Fuzzing Enhancements

Will Bergen

***Abstract*—Fuzzing is a software testing process which has grown immensely in popularity over the past 30 years. It has grown from a naive, yet powerful, completely random process into what's become an advanced area of research in which many different technologies are combined to form sophisticated, intelligent fuzz testing systems. In this work, we will canvas this development, focusing on the inclusion of enhancements on the basic purely random idea, and on how the effectiveness of fuzzing is measured.**

To this end, we have developed a review protocol and carried this out on papers related to fuzzing. Through this work, we've identified trends within this area of research, and have gained a better idea of which currently implemented enhancements are particularly popular, or effective, and why.

I. Introduction

Fuzzing is a form of software testing in which random data is generated and used as input for the system under test (SUT). The idea was first proposed by Miller et al. [1] in 1990 who generated random strings and used these as input for UNIX command-line programs. Symbolic execution, an alternative software testing method under which conditional statements within a program are considered functions to be solved, and which thus allows a program's state space to be systematically explored was the first major enhancement applied to the idea of fuzzing. In 2005, Godefroid et al. [2] incorporated symbolic execution into a tool they named DART (Directed Automated Random Testing) which utilized it to prevent the production of inputs which exercised already exercised code paths. Since then, scores of papers have been published further enhancing fuzzing with technologies ranging from dynamic taint analysis (DTA) to machine learning (ML), and many open-source fuzzers have been developed. Since Miller's 1990 paper, fuzzing has evolved into a complex landscape, with technologies and fuzzers now widely varied. Some important distinctions and classifications are as follows.

A. Mutational vs. Generational

One way to categorize fuzzers is by how they produce their inputs. Generally, mutational fuzzers require some kind of base-case - perhaps well-formed with respect to the SUT - which they then mutate to produce their inputs. Generational fuzzers on the other hand generate their inputs from scratch. They make use of some type of model or 'grammar' which provides knowledge about the SUT's input expectations, thus allowing the creation of valid inputs.

B. {White,Grey,Black} Box Spectrum

Another way to classify fuzzers is by their knowledge of the SUT. A black-box fuzzer assumes no knowledge of the SUT, while a white-box fuzzer has full knowledge. In between these two extremes lies grey-box, in which some knowledge of the SUT is available. Some fuzzing techniques, like IoTFuzzer [3], learn about a protocol as it goes, which over the course of testing moves the fuzzer away from a black-box situation toward a whiter one. There are also many grammar producing techniques, like Learn & Fuzz [4], Skyfire [5] and Bastani et al.'s 2017 work [6], which together make a more fine-grained taxonomy of this spectrum difficult.

C. Purpose

Not all fuzzers are designed with the same goals in mind. Over time, a clear distinction has emerged between three classes of fuzzers based on their purpose. Directed fuzzers attempt to exercise particular portions of code such as newly patched pieces as in AFLGo [7] and SemFuzz [8], or locations likely vulnerable to specific issues as in RAZZER [9] and TIFF [10]. Coverage-based fuzzers attempt to exercise as much code as possible [3, 11–14]. Finally there are fuzzers designed to reveal specific types of bugs [9, 10, 15, 38].

Given this complex landscape, we've undertaken the task of analyzing the existing literature on fuzzing to develop a better understanding what the current state of the field of fuzzing looks like. To this end, our paper makes the following contributions.

1) We analyze the currently implemented fuzzing enhancements to develop an overall view of the field.
2) We analyze the metrics by which the effectiveness of those enhancements has been measured.
3) We consider the potential of those enhancements, and consider particularly promising enhancements, and combinations of enhancements.

II. Background

An important feature of the modern fuzzing landscape is the presence of popular, open-source fuzzers which though not necessarily the output of academia have had a powerful impact on the field. At present, the most prominent is American Fuzzy Lop (AFL) [11]. AFL at its core is a coverage-based fuzzer. It is a very common foundation for current academic work [7, 12, 16, 17]. ALF and variants of it are also often used as a benchmark against which to test other fuzzers as in QYSM [18], NEUZZ [19] and Angora [3]. Another prominent contemporary fuzzer is SyzKaller [20]. SyzKaller is similar to AFL in that it is a standard against which to test fuzzing ability and is used as a basis for improvements as in Moonshine [21], however it is specifically targeted toward kernels - originally the Linux kernel. In this way it illustrates the increasing

specificity of modern fuzzers, as researchers begin to identify nuances within the field of fuzzing, and work to address them. SyzKaller is emblematic of one class of problems, namely the targeting of a system with an extremely regimented input scheme. Other issues which will be treated at greater length in the following sections include the well-known issue of state-space explosion, and difficulty in implementing a feedback system. Many of the fuzzers which are currently considered very influential or state-of-the-art, directly tackle one or more of these problems.

A. State-space explosion

Any technique that utilizes symbolic-execution of any kind suffers acutely from the massive state-spaces of most non-trivial programs. As a result, a common component of many symbolically-enhanced techniques is to use symbolic execution sparingly. Driller [14], for instance, uses a combination of concrete and symbolic execution (concolic execution) to determine inputs which will satisfy complex checks. Similarly FFuzz [17] uses symbolic execution to break-out of complex areas in which mutation isnt finding new paths. Though similar in nature to Driller, FFuzz targets the entire system stack, specifically kernel code.
The issue of state-space explosion, and the prominence and power of symbolic execution has also lead to the development of improved constraint solving solutions tailored specifically to fuzzing as in QYSM [18]. Angora [3] addresses this issue by using techniques which approximate symbolic execution, ie. which solve constraints.

B. Difficulty with highly structured input

Though it is relatively easy to generate completely random input, such an approach is unlikely to exercise many deep paths in a program as it is likely that most inputs will cause the SUT to reject them early in its execution. An excellent example of this problem is a kernel. A kernel's API is essentially the system call (syscall) interface, but just invoking random syscalls with random parameters will exercise very little of the kernel's code, as many of the calls require as input valid results of other syscalls, or have complex semantics. SEMFUZZ [8] is built on the aforementioned SyzKaller, and utilizes Natural Language Processing (NLP) of semantic information like bug reports and git log notes to better navigate the kernel interface. In a completely different area, IoTFuzzer [22] deals with the same problem by learning the protocols used in phone-to-IoT-device communication in order to fuzz the software on the device.

C. Feedback Mechanisms

Most modern fuzzers utilize some form of feedback loop, such that the results of previous runs can be used to guide mutation, seed selection (the selection of what to base mutation on), etc. This can be done via compile time or runtime instrumentation in the case of most white and grey box scenarios, but is sometimes impossible, as in the case of fuzzing embedded devices running close-source firmware. In such a case, as demonstrated by Muench et al. [23], the naive instrumentation stand-in strategy of checking to see if the device is still live, is often insufficient to achieve an understanding of potential vulnerabilities.

Other fuzzers not explicitly mentioned above but which further emphasize this complex landscape include VUzzer [13], a coverage-based fuzzer which combines DTA with evolutionary algorithms to prioritize deep paths over frequently exercised ones. TIFF [10] is a directed extension of VUzzer which makes use of Data-Structure Inference (DSI) to direct fuzzing toward bugs resulting from specific classes of memory-corruption errors by specializing mutation for each class. NEUZZ [19] novelly aims to increase code-coverage by using a fuzzer to learn an abstraction of a SUT, and analyzing that abstraction to optimize coverage in the fuzzing of the original program. As a final example of novel contemporary deviations from Miller's original work, T-Fuzz [24] modifies the SUT when it fails to pass difficult input checks by reversing the logic of the conditional statements.

III. Protocol

The goal of this literature review is to assess what enhancements have been applied to the process of fuzzing, and to determine what metrics have been used in their analysis. The protocol consists of the following research questions.

- **RQ1** What techniques were combined with fuzzing?
- **RQ2** How was the effectiveness of the enhancement measured?
- **RQ3** What are the most promising enhancements currently implemented, based on what measures?

To this end, we have analyzed 120 papers related to fuzzing. Of these, 80 were relevant to the questions, in that they were specifically about fuzzing, enhancements to fuzzing, or reviewed fuzzing or fuzzing related techniques.

A. RQ1 - What techniques were combined with fuzzing?

Before breaking down enhancements by category, we must first operationalize our question. Specifically, we need to make clear the distinction between specific techniques combined with fuzzing from specific areas fuzzing has been applied to. As fuzzing is a popular form of software testing, it has been used against a diverse set of targets. One such case is TensorFuzz [25] which is a coverage-guided fuzzer designed to explore neural networks. Because TensorFuzz seeks to uncover erroneous neural-network behavior and not software bugs, for the purposes of this review, TensorFuzz and other similar techniques that do not target bugs in software are excluded, save any system-agnostic enhancements they put forward.
Additionally, most sophisticated fuzzing techniques include modifications that are fairly specific to their implementation, and sometimes used as the basis for quantitative analysis. For instance, REST-ler [26], which fuzzes REST APIs, compares various search strategies ie. random-walk vs breadth-first search. Other examples include VUzzer's 'impact coefficient' which varies the punishment of basic blocks leading to error-handling code, and TIFF's 'period' variable which controls

how often to prioritize seeds with the biggest number of integers in them. Such variables are sometimes referred to as tunable parameters. Their prevalence in the field of fuzzing is perhaps best evidenced by the fact that on AFL's website, one of its selling points reads:

> **No tinkering required.** In contrast to most other fuzzers, the tool requires essentially no guesswork or fine-tuning. Even if you wanted to, you will find virtually no knobs to fiddle with and no "fuzzing ratios" to dial in.

For the purposes of this review, parameter tuning is not considered an enhancement.

1) Symbolic Execution: Symbolic execution was the first real enhancement combined with fuzzing in Godefroid et al's DART [2] in 2005, where it was used to prevent the production of inputs which exercised the same path. In their 2008 work [27], Godefroid et al. included code-coverage based assessments, heralding what was to become a major component of fuzzing work, and indeed a class of fuzzers commonly known as coverage-based, their goal being to maximize code-coverage. Finally, in 2012, their work culminated in SAGE [28], a descendant of DART.

Symbolic execution has also been used to aid more specific tasks in fuzzing. The misleadingly named SmartFuzz [29] for instance, used DTA to narrow the scope of their symbolic execution component to only constraints incorporating tainted data. Though there was no random generation of data, and therefore not truly a fuzzer, this model was adopted by others. Wang et al. [30] for instance, used it to address the highly-structured input difficulty, particularly inputs which included some form of checksum. Similarly, Haller et al. [15] focused specifically on targeting buffer overflows and used DTA and symbolic execution to guide fuzzing toward exercising areas likely to contain vulnerabilities, like looping array accesses.

Given the popularity of symbolic execution and its potential for high coverage, fuzzing-specific improvements to constraint-solving - the most computationally expensive component of symbolic execution - are still being proposed. For instance, Pak [31] developed a transforming step on gathered constraints which reduced the number of times the solver needed to be invoked, while Lanzi et al. [32] utilized a heuristic limiting of interesting constraints. Finally, as concolic execution, covered in section III.4, shares many of the same components as pure symbolic execution, more enhancements have been designed and implemented under its guise which could in many cases be incorporated into a symbolic-only approach eg. QYSM [18], jFuzz [33].

2) Evolutionary Algorithms: The first work to combine an evolutionary approach with fuzzing was published in 2007 by Demott et al. [34], in which run-time analysis and partially-formed initial seeds were combined with a fitness function that rewarded increased code-coverage. Several approaches have used evolutionary techniques to reward less-often exercised code paths, as in Sparks et al. [35]. A relatively simple method of doing this is used in ALF, which rewards any input which exercises a new code path by using it as a seed for future runs [36]. VUzzer, in addition to using DTA (directly addressed in III.3), built on this idea, punishing error-handling blocks and rewarding blocks within nested control structures [13].

The aforementioned fuzzers which take advantage of evolutionary techniques are all coverage-based fuzzers, however several directed approaches have also been developed. An early herald of VUzzer was Rawat et al.'s 2010 [37] work which utilized TA and evolutionary algorithms to direct fuzzing toward potentially vulnerable areas, specifically those which might contains buffer-overflows. KameleonFuzz [38] utilized the same two techniques, but targeted web-apps with a fitness function designed to uncover cross-site scripting (XSS) vulnerabilities.

3) Dynamic Taint Analysis: DTA was first used to enhance fuzzing in 2009, when it was implemented as a component of both BuzzFuzz [39], and DXFuzzing [40]. Both used DTA to determine which bytes of their input were being used in potentially vulnerable points identified in BuzzFuzz by source code analysis, and in DXFuzzing by static analysis of the binary. These early uses of DTA define its classic use in fuzzing, which is to map specific portions of input with interesting sites within the program, and to act as a loose guide for targeting those sites. Those portions of input can then be mutated. Specific applications of this paradigm include Rawat et al. [37] and Dowser [15] which both hunted for buffer overflows, FLAX [41], which sought various classes of vulnerabilities in JS applications, TaintScope [30], and TaintFuzz [42].

In 2017, VUzzer extended the use of DTA to include assisting in type inference. In the same year, Steelix [43] proposed a lighter-weight pseudo DTA which selectively instrumented interesting conditional statements in the SUT, and mutated bytes around those observed in the input.

Last year, DTA was used in several acclaimed fuzzers. Novelly, the aforementioned IoTFuzzer used DTA to identify functions within an IoT device's Android-based control application which took communication protocol fields as arguments, allowing the fuzzer to use the application's own communications infrastructure to produce well-formed inputs. TIFF built on VUzzer, enhancing its DTA-assisted type inference with a modular data-structure inference (DSI) component, and orienting it to specifically seek integer and buffer overflows.

Finally, Angora, built on AFL, was presented. Angora's authors were interested in a way of "solving path constraints without symbolic execution" and their solution relies heavily on DTA [3]. To address issues of overhead introduced by DTA, they introduce and demonstrate the efficiency of an improved taint-storing structure, and only collect taint information once per seed inputs used. Next, in keeping with VUzzer and TIFF, the Angora authors also use DTA to infer type information by identifying bytes compared collectively. Finally, they use DTA to determine when the result of a function which reads input is used in a conditional statement, and vary the length

of that portion of input aggressively in the hopes of quickly triggering an overflow.

4) Concolic Testing (Concrete + Symbolic Execution): In 2009, Jayaraman et al. [33] published the first paper where techniques to improve the speed of concolic execution's constraint solving component were considered in relation to fuzzing, and suggested a new one: 'name-independent caching'. This improvement provides an equivalence function for constraints, and allows the old, cached solution to be used when an equivalent is found.
Nearly a decade later in 2018, Yun et al. proposed a number of novel strategies for the same problem in the form of an improved concolic executor tailored for fuzzing called QYSM [18]. Their improvements included dropping the use of an intermediate representation layer during symbolic emulation, and loosening the soundness requirement of constraint solving such that not all constraints necessarily needed to be solved. QYSM's main comparator is Driller [14], which in 2016 incorporated a concolic execution component into AFL, allowing the fuzzer to escape constraints which mutation wasnt overcoming.

5) Probabilistic and Heuristic: In their 2008 work, Sparks et al. [35] used a Markov chain to model the state-space exploration process of their SUT. Markov chains have remained popular for this purpose, and form the basis for some enhancements. They have, for instance, supported coverage improvement as an optimization problem as in AFLGo [7], and coverage efficiency improvement based on power schedules as in AFLFast [12]. VUzzer uses them to model control-flow graphs, while Bottinger et al. uses the concept to "formalize fuzzing as a reinforcement learning problem" [44].
Many, if not all, aspects of fuzzing have been modified and enhanced by heuristic schemes. One area they've improved is the generation of inputs. Godefroid et al. [27] utilized a heuristic-based search algorithm to generate well-formed inputs. Following this paradigm, Rebert et al. explored various input generation schemes for fuzzing and showed that heuristics outperformed random [45].
Another place heuristics commonly appear in modern fuzzers are in areas which present particular difficulty for automated techniques. One such example is in the analysis of flow-control though indirect jumps, as is the case in Hawkeye [46]. In a similar vein, Steelix, IoTFuzzer and TIFF all utilize heuristically enhanced mutation strategies. In the case of TIFF for instance, mutation periodically chooses seeds with the highest number of vulnerable types, like INT, and mutates them to edge cases ie. 0, maximum_integer_value, etc. Such mutation improvements are often referred to as adaptive mutation, [46, 47], and are in some cases also based on probabilistic inference, as in the case of SymFuzz [47].

6) Machine Learning: Over the past two years, machine learning techniques have become increasingly popular to combine with fuzzing. 2017 saw neural networks used to produce input grammars by Godefroid et al. [4], and to learn effective mutation strategies by Rajpal et al. [48]. Last year Hu et al. [49] used generative adversarial networks to learn industrial network protocols, while Bottinger et al. "experiment[ed] with deep Q-learning" [44]. Like Rajpal et al. [48], one of Angora's improvements is based on using machine learning to guide mutation [3]. Perhaps most novelly, NEUZZ generates an neural-network based abstraction of the SUT, and then analyzes the abstraction to develop effective code-covering mutation strategies for fuzzing the SUT [19].

B. RQ2 - How was the effectiveness of the enhancement measured?

Broadly, there are two main schools of metrics used to analyze the efficacy of a fuzzer, a quantitative school and a novelty school. In the quantitative school, metrics include time vs. coverage-percentage attained, time to reach a desired target, time vs. number of symbolic solver invocations, and number of bugs found. Most often, these measures are compared to other similar fuzzers which have used similar technologies and metrics. For example, QYSM, which utilizes concolic execution, compares coverage attained against Driller, an earlier but similar, concolic-enhanced fuzzer. Both also counted the number of bugs they found, a metric found in essentially all papers on fuzzing, beginning with Miller et al.'s seminal 1990 paper [1]. After number of bugs found, coverage attained is by far the next most common fuzzing metric and has been in use since Godefroid et al.'s 2008 work [50]; reasonable considering the most common class of fuzzer is coverage-based. Recently however, directed fuzzers have begun gaining in popularity which challenges the prevalent coverage-oriented assumptions underlying the widespread use of coverage-based techniques.
It should also be noted that, similar to parameter tuning, many research papers describing fuzzers use metrics derived specifically to improve their own implementations. For instance, Rajpal et al. [48] compares AFL, along with four versions of their implementation which use varying neural-network architectures on code-coverage per layers of neural-network. Though certainly interesting from a research perspective, from a meta analysis perspective such methodologically specific setups may lead to confusion, as, for instance, vanilla AFL doesnt use neural networks. As in the case of parameter tuning, we will not consider such technique-specific measurements here.
The novelty school is less diverse than its quantitative counterpart, but ubiquitous across time and technology. There is a tendency to consider the severity of bugs found, as in TaintScope [30] and Learn & Fuzz [4], as well as the potential exploitability, as in IMF [51] and IFuzzer [52]. Finally, there is a tendency toward emphasizing bugs found in popular or widely used SUTs.

1) Quantitative Results: The following are metrics used in fuzzing research which fall within our first, quantitative school. We exclude pure number of bugs found, as though it

is quantitative, the variety of systems targeted and techniques used render it not terribly useful from a meta perspective. It is however quite useful in relation to time, especially within specific areas of fuzzing research, as will be covered in the coming sections.

a) Code Coverage: Even before Godefroid et al. [50], there were papers which considered code-coverage as an important measurement of fuzzing, however did not explicitly report on it such as Banks et al. [53], and Demott et al. [34]. After Godefroid et al. [50], reporting coverage became commonplace [14, 31, 33, 54]. Since then, all coverage-based fuzzers present code-coverage metrics.
Early generational fuzzers were more likely to measure, often by plotting, number of generations vs. code coverage as in JFuzz [33] and Pak's 2012 work [31]. Similarly, mutational fuzzers reported the number of mutations vs code-coverage as in Kargen et al. [54]. Building on this trend, metrics relating the time it took a fuzzer to achieve some amount of coverage began to appear. To our knowledge, the first such use was by Stephens et al. in their 2016 work [14].
Recently, research has begun distinguishing between line and edge coverage as in Angora [3] and NEUZZ [19], and basic-block coverage as in VUzzer [13] and TIFF [10]. The current version of AFL considers all three [36]. Another recent twist on the coverage metric is considering the change in coverage growth-rate over time as in Steelix [43] and CollAFL [16].

b) Symbex/Concolic Specific: Due to the prevalence of symbolic and concolic enhancements, and their inherently large overhead, some research, like SAGE [28] and Pak's 2012 work [31], measured these components of their systems directly. This is perhaps best demonstrated by QYSM, which cites this overhead as one of the main problems facing modern fuzzers, and which is devoted to making those components more efficient [18]. Other works which count time spent solving constraints include Godefroid et al.'s 2008 work [50], and Dowser [15].

c) Bugs Found Over Time: Coverage-based fuzzing is interested in exercising as much code in as little time as possible, therefore bugs found per unit time is a natural metric which combines the gold standard of what we found with efficiency. The VUzzer metrics for instance are focused on time to find vulnerabilities, per their lightweight fuzzing theory [13]. Other fuzzers which utilize this metric include TaintFuzz [42], CollAFL [16], Steelix [43] and FairFuzz [55].

d) Time to Target/Vulnerability: Alternatively, directed fuzzers try to minimize the time it takes to exercise a particular portion of code. SemFuzz, for instance, measures the amount of time required to reach a target, and then to trigger the vulnerability [8]. Similarly, RAZZER measures the time it took to produce an input which produced a race-condition bug in the Linux Kernel [9]. Both SemFuzz and RAZZER used SyzKaller as their main comparator. Other fuzzers which follow this paradigm are Rawat et al.'s 2010 work [37], Hawkeye [46] and AFLGo [7].

e) Trend toward fuzzing metrics; away from technique metrics: This trend is perhaps best exemplified by the case of evolutionary algorithms. Older evolutionarily-enhanced fuzzers often counted the number of generations their evolutionary process required to achieve a desired result, such as a crash in a known vulnerable program as in Rawat et al. [37] and KameleonFuzz [38]. Currently, evolutionary enhancements are pretty common, but not specifically measured in this fashion, as it's the overall performance of the fuzzer which usually takes center stage, as in VUzzer [13]. As evolutionary algorithms were adopted early in the history fuzzing, its not unreasonable to assume that a similar effect will occur in other areas of enhancement, where rarified, technique-specific metrics give way to generic fuzz-related metrics.

f) Balancing SUT Novelty and Reproducibility: Finally, the value of SUT novelty should be addressed in quantitative terms. Some works have targeted highly specialized SUTs, like GANFuzz [49], which fuzzed industrial network protocols. While more software testing is nearly always preferable to less, and the validity of meta analyses (which we'll speak to more in section IV) is improved by greater diversity amongst the population of softwares tested, much of the current work on fuzzing has to do with enhancing extant fuzzing paradigms. Such incremental work inherently benefits from the ability to run tests on software which has also been fuzzed by other researchers as this provides a basis for comparison. Given this situation, there seems to be a need to strike a balance between the overall good gained by fuzzing as diverse a group of SUTs as possible and not undermining the fuzzing community's ability to perform reliable, reproducible tests.

2) Novel Results: In comparison to the quantitative metrics just described, more qualitative considerations are also often present. We break these down into two classes, the first of which revolves around an analysis of the severity or exploitablity of discovered bugs. The second class consists of an awareness and consideration of the popularity, and related potential impact of the SUTs.

a) Severity/Exploitability: While recent works often include an evaluation of their system on synthetic data sets like the LAVA-M and DARPA CGC binaries, many also present a collection of results from currently deployed programs [3, 10, 13, 43]. When considering these production programs, newness of bugs found is sometimes considered as was done by Wang et al. [30]. More common, is to consider their severity in some way, by either hand-examining the bugs found as in IMF [51], or by utilizing a tool like *!exploitable* [56] which utilizes heuristic analysis of a crash to assign a likelihood of exploitability as in VUzzer [13], IFuzzer [52] and TIFF [10].

b) Impact: The programs used as test cases for a fuzzer vary, however there is a tendency to use programs that are widely deployed, or popular, as these tests naturally make more compelling statements about a fuzzer's ability. Widely deployed programs can generally be assumed to be more mature than other tests, and likely have already undergone software testing of various kinds. AFLGo, for instance, emphasized the age of the patch that introduced a found bug in relation to their directed fuzzing of patched code [7]. Popular real-world programs to fuzz include ffmpeg [15, 18, 29, 45], the PDF library libpoppler [10, 13, 15, 24, 45, 54] and compression libraries like bzip and gzip [7, 29, 43, 57].

C. RQ3 - What are the most promising enhancements currently implemented, based on what measures?

After exploring the field of fuzzing, its clear that there is no single best fuzzer, or fuzzing technology. Certain techniques lend themselves to different areas. For instance, heuristics and static analysis have enhanced directed work, while concolic execution has been used to effectively improve code-coverage. Most all types of fuzzing research have benefited from taint-analysis. We'll therefor consider this question per fuzzing purpose.

a) Directed: Several recent contributions to this area strongly suggest what currently appear to be the most promising enhancements, or combination of enhancements. Considering approaches that are purely directed, ie. they target patch code, several works stand out. Hawkeye [46] uses a novel heuristic-enhanced static analysis step to improve its understanding of control flow and block distance, and incorporates this into a fuzzing loop which also utilizes adaptive mutation to guide fuzzing toward a target. Hawkeye's authors drew inspiration from AFLGo [7]. In AFLGo, the problem of 'directing' was operationalized as an optimization problem. Hawkeye builds on this, by adding seed prioritization via power scheduling. The Hawkeye authors use "Time-To-Exposure" (TTE) to capture the time it took the fuzzer to generate an input which triggered a targeted vulnerability, and by this metric - clearly valid for a directed system - demonstrated significant improvements over AFLGo.
Similarly, RAZZER [9], which utilized static analysis as well, also used a TTE-like measurement, and demonstrated improvements over its base system, Syzkaller. In the most novel augmentation to this class of fuzzing, the authors of SEMFuzz [8] used NLP to identify targets, and then measured TTE.
Based on these works, it appears that directed-fuzzing, though highly dependent on targets chosen, type of SUT, etc. is enhanced, perhaps more promisingly, by heuristic-based improvements to static analysis and mutation strategies, and by complimenting such gold-standard methods with novel techniques ie. NLP.

b) Coverage-based: Improvements in coverage-based fuzzing are perhaps demonstrated best by Angora [3], which makes use of DTA, and DTA-efficiency enhancements, machine learning, data-type inferencing and heuristic-enhanced mutation. In this way, it exemplifies the current State-of-the-Art fuzzer, one which takes advantage of all the work done before it, and combines that work into a single new tool. Its authors compared it to Steelix, VUzzer and AFL, each of which they drew inspiration from. By running tests on both the LAVA-M set, and on real-world programs, they demonstrated improved code-coverage and bug-finding ability. This work demonstrated the significantly varied potential uses of DTA, and its continuing value as an enhancement especially as it applies to type-inferencing, like in TIFF [10] and Angora [3]. QYSM also utilized a code-coverage metric, but instead of combining many enhancements, it worked to improve a single one: concolic execution. Understanding that concolic execution is expensive, but highly valuable to code-coverage, they proposed augmentations, compared their system against AFL and Driller, and demonstrated improvement [18]. This demonstrates the continued usefulness of symbolic and symbolic-like enhancements.

c) Bug/Vuln Specific: This category of fuzzing is particularly difficult to analyze for promising techniques due to its inherently-specific nature, ie. integer overflows in user space compared to race conditions in a kernel. For instance, TIFF utilized DTA to drive its goal of enhancing fuzzing with data-type inference, and heuristics to focus mutation on integer and buffer overflows. As the authors of TIFF describe their work, TIFF is 'bug-directed', and as such shares features of the goals of both coverage-based and directed fuzzers. While it is important to effectively exercise code which may contain the sought after bugs, it is also important to still maintain reasonable code coverage. For this reason, TIFF's authors compare it to VUzzer and AFLFast [12], both in terms of speed of bug discovery, and in terms of code coverage. TIFF's efficacy, like Angora's, demonstrates the continued value of DTA, and heuristics, as well as the promise of type inference.

IV. Conclusion

In section III.A we presented an exploration of the various techniques which have been used to enhance fuzzing. Broadly, symbolic execution, evolutionary algorithms, taint analysis, concolic execution, heuristics and machine learning classify what technologies have been applied to fuzzing. As these also capture a large portion of Computer Science, we're lead to perhaps the most clear trend in fuzzing: most promising, up-and-coming technologies will eventually be applied to fuzzing. Similarly, as fuzzing continues to demonstrate a strong ability to uncover new bugs and vulnerabilities, and matures as a software-testing technique, there's an increasing interest in applying it to areas beyond the classic UNIX binaries which accept standard input as was done in Miller's early work [1, 58]. Works like IoTFuzz and Muench et al. [23] have begun to apply the techniques to embedded devices, and GANFuzz

[49] has targeted industrial control networks. RAZZER and FFUZZ have begun to expand the classic target to include the full system stack, including the kernel, while works like Dowser, TIFF, and KameleonFuzz have targeted specific types of vulnerabilities.

As suggested in section II the modern fuzzing landscape has a few notable foundational fuzzers, most obviously AFL and Syzkaller. Though significantly newer than AFL, Syzkaller has already spawned several revisions and specialized augmentations including RAZZER [9], SemFuzz [8] and Moonshine [21]. AFL, famous for its flexibility, has been the basis for a host of other fuzzers [7, 12, 16, 17, 55] and augmentations including Skyfire [5], and the works of Xu et al. [59] and Rajpal et al. [48]. This branching development epitomizes the modern scientific development strategy of not reinventing the wheel. More novel and field-advancing work can be achieved if common, high-quality standards are used.

Such standards are also at the core of one of the issues only now starting to be addressed in fuzzing research: the difficulty of meta-analysis and reproducing experiments. In 2016, Dolan-Gavitt et al. [60] introduced a method for injecting vulnerabilities into programs on a large scale. The resulting set of binaries, called LAVA-M and referenced multiple times in this paper, have begun to see widespread use in fuzzing literature [3, 10, 13, 16, 24, 43] and allow measurements like TTE, and number of bugs found per unit time, to be meaningfully compared across techniques, as known bugs can be individually uncovered and compared. DARPA's Cyber Grand Challenge (CGC) set of binaries is another like set being adopted more today [13, 14, 24, 43]. The tendency for modern fuzzers to include such standardized measures is certainly positive, and will benefit the community.

REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, pp. 32–44, Dec. 1990.

[2] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, pp. 213–223, June 2005.

[3] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *CoRR*, vol. abs/1803.01307, 2018.

[4] P. Godefroid, H. Peleg, and R. Singh, "Learn & fuzz: Machine learning for input fuzzing," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, (Piscataway, NJ, USA), pp. 50–59, IEEE Press, 2017.

[5] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 579–594, May 2017.

[6] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," *SIGPLAN Not.*, vol. 52, pp. 95–110, June 2017.

[7] M. Bohme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), pp. 2329–2344, ACM, 2017.

[8] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), pp. 2139–2154, ACM, 2017.

[9] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), IEEE Computer Society, may 2019.

[10] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "Tiff: Using input type inference to improve fuzzing," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, (New York, NY, USA), pp. 505–517, ACM, 2018.

[11] M. Zalewski, "American fuzzy lop." http://lcamtuf.coredump.cx/afl/.

[12] M. Bohme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 1032–1043, ACM, 2016.

[13] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *NDSS*, Feb. 2017.

[14] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 01 2016.

[15] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, (Berkeley, CA, USA), pp. 49–64, USENIX Association, 2013.

[16] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, May 2018.

[17] B. Zhang, J. Ye, X. Bi, C. Feng, and C. Tang, "Ffuzz: Towards full system high coverage fuzz testing on binary executables," in *PloS one*, 2018.

[18] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium USENIX Security 18)*, pp. 745–761, 2018.

[19] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program learning," *CoRR*, vol. abs/1807.05620, 2018.

[20] "syzkaller." https://github.com/google/syzkaller.

[21] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing os fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 729–743, USENIX Association, 2018.

[22] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *NDSS*, 01 2018.

[23] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *NDSS*, 01 2018.

[24] M. P. Hui Peng, Yan Shoshitaishvili, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 697–710, 2018.

[25] I. G. Augustus Odena, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," *arXiv preprint arXiv:1807.10875*, 2018.

[26] M. P. Vaggelis Atlidakis, Patrice Godefroid, "Rest-ler: Automatic intelligent rest api fuzzing," *arXiv preprint arXiv:1806.09739*, 2018.

[27] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *SIGPLAN Not.*, vol. 43, pp. 206–215, June 2008.

[28] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, pp. 20:20–20:27, Jan. 2012.

[29] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, (Berkeley, CA, USA), pp. 67–82, USENIX Association, 2009.

[30] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution," *ACM Trans. Inf. Syst. Secur.*, vol. 14, pp. 15:1–15:28, Sept. 2011.

[31] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," Master's thesis, School of Computer Science Carnegie Mellon University, 2012.

[32] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari, "A smart fuzzer for x86 executables," in *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, (Washington, DC, USA), pp. 7–, IEEE Computer Society, 2007.

[33] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jfuzz: A concolic whitebox fuzzer for java," in *Proceedings of the First NASA Formal Methods Symposium*, pp. 121–125, 01 2009.

[34] J. Demott, D. Richard, R. Enbody, D. William, and W. Punch, "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing," *BlackHat and Defcon*, 02 2007.

[35] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated vulnerability analysis: Leveraging control flow for evolutionary input

crafting," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 477–486, 01 2008.

[36] M. Zalewski, "Technical "whitepaper" for afl-fuzz." http://lcamtuf.coredump.cx/afl/technical_details.txt.

[37] S. Rawat and L. Mounier, "An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light," in *Proceedings of the 2010 European Conference on Computer Network Defense*, EC2ND '10, (Washington, DC, USA), pp. 37–45, IEEE Computer Society, 2010.

[38] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box xss detection," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, (New York, NY, USA), pp. 37–48, ACM, 2014.

[39] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 474–484, IEEE Computer Society, 2009.

[40] Z. Wu, J. Atwood, and X. Zhu, "A new fuzzing technique for software vulnerability mining," in *International Conference on Software Engineering*, 01 2009.

[41] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications," in *Proc. of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[42] G. Liang, L. Liao, X. Xu, J. Du, G. Li, and H. Zhao, "Effective fuzzing based on dynamic taint analysis," in *2013 Ninth International Conference on Computational Intelligence and Security*, pp. 615–619, Dec 2013.

[43] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *ESEC/SIGSOFT FSE*, 2017.

[44] K. Bottinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," *ArXiv e-prints*, Jan. 2018.

[45] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (Berkeley, CA, USA), pp. 861–875, USENIX Association, 2014.

[46] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), pp. 2095–2108, ACM, 2018.

[47] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*, pp. 725–741, May 2015.

[48] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *CoRR*, vol. abs/1711.04596, 2017.

[49] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "Ganfuzz: A gan-based industrial network protocol fuzzing framework," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pp. 138–145, ACM, 2018.

[50] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *In NDSS*, 2008.

[51] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), pp. 2345–2358, ACM, 2017.

[52] S. Veggalam, S. Rawat, I. Haller, and H. Bos, *Ifuzzer: An evolutionary interpreter fuzzer using genetic programming*, vol. 9878 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 581–601. Springer/Verlag, 9 2016.

[53] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: Toward a stateful network protocol fuzzer," in *Proceedings of the 9th International Conference on Information Security*, ISC'06, (Berlin, Heidelberg), pp. 343–358, Springer-Verlag, 2006.

[54] U. Kargen and N. Shahmehri, "Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 782–792, ACM, 2015.

[55] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, (New York, NY, USA), pp. 475–485, ACM, 2018.

[56] J. Foote, "Gdb 'exploitable' plugin." https://github.com/jfoote/exploitable, 2015.

[57] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 78–86, Jan. 2007.

[58] B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of unix utilities and services," tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 1995.

[59] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), pp. 2313–2328, ACM, 2017.

[60] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 05 2016.