FuzzM: A Model-Based Approach to Grey-Box Fuzzing

Ryan Coppa
Rockwell Collins

Grant Foudree
Rockwell Collins

David Greve
Rockwell Collins

Abstract

Fuzz testing is a form of automated robustness testing that employs random, invalid or unusual inputs to search for unknown and potentially exploitable system behaviors. Traditional fuzz testing techniques do not describe internal system behavior, instead they rely mutations of input sequences driven by code coverage metrics. These techniques lack the ability to explore deep system state. In this paper we describe model-based fuzzing, a fuzzing technique that utilizes both a mathematical model to guide the fuzzing process and a constraint solver to deduce high-quality tests capable of targeting deep system behaviors that random testing alone would be unlikely to reach. We describe the model-based fuzzing framework FuzzM, demonstrate how it can be used to model a simple system, and compare its performance with several off-the-self fuzing solutions.

1 Introduction

Fuzzing is a form of robustness testing in which random, invalid or unusual inputs are applied while monitoring the overall health of the system. Health monitoring may include detecting crashes, exceptions, lock-up, reduced throughput, runaway memory usage, or excessive power consumption. Such anomalous events may be indicative of potentially exploitable behaviors in the system. The use of overall system health as a testing oracle, rather than a specific expected functional response, distinguishes fuzz testing from more traditional testing approaches.

Historically, fuzzing has been successful in finding bugs [1, 2]. The efficacy of simple random fuzzing, however, is often limited by software well-formedness checks that are unlikely to succeed on random data. CRC checks on Ethernet packets, for example, are unlikely to be correct by chance thus most packets generated at random are likely to be immediately discarded. Instead, generational fuzzers allow the user to specify data format templates and to compute essential content such as CRCs programmatically [4, 6]. Random (but well structured) inputs are then constructed by the fuzzer by filling in these data templates. More complex protocol behaviors can then be implemented in an ad-hoc manner. The implementation details of the ad-hoc implementations, however, could be implemented with less rigor and would benefit from fuzzing.

More recently, instrumented fuzzing has emerged as a new technique to direct input transformations to better satisfy complex software validation functions. Popular instrumented fuzzers include AFL [18] and Honggfuzz [11]; each generates inputs by leveraging additional analyst-provided program information. For example, AFL employs compile-time instrumentation to capture coverage information. This feedback is then used in conjunction with the AFL fuzzer to guide new input mutations, with the goal of driving broader software covereage. While such feedback-driven fuzzing techniques have demonstrated success, the mutation strategy still relies on generating massive amounts of mutated input over time. Consequently these techniques are slow and ultimately reliant on some degree of luck to discover all execution paths.

To address these issues, new hybrid fuzzing strategies have sought to leverage symbolic execution and taint analysis techniques to find deeper software flaws [14–16]. For instance, Driller [16] uses selective concolic execution to force the exploration of new program paths when the fuzzer's mutation mechanism fails to randomly discover such paths. Similarly, Vuzzer [15] uses data-flow and control analysis to infer fundamental features of an application to help inform the fuzzer's mutation strategy. While these new fuzzing strategies demonstrate significant improvement over conventional instrumented fuzzers such as AFL in the Cyber Grand Challenge [8], limitations still persist. Input generation based on symbolic execution alone often fails to explore deep paths [17]. For example, Driller fails to find vulnerabilities in many of the Cyber Grand Challenge datasets. Furthermore these techniques still assume complete and unfettered access to the software binary for the purpose of analysis and instrumentation, which may not necessarily be possible in all vulnerability analysis cases.

In this paper we describe model-based fuzzing, a technique that employs a mathematical model of system behavior to guide the fuzzing process. Whereas a smart fuzzing framework constructs fuzz tests by filling in data structure templates, a model-based framework deduces tests from a behavioral model using a constraint solver. Coverage metrics and model features are used to formulate test objectives that are then expressed mathematically as logical constraints. The constraints and the model are passed to a constraint solver which in turn deduces an input that will cause the device under test to exhibit the desired behaviors. The use of a constraint solver in this manner enables the creation of high-quality tests capable of targeting deep system behaviors that random testing alone would be unlikely to reach.

2 Model-Based Fuzzing Leveraging Lustre Models

FuzzM is a model-based fuzzing framework that leverages Lustre as a modeling language and the JKind [9] model checker as a constraint solver. Lustre [13] is a formally defined, declarative, synchronous dataflow language useful for modeling reactive systems. JKind is an SMT-based infinite-state model checker for safety properties expressed in Lustre. FuzzM is not designed to fuzz or search for errors in Lustre models. FuzzM is intended to fuzz systems whose abstract behaviors are described by Lustre models. It generates actual tests with the intent of applying those tests to an operational system while monitoring that systems health in search of implementation vulnerabilities. FuzzM is also not designed to test a system for conformance with a Lustre model. Rather, FuzzM treats the Lustre model as an abstraction of the underlying system behavior and utilizes it to guide the fuzzing process (by either targeting or avoiding specific features) in an attempt to direct exploration the implementation state space.

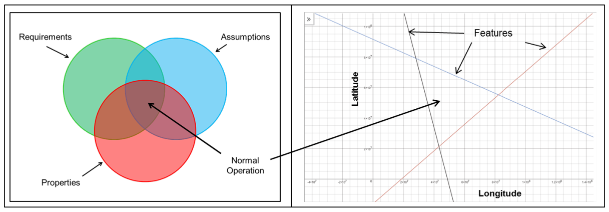

Figure 1: Conceptual feature Venn Diagram and Example Linear Features

The model-based fuzzing philosophy is to target behaviors that are known and to fuzz behaviors that are unknown. A feature is a Boolean expression that appears in a model or as part of a constraint. Features are important from a testing perspective because they typically partition the behavior of the system in interesting ways. In treating the model as a guide for exploring the behavior of the system we assume that, if there is a flaw in the system, it is likely to be near a model feature boundary. This approach extends the concept of boundary value testing. In boundary value testing, testing effort is focused on edge conditions and corner cases by employing test cases that explore extreme or boundary values from the input domain. We extend the notion of boundary value testing to include sets of inputs that just barely satisfy or violate arbitrary model features. For instance, if an array is defined as being 10 bytes in the length, the fuzzer should explore values slightly larger and slightly smaller than 10. Boundary value fuzzing means that we attempt to select random input vectors from a distribution that favors vectors close to model features. Figure 1 illustrates both conceptual features in a Venn diagram and actual model features from a pedagogical example in which a triangular region in the x-y plane is defined by the intersection of three linear features.

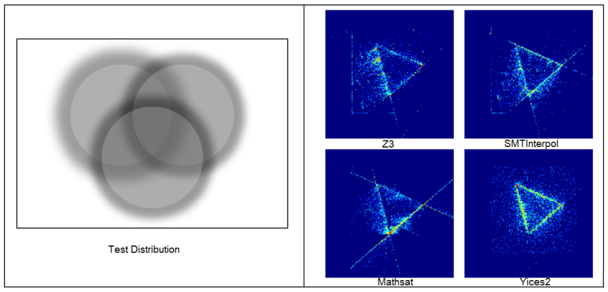

Figure 2: Conceptual and Measured Test Distributions

Figure 2 shows both conceptual and measured statistical distributions of tests around the model features generated by four different solvers. The heat map distributions in Figure 2 are generated by plotting thousands of solver solutions to random constraint queries involving arbitrary combinations of the three linear features from

Figure 1. Note that the test distributions are attracted to but distributed around the feature boundaries. Model based fuzzing uses a constraint solver to target behaviors that are known (the model features) and then fuzzes around them to search for behaviors that are unknown (behaviors not captured by the model).

3 Architecture

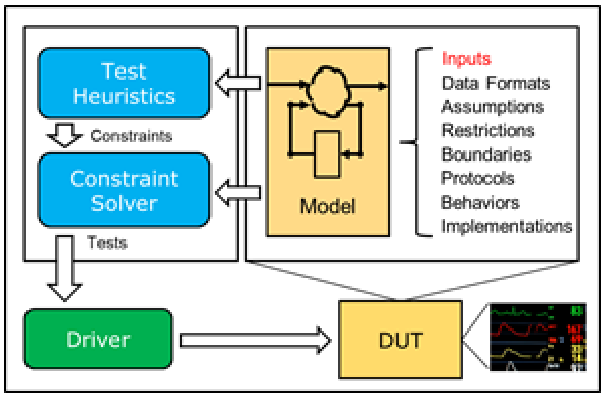

Figure 3: Model-Based Fuzzing Architecture

Figure 3 shows the overall architecture of a model-based fuzzing framework. A model that describes various features of the Device Under Test (DUT) drives the process. The model is processed by the Test Heuristics where coverage metrics and model features are used to generate test objectives that are expressed mathematically as logical constraints. The constraints and the model are passed to a constraint solver that deduces an input that will cause the DUT to exhibit the desired feature. The deduced input is passed to a driver interface that has been integrated with a target. The driver converts the fuzzer output into an input that is broadcast to the DUT, which has been augmented with health monitoring facilities capable of detecting and recording system failures. The overall test generation process is intended to run continuously.

4 The Role of Generalization

FuzzM employs coverage directed testing to formulate constraints that drive the solver to visit states that random testing alone would be unlikely to reach. While progress against traditional coverage metrics can be measured against the finite Lustre model of system behavior, the objective of fuzz testing is to explore the behavior of a system beyond the model to search for otherwise unknown vulnerabilities. Because the unknown state space being explored by the fuzzer is likely to be substantially larger than the model state space, effective exploration of that space requires the rapid generation of large numbers of test vectors. Unfortunately, the need to generate tests quickly is antithetical to using a constraint solver for test generation [7,10]. Even for relatively small models, a single invocation of JKind requires approximately one second of execution time.

Also, it is surprisingly difficult to induce JKind to produce random solutions. While JKind supports the SMT-LIB random-seed option, random seeds dont have a significant impact on the resulting solutions. Rather, solutions tend to cluster around model and constraint features or near special values like zero. To avoid clustering specially crafted hypotheses can be employed to drive the solver away from previous solutions. The presence of large, complex hypotheses, however, can easily degrade solver performance.

Rectilinear Generalization helps to meet our model-based fuzzing objectives by decoupling the constraint solver from the test generation process. Generalization is a technique that converts a single constraint solution into a set of solutions that cover the state space in the region of the original solution. Given a generalized solution, test generation simply involves sampling the solution set. This sampling process is amenable to efficient implementation and is capable of rapidly producing large numbers of high-quality tests from a single solver invocation. The sampling of the solution set can also be randomized, shifting much of the onus of generating desirable, random test distributions from the solver to the sampler. Under this configuration the JKind solver can be directed towards producing solutions that target hard to reach model features while generalization can be relied upon to provide the bandwidth needed to explore the state space around those solutions in search of proximate vulnerabilities. Using *Trapezoidal Generalization* [12] FuzzM is able to magnify a JKind solution stream of 1 vector per second into a fuzzing stream of nearly 2000 vectors per second.

5 Application of FuzzM

The following section outlines the two steps required to deploy FuzzM against a new target. First, a Lustre model of the target must be implemented. Second, a relay must be constructed to translate between the fuzzer and the target interface.

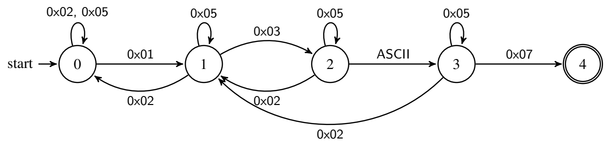

Figure 4: FSM Diagram

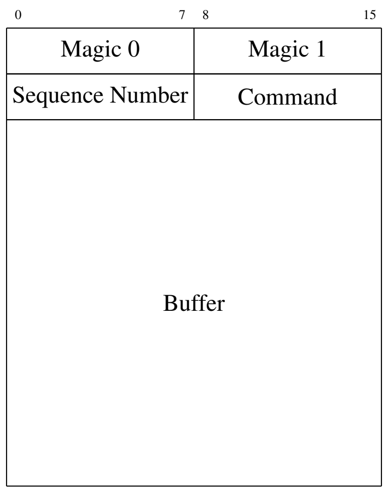

Figure 5: FSM Packet

Figure 4 represents a fictional finite state machine (FSM) implementation, with the transitions annotated with valid payload values. In order to transition state, a properly constructed sequence of inputs must be produced, else state will reset to 1. Upon successful entry into state 4, a synthetic vulnerability (segmentation fault) is triggered causing the executable to exit. Figure 5 illustrates the FSM packet structure. The first step for FuzzM usage is to capture the target message structure, which is accomplished using Lustre as follows:

```
type byte    = int;
type fsm_msg = struct
{
  magic0 : byte;
  magic1 : byte;
  seq    : byte;
  cmd    : byte;
  buff   : byte[16]
}
```

Once the message structure is captured, individual packet values can be constrained. For instance, this fictional FSM requires two magic bytes to remain constant. Expressing this requirement in Lustre involves the following code:

```
magic0_ok   = (msg.magic0 = 170) ; -- 0xaa
magic1_ok   = (msg.magic1 = 187) ; -- 0xbb
```

More complex interactions can also be modeled using operators indicating time-based relationships. The following section of Lustre represents valid next state transitions based on the current state:

```
function st0() returns (y: int);

next_st = (if (cmd_reset) then 0 else
if (st0 and st0_ok) then
   (if (cmd_hello) then 1 else 0) else
if (st1 and st1_ok) then
   (if (cmd_data)  then 2 else 1) else
if (st2 and st2_ok) then
   (if (cmd_file)  then 3 else 2) else
if (st3 and st3_ok) then
   (if (cmd_disco) then 4 else 3) else
0);

st = st0() -> (pre next_st);
```

As our fictional FSM was developed with a standard IP/UDP interface, the relay can be constructed using standard Python libraries, as shown below:

```
length = int(test_vector['length'])
msg = bytearray(length)
if (0 < length):
   msg[0] = int(test_vector['msg.magic0'])
if (1 < length):
   msg[1] = int(test_vector['msg.magic1'])
if (2 < length):
   msg[2] = int(test_vector['msg.seq'])
if (3 < length):
   msg[3] = int(test_vector['msg.cmd'])
for index in range(0,length-4):
   name = 'msg.buff[' + str(index) + ']'
   byte = int(test_vector[name])
   msg[4 + index] = byte

sock.sendto(msg,
           (target_ip, target_port))
```

The Lustre model and relay can be found at the GitHub page for FuzzM, located in the Availability section below.

The fictional FSM executable was evaluated using FuzzM, AFL, and Honggfuzz. Using the previously described Lustre model, FuzzM located the vulnerable section of code after 82,000 samples. Both AFL and Honggfuzz (running in persistent mode) were unable to locate the vulnerable section after 25,000,000 and 100,000,000 samples respectively.

6 Comparing FuzzM Performance

In order to demonstrate the capabilities of FuzzM, a comparison including several open-source fuzzers was performed on a complex embedded target. Each fuzzer was evaluated on their speed and code coverage, as well as relevant features and usability. Our selection was comprised of two function-based fuzzers, AFL [18] and Honggfuzz [11], along with two network fuzzers, Boofuzz [3] and Radamsa [5]. Function-based fuzzers operate by redefining the entry-point of a program to provide a fuzz vector as a parameter to be used as input to the program. Upon execution, the fuzzer repeatedly invokes this

	FuzzM	Honggfuzz	AFL	Radamsa	Boofuzz
Speed (tests/sec)	277	43590	14500	10	10
Line Coverage	4.4631%	3.6900%	3.7195%	3.6543%	3.5342%
Branch Coverage	1.6464%	1.3174%	1.3631%	1.3100%	1.2542%

Figure 6: Quantitative Comparison

user-defined function with various inputs it intelligently generates by mutating a set of valid, example data, in order to explore new sections of code. In comparison, the network fuzzers operate by interpreting a protocol template with various constraints, electing to fuzz specific fields defined in the specification by sending the vectors over a UDP socket to the target.

Since most programs contain a significant amount of branches, an effective fuzzer must be able to deduce the proper inputs that satisfy complex graphs of branch conditions in order to test the entire program. In order to accomplish this, FuzzM derives its tests from a series of intricate models created in Lustre. In contrast, AFL, for example, inserts compile-time instrumentation between blocks of code in order to record which inputs lead to an undiscovered block of code, building a map of execution drivers over time [18]. Network fuzzers, being detached from the host and oblivious to the source code, are unable to monitor the program's execution directly, and consequently are incapable of dynamically altering their execution drivers to discover new blocks of code without additional tools.

Our testing scenario consisted of running each fuzzer independently for one hour, followed by gathering the coverage data and interpreting the results. The functional fuzzers operate over a corpus of valid messages, this corpus was generated by a sample of valid inputs to the system captured during normal operation. The FuzzM model was derived from software requirements imposed on the target. All of the tests were performed from a Docker container that included all the dependencies in order to maintain a consistent testing environment. To address the issue of having multiple entry-points for fuzzer data (network vs. function-based), a common input interface was created. With regard to the network fuzzers, we had difficulty allowing them to run at full throttle and unfortunately had to limit their speed by inserting a 100ms delay between fuzz vectors. Consequently, it is reasonable to assume that this likely altered our results as the function fuzzers were not subject to this restraint and were able to fuzz the program a significantly higher amount of times.

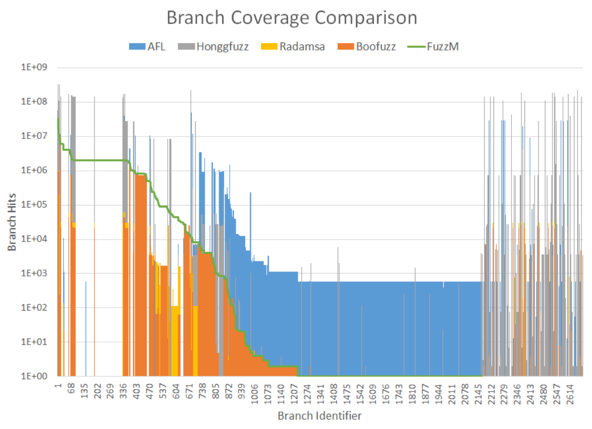

Figure 7: Branch Coverage Comparison

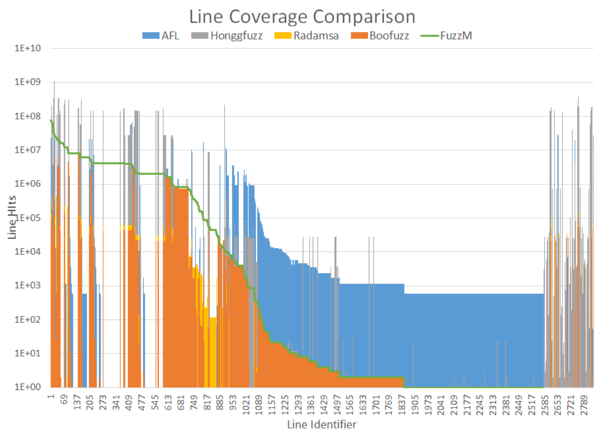

Figure 8: Line Coverage Comparison

Figure 7 illustrates the number of times each unique code branch was executed (note the logarithmic scale). The green line represents FuzzM performance while each colored bar represents a different open source fuzzer. Given the intimate relationship between function-based fuzzers and code, large coverage is expected. Of note are the areas where FuzzM executed a large number of branch hits, as opposed to small or zero counts for the others. Figure 8 illustrates similar results in the context of line execution count.

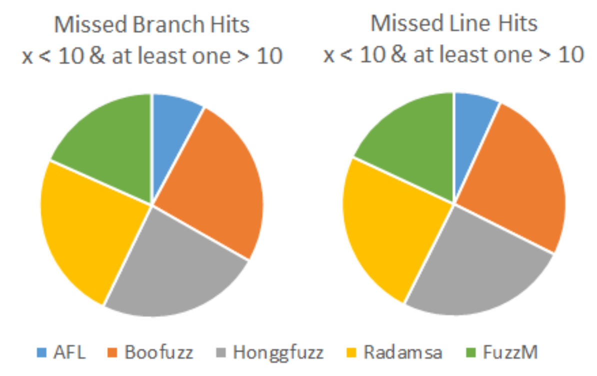

Figure 9: Missed Coverage Comparison

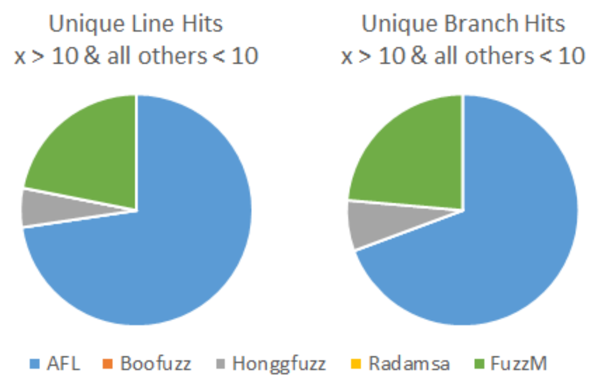

Figure 10: Unique Coverage Comparison

Further analysis of the coverage metrics reveal the function-based fuzzers and model-based fuzzing occupy complementary compartments within the fuzzing domain. As illustrated in Figure 9, while AFL has the fewest misses (defined as for a given branch, the count was less than 10 hits while at least one other fuzzer had greater than 10 hits), it still missed greater than zero while the other fuzzers were evenly distributed. In addition, Figure 10 illustrates AFL again with the most with FuzzM still claiming approximately 25% of the unique hits.

7 Conclusion

Based on performance comparison results, FuzzM demonstrates unique proficiency in targeting areas of software unreachable by state of the art fuzzers. This ability is provided by properly constructed Lustre models based on knowledge of the target, as opposed to driving fuzzing behavior based on coverage metrics and a representative set of inputs (AFL, Honggfuzz) or mutation/randomness (Radamsa, Boofuzz). As expected, overall coverage performance lies between functional/white box and black box fuzzers.

The strengths of FuzzM can be leveraged in environments where full access to implementation artifacts (e.g. firmware images, binaries, etc) is not possible, such as embedded systems. In this case, employing model-based fuzzing will provide a higher rate of vulnerability detection as opposed to traditional network fuzzers. The benefit is derived from the information contained in the FuzzM models, allowing the fuzzer to intelligently explore the target state space.

8 Future Work

The Lustre model generation and refinement process currently relies entirely on an operator. Initial model generation could be derived from observed target data captures, relieving the operator of manual construction. Incorporating feedback from the target could be used to drive model constraints, enabling a more refined representation of target behavior. In addition, if a sufficient observables are available, the fuzzer could be driven to explore new or desired functionality of the target.

9 Acknowledgements

This work was sponsored by DARPA/AFRL Contract FA8750-16-C-0218. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited

10 Availability

Complete source code for FuzzM, including all examples, is available via GitHub from:

`https://github.com/collins-research/FuzzM`

References

[1] Annonuncing oss-fuzz. `https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html`.

[2] Announcing project springfield. `https://blogs.microsoft.com/ai/microsoft-previews-project-springfield-cloud-based-bug-detector/`.

[3] Boofuzz. `https://github.com/jtpereyda/boofuzz/`.

[4] Peach fuzzer. `http://www.peach.tech/products/peach-fuzzer/`.

[5] Radamsa. `https://github.com/aoh/radamsa/`.

[6] AITEL, D. An introduction to spike, the fuzzer creation kit.

[7] CADAR, C., DUNBAR, D., AND ET AL., D. R. E. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI 8* (2008), 209–224.

[8] CGC, D. Darpa cyber grand challenge binaries. `https://github.com/CyberGrandChallenge`.

[9] GACEK, A. Jkind - an infinite state model checker for safety properties in lustre. `https://github.com/agacek/jkind`.

[10] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. sage: whitebox fuzzing for security testing. *Queue 10*, 1 (2012), 20.

[11] GOOGLE. Honggfuzz. `https://github.com/google/honggfuzz/`.

[12] GREVE, D., AND GACEK, A. Trapezoidal generalization over linear constraints.

[13] HALBWACHS, N., CASPI, P., RAYMOND, P., AND PILAUD, D. The synchronous data flow programming language lustre. *Proceedings of the IEEE 79*, 9 (Sep 1991), 1305–1320.

[14] LI, Y., CHEN, B., AND ET AL, M. C. Steelix:program-state based binary fuzzing. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), 627–637.

[15] RAWAT, S., JAIN, V., AND ET AL, A. K. Vuzzer: Application-aware evolutionary fuzzing. *NDSS* (2017).

[16] STEPHENS, N., GROSEN, J., AND ET AL, C. S. Driller: Augmenting fuzzing through selective symbolic execution. *NDSS 16* (2016), 1–16.

[17] WANG, X., ZHANG, L., AND TANOFSKY, P. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. *ISSTA* (2015), 199210.

[18] ZALEWSKI, M. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`.