If one of the mutated inputs led to execution of previously unexecuted code: add testcase to input queue
4. Repeat from 2. until input queue is empty

This algorithm has the property of exploring the target program's state-space through evolving the input given to the program. While of course, starting from a good corpus drastically improves the overall performance of the fuzzer (i. e. in terms of achieved input complexity vs. CPU cycles spent), an experiment [Za14] by M. Zalewski (the author of AFL) showed that even an absolutely meaningless input with basically no coverage can be used as a corpus for a good evolutionary fuzzer to produce coverage-rich inputs. In this extreme example, Zalewski used only one file containing the ASCII-string "hello" as an initial corpus for fuzzing the target, a JPEG decoder. After just one day of fuzzing, AFL generated a wide array of actually valid JPEG files that make use of a variety of JPEG features – an excerpt of this experiment's results is shown in Figure 1.

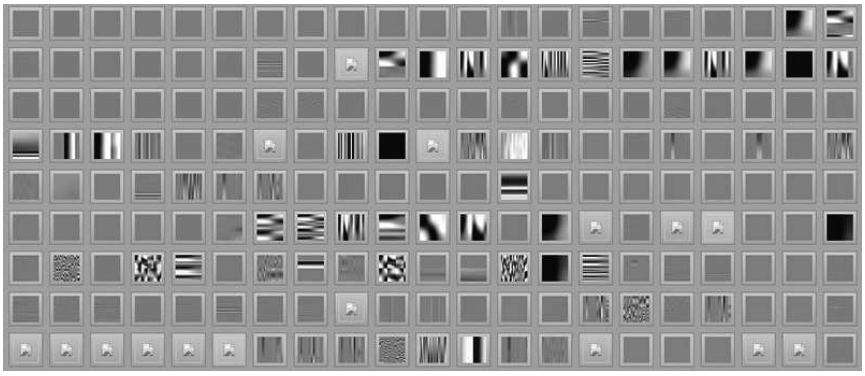


Fig. 1: JPEG images generated mutationally by AFL [Za14]

3 Measuring Code Coverage with LLVM's SanitizerCoverage

In order to develop such a coverage-guided fuzzer, it is crucial to gather data about the program's actual execution flow under a certain input, giving insight about the fuzzer's code-coverage. The LLVM Compiler-Suite offers a solution to this problem with a compile-time instrumentation called *Sanitizer Coverage* [Te15], or Sancov for short. At compile time, the tool's user has to choose between two relevant modes of code-coverage as they are offered by Sancov:

Basic-Block Level:
A widespread way to track coverage is to instrument every generated conditional jump instruction, effectively splitting the code into basic blocks of execution. Those blocks are the unit for the block level coverage method which can be illustrated by the following, C-like pseudocode:

```
1 int reset(int *x) {
2     if(x)            // A
3         *x = 0;      // B
4     return 1;        // C
5 }
```

While line 1 and 5 are irrelevant since they contain no control-flow altering logic, basic blocks in the assembly code generated from that snippet would be line 2 (A), 3 (B) and 4 (C). The comparison in block A takes place once the function is called, the write-operation in block B is dependent on the former comparison and the return instruction in block C is again executed unconditionally. Sancov offers output for this measurement in two different formats which may be chosen at runtime through an environment variable, either a list of instruction-pointers for basic-blocks (i. e. their first instruction's address) or a ASCII-bitmap where each index corresponds to a basic block and a 0 or 1 character indicates whether the block was executed or not. This level of instrumentation has a runtime overhead of around 30% on average, depending on the properties of the instrumented program.

Edge Level:
When imagining the execution flow as a graph of basic-blocks, a shortcoming of simple block-based coverage measurement emerges. While the coverage actually measured by block-level instrumentation relies on executed basic blocks (the graph's vertices), the edges in this graph represent state transitions which are more meaningful than pure basic blocks because a bug can seldom be found in a basic block alone but rather emerges from a faulty state transition. In the example code, without prior knowledge about the logic utilized by the program it is not clear whether a state transition from A to C happened only via B or directly, without invoking B first. Those edges are called critical edges, and by introducing dummy basic-blocks into those edges, the execution graph's edges can be instrumented analogous to basic-block coverage. This again increases the runtime overhead to 40% on average.

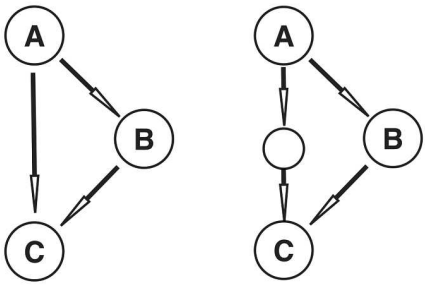


Fig. 2: Comparison between Block Coverage (left) and Edge Coverage (right)

4 Distributed Fuzzing: Problems and Challenges

The high-level organisation of the fuzzer follows a master-worker pattern in which a master-node is responsible for multiple workers, but multiple masters can be present if needed. While the master-node keeps the state of the overall fuzzing process, governs the testcase-queue and analyzes e. g. logfiles of crashes or code-coverage information, the worker-node's purpose is to execute testcases specified by the master and report the execution results back.

In order to avoid complex synchronization scenarios between a multitude of worker-nodes, one of the fuzzer's major design goals is to keep as little state as possible and do it on the master-node(s) only, wherever possible in the (itself distributed) persistent storage engine. This design allows the fuzzer to be scaled infinitely in theory. Since worker- and master-nodes are connected via a network-connection, one of the aims for a high throughput of target-program executions is to keep the network's bandwidth-utilization as low as possible. The problem that must be solved to achieve this is to actually produce the mutations on the worker-node itself and execute them right away while still centrally guiding the fuzzing process by e. g. specifying the testcase to be mutated, which mutation should be used as well as the amount of mutated testcases to produce.

5 Designing Evofuzz: Architecture and Implementation

Master

The master server is written in Python 2.7 and is responsible for central, stateful tasks. Its main functionalities are bundled into modules, each run in their own thread and communicate via a simple, message-passing based IPC implemented using the *threading.Queue* class. The persistence layer is built around Redis [SN10] (short for "Remote Dictionary Server"), an in-memory database with optional persistence to disk that offers simple but effective datastructures like e. g. hashes, lists and sets. Redis itself is highly scalable, on the one hand in a redundant way with many synchronized nodes propagating updates through the cluster to form a high-availability system with failover capabilities (called Redis Sentinel) and in a sharded fashion on the other hand (Redis Cluster), theoretically allowing the fuzzer to utilize up to $2^{14} = 16384$ nodes as one big, distributed in-memory database (this number is an artifact of Redis Cluster's implementation, however the authors recommend 1000 nodes as a reasonable upper limit for stable operation).

Notable modules of the master include the *ProvisionServer* which provides (target program) binaries, testcases and configuration to the workers on request and the *ReportServer* which generates a job description for the next chunk of work to be sent out to the worker while at the same time receiving their reports. The latter are redirected to the *ReportAnalyzer* module which is responsible for implementing the evolutionary fuzzing-logic of enqueuing mutations of testcases that lead to previously unexecuted code. Additionally, signatures of the newly found basic-blocks (or edges if that coverage method is used) are pushed to the *UpdatePublisher*, a caching layer that propagates those newly found signatures to the worker-nodes in order to prevent a high network utilization by useless re-submissions.

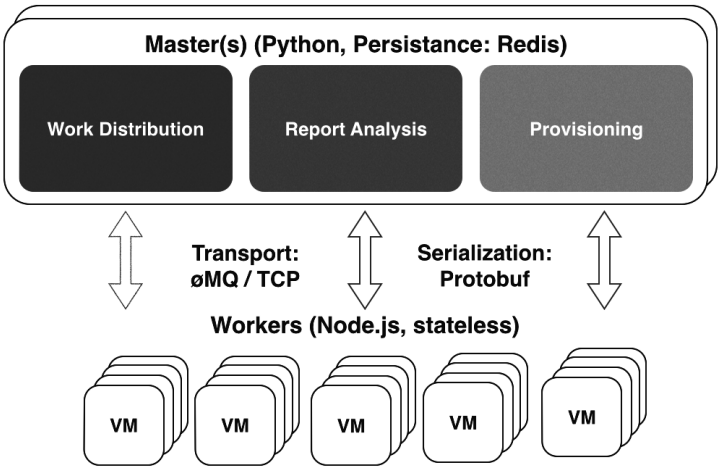


Fig. 3: Master Architecture and Overview

Worker

Since the main purpose of a worker node is to mutate data and actually run the target-program as fast as possible and at the same time, keep as little state as possible, the author chose to develop it in Javascript (JS) using NodeJS. The main reason for this choice is the comparably easy implementation of asynchronous, heavily callback-oriented actions: nearly all APIs to the underlying Operating System's features like spawning processes or doing network-/fileystem-IO are available as fully asynchronous versions by default since JS embraces this style of programming. Also, NodeJS is powered by Google's V8 JS interpreter which makes use of just-in-time compilation, leading to negligible performance disadvantages compared to native languages like C or C++ while being suited for explorative, rapid development due to JS being a memory-safe and loosely typed language.

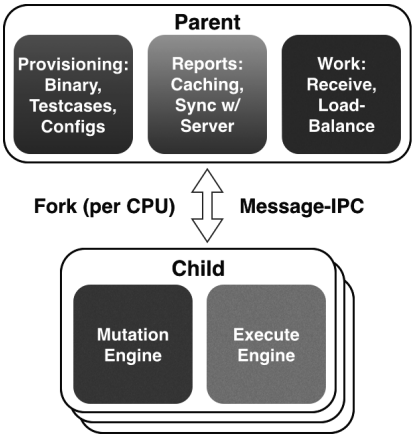


Fig. 4: Worker Architecture

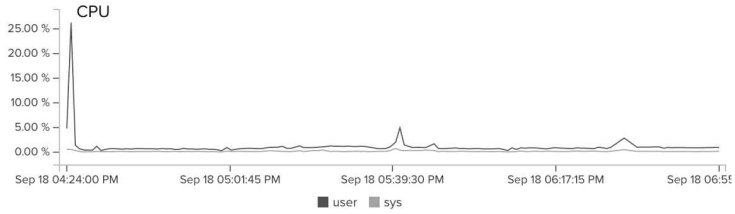
As shown in the figure above, the worker architecture is again separated into a parent and one or more multiple child processes. The parent process manages the connection to the master-node and initializes the fuzzing-loop by requesting a provision from the master, effectively setting up the (virtual) machine this process is running on by e. g. transferring the target program and it's command line arguments and environment variables from the master. Other responsibilities include buffering reports (coverage and crash information) before sending them to the server in chunks, subscribing to the *UpdatePublisher* and maintaining a cache to prevent redundant reports as well as load-balancing incoming work between children.

Those children-processes implement the actual generation of testcases, generating all possible mutations of a certain kind for a single testcase before starting to use the next mutation. Furthermore, those processes execute the target program with the mutated input, collect the Sancov-output and probe for the existence of a crashlog after each run and eventually send the results of those operations up to the parent process for reporting to the master.

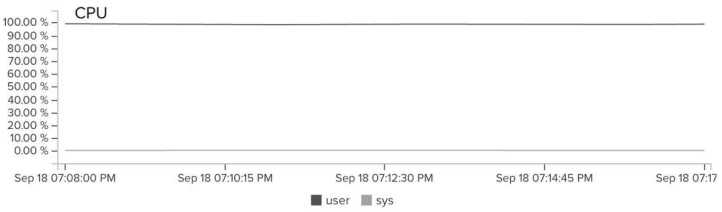
6 Performance and Scalability

The design outlined in the previous section allows the fuzzer to be scaled arbitrarily in theory while this has practical limitations of course (e. g. the resources of the machines used, the network connection between them or database capacities). The actual performance in terms of execution runs per second however is highly dependent on the inner loop, i. e. the execution of a single testcase. Because of the use of AddressSanitizer (another sanitizer from the LLVM suite that instruments memory accesses to detect e. g. out-of-bounds reads) and Sancov in conjunction with the lack of in-process fuzzing mechanisms for the sake of genericness, this performance-critical part of the fuzzing loop is not (yet) as optimized as it could be – currently, the fuzzer exhibits between 120 and 150 runs per second per core on average with a Intel Xeon CPU (E5-2630L) clocked at 2.4GHz (AFL for comparison reaches around 800 executions per second on this CPU but can only utilize one core), obviously varying wildly for different target programs. As stated previously, this shortcoming can be overcome easily by scaling horizontally, i. e. adding more worker nodes to the cluster due to the fuzzer's distributed design. The reason why this is a necessary requirement for a long-running fuzzer is that performance may not only be measured in runs per second or the percentage of basic-blocks discovered but also in terms of cost-effectiveness: It is a lot cheaper to rent 100 separate machines with 1 core each for an hour than it is to rent a single 100-core machine while at the same time, the latter is also more prone to error due to being a single point of failure. Furthermore, effectiveness is influenced by resource utilization: It is mandatory for the fuzzer to utilize all available ressources at all times, meaning 100% CPU utilization at every point in time for a worker node and preferably very little load on a master node so that new work may be handed out without latency, thus preventing idle times (as shown in Figure 5).

Fig. 5: CPU utilization on different node types



(a) Master (1 CPU, 512M, 30 Workers)



(b) Worker

7 Usage

Prior to configuring and starting the fuzzer, the user needs to compile the desired target with AddressSanitizer and Sancov enabled. This boils down to the use of 2 simple command line flags in a recent version of *clang*:
`clang target.c -fsanitize=address -fsanitize-coverage=X`
where `X` is replaced by a number to indicate which coverage instrumentation should be used, i. e. 2 for block-level and 3 for edge-level coverage. The actual usage of the fuzzer can be summarized into 3 high-level steps:

1. Initialize Redis with config and corpus using project-specific `initRedis.sh` (located in `misc/`)
2. Run `master.py` on master node
3. Start desired number of worker nodes and run `worker.js <master IP>`

This process can of course be automated for specific cloud-computing hosters, making use of their APIs for e. g. fully automated deployment of an arbitrarily sized fuzzing cluster. While the results outlined in this paper have mainly been produced by using *Digital Ocean*, a deployment on e. g. *Amazon Web Services* would work analogous. After a successful fuzzing run, crashes may be extracted from the database by simply running the utility in `misc/analyzeCrash.py`.

```
sh-3.2$ tar -xzf fuzz.tgz && cd fuzz && ls
README.md       master          misc            protos          worker
sh-3.2$ cd misc && sh initRedis.sh
OK
OK
(integer) 1
(integer) 1
(integer) 1
(integer) 1
sh-3.2$ cd .. && virtualenv master; cd master && . bin/activate; make &> /dev/null
New python executable in master/bin/python
Installing setuptools, pip...done.
(master)sh-3.2$ python master.py
[ProvisionServer] got PROVISION request
[ProvisionServer] got STATE request
[ControlServer] worker connected, total: 1
[ReportServer] initiating
[Bitflip] <DEBUG> initiated
[ProvisionServer] got TESTCASE request
[Bitflip] <DEBUG> next -> True
[Bitflip2] <DEBUG> initiated
[ReportAnalyzer] 1 new blocks/edges found!
[Bitflip2] <DEBUG> next -> True
[ReportAnalyzer] \o/ got a new crash \o/
[ReportAnalyzer] 1 new blocks/edges found!
[Bitflip4] <DEBUG> initiated
[Bitflip4] <DEBUG> next -> True
[Byteflip] <DEBUG> initiated
[Byteflip2] <DEBUG> initiated
[ControlServer] worker 26 disconnected, requeing 4 job(s)
^CTerminating, sending kill-signal...
Terminated: 15
(master)sh-3.2$
```

```
sh-4.3$ tar -xzf fuzz.tgz && cd fuzz
sh-4.3$ cd worker && sh prepareFS.sh && make > /dev/null
npm WARN package.json evoFuzz@0.0.1 No repository field.
npm WARN package.json evoFuzz@0.0.1 No README data
sh-4.3$ node worker.js 10.0.2.2
coverage initiated, pathlen: 9
Worker online
Worker online
Worker online
Worker online
bitflip
bitflip
worker #2 got work, 1000 testcases generated
worker #1 got work, 1000 testcases generated
bitflip
worker #3 got work, 48 testcases generated
bitflip2
worker #4 got work, 1000 testcases generated
bitflip2
worker #3 got work, 1000 testcases generated
got update! added 0 hashes
=== 183.06666666666666 runs/sec ===
received crash from worker 2!
got update! added 0 hashes
bitflip2
worker #2 got work, 48 testcases generated
bitflip4
worker #4 got work, 1000 testcases generated
bitflip4
worker #2 got work, 1000 testcases generated
bitflip4
worker #3 got work, 48 testcases generated
byteflip
worker #1 got work, 256 testcases generated
byteflip2
worker #3 got work, 256 testcases generated
^C
sh-4.3$
```

Fig. 6: Set up of the fuzzer + CLI interface while running

8 Results

The three main objectives of the evolutionary fuzzer as outlined in this paper are a design that enables it to be scaled horizontally on commodity hardware, the ability to discover more complex inputs leading to previously unseen behaviour as well as finding crashes, obviously. The design choices that allow the fuzzer to achieve the first of the mentioned goals have been explained in depth in the previous parts, the fulfilment of the second objective can be measured easily by keeping track of the number of discovered basic blocks in an exemplary target program (`libharfbuzz` in this case, an advanced font parsing library used e. g. in Google Chrome). Also, the general effects of lowering the time needed to trigger the execution of those new basic blocks by simply adding more workers to the cluster has been successfully demonstrated (see Figure 7, comparability is given since only deterministic mutations have been used for this measurement). In a few test runs conducted with various numbers of worker nodes, the fuzzer was able to discover crashes in numerous targets, including `libharfbuzz`, `libfreetype`, `speexdec` and `p7zip` just to name the most important ones. It has been shown that the tool fulfills all of it's objectives in a sufficient, though still improvable way.

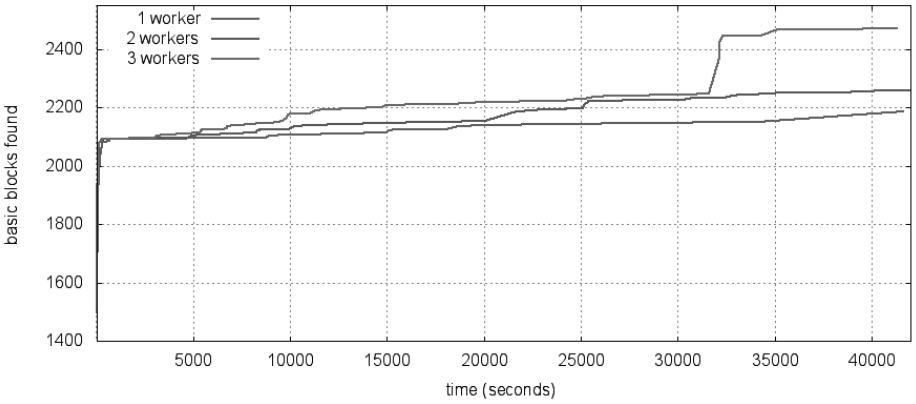


Fig. 7: Discovery of basic blocks with different numbers of workers in `libharfbuzz`

9 Criticism and future work

A few known issues in the actual implementation (rather than the design) exist at the current point of development that have not yet been tackled because of time constraints. The main problem the author currently faces in terms of performance improvements is the inner-loop optimization in order to make the fuzzer yield more executions per second per core: the possibilities of the implementation in Node.js have been exhausted to a maximum, meaning that further improvements like generic in-process fuzzing or forkless execution as well as saving copy- and process-spawning overhead would need a rewrite of the worker-implementation in C/C++ or another hardware-close language that compiles to native code and allows more fine granular access to the underlying operating system's API.

Another major improvement would be automatic testcase minimisation, the so called pruning: once a testcase leading to new behaviour is found, it could be repeatedly trimmed in size at different positions while examining which is the smallest mutation (by number of bytes) that still leads to the exact same behaviour of the target program.

An improvement that was planned but not yet implemented is the automated reproducibility check for crashes: A crash that can not immediately be reproduced is worthless for the fuzzing process since there may be numerous indeterministic reasons why a program could crash without the crash being related to the actual input which is why those inputs should be instantly discarded rather than having the user figure out which crashes can be actually reproduced. Finally, the software could be made more usable by a more intuitive interface, e. g. a Web-Interface.

10 Conclusions

The author proposed a design for a distributed, evolutionary fuzzing solution that may be used with arbitrary target programs (as long as they are available in source code and compile in clang / clang++) and as such, is generic. An upper limit for its scalability is yet to be found and the fuzzer behaved as expected even in tests with more than 50 worker nodes, yielding over 100,000 executions per second in total. The tool introduced in this paper may be used by developers or security researchers alike to fuzz for security bugs, even if no sophisticated corpus is available for the target program. Also, the codebase is kept quite small as well as comparably clean and modular, enabling interested parties to improve or customize the solution for their individual needs and expectations.

References

[DEP07] DeMott, Jared; Enbody, Richard; Punch, William: , Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. `https://www.blackhat.com/presentations/bh-usa-07/DeMott_Enbody_and_Punch/Presentation/bh-usa-07-demott_enbody_and_punch.pdf`, 2007. Talk at Blackhat.

[Mi88] Barton Miller: Random fuzz testing. `http://pages.cs.wisc.edu/~bart/fuzz/`, accessed 8.12.2015.

[MP07] Miller, Charlie; Peterson, Zachary N. J.: , Analysis of Mutation and Generation-Based Fuzzing, 2007.

[SN10] Salvatore Sanfilippo: Redis. `http://redis.io`, accessed 8.12.2015.

[Te15] The Clang Development Team: Sanitizer Coverage. `http://clang.llvm.org/docs/SanitizerCoverage.html`, accessed 8.12.2015.

[Za14] Michal Zalewski: Pulling JPEGs out of thin air. `http://lcamtuf.blogspot.de/2014/11/pulling-jpegs-out-of-thin-air.html`, accessed 8.12.2015.

[Za15] Michal Zalewski: American Fuzzy Lop's technical details. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, accessed 8.12.2015.