A Review of Machine Learning Applications in Fuzzing

Gary J Saavedra
Kathryn N Rodhouse
Daniel M Dunlavy
Philip W Kegelmeyer
Sandia National Laboratories
Albuquerque, NM, USA
{gjsaave,knrodho,dmdunla,wpk}@sandia.gov

ABSTRACT

Fuzzing has played an important role in improving software development and testing over the course of several decades. Recent research in fuzzing has focused on applications of machine learning (ML), offering useful tools to overcome challenges in the fuzzing process. This review surveys the current research in applying ML to fuzzing. Specifically, this review discusses successful applications of ML to fuzzing, briefly explores challenges encountered, and motivates future research to address fuzzing bottlenecks.

KEYWORDS

Fuzzer, fuzzing process, machine learning applications, vulnerability assessment, symbolic execution

1 INTRODUCTION

Fuzzing is a technique in which a large number of generated inputs, both valid and invalid, are fed into a program to search for flaws and vulnerabilities. Fuzzers, or automated tools to perform fuzzing, have played an important role in quality assurance, system administration, and vulnerability assessment over the last three decades [15, 16, 34, 48]. Modern fuzzers now incorporate techniques from other disciplines; in this survey, we explore how some modern fuzzers incorporate different types of machine learning (ML). We specifically focus on fuzzers used for vulnerability assessment due to their widespread use.

Machine learning is a method for training computer models to perform some operation without being explicitly programmed. ML techniques are applied across a wide range of problems, from image processing to sequence modeling [21, 26, 47]. In this survey, we focus on three main types of ML, each of which is suited for a different type of task. *Supervised learning* is used to train a model to identify the *class* label of a given data point, such as whether an image contains a specific object. This type of ML requires data sets where each data point is given an explicit label. *Unsupervised learning* is used to train a model to find patterns or similarities between data points rather than for classification. This type of ML is used when the data does not have explicit labels. *Reinforcement learning* is used to train a model, often referred to as an *agent*, to take an optimal set of actions in an environment. This type of ML rewards the agent for each action it takes in the environment. Similar to supervised learning, the rewards act as a label for the agent and provide an indication of the optimal actions to take. Thus, the agent can be trained to take a set of actions which lead to the highest reward. Each of these three types can also take on a particular form of ML called *Deep learning*. Deep learning refers to a type of hierarchical learning that can be used to more effectively learn the underlying features and structure of a set of data points [21].

In this survey, we explore how ML has been applied to address core research questions in fuzzers used for vulnerability assessment. This survey is representative rather than exhaustive; we do not attempt to discuss all applications of ML in fuzzers. In Section 2, we give a brief overview of the fuzzing process. In Section 3, we explore previous and ongoing research applying ML techniques to fuzzing. We also discuss difficulties associated with applying ML to fuzzing. We conclude with a summary of how ML has been applied to fuzzing and how ML might be applied to fuzzing in the future.

2 OVERVIEW OF FUZZING

In “A Review of Fuzzing Tools and Methods” [17], Fell explores the use of fuzzers for vulnerability assessment. We recommend Fell’s treatment of fuzzing for the interested reader as he describes the process in depth. Here, we provide only a brief overview of fuzzing to frame our discussion of where and how machine learning (ML) techniques have been applied, which will be covered in Section 3.

Again, fuzzing is a technique in which a large number of generated inputs, both valid and invalid, are fed into a program to search for flaws and vulnerabilities. Fuzzers automate much of this process, taking in initial program knowledge and sending out any interesting program states discovered [36]. Often, a human user both provides initial program knowledge and analyzes these output program states. Traditionally, "interesting program states" were program crashes that identified flaws and vulnerabilities in the progam, but more complex program monitoring techniques now allow for identification of other types of interesting states.

The goal of a fuzzer is to create inputs that cause the program to execute program paths, discovering those that lead to interesting program states. Thus, fuzzers are frequently measured by the diversity of program paths explored, termed *coverage*.

In this section, we introduce types of fuzzers and break down the fuzzing process into stages. We walk through each stage, showing how the different types of fuzzers approach some stages differently. We introduce the types of tasks that exist in each stage, framing the most compelling challenges that are currently not fully solved at each stage. Finally, we touch on ways to compare fuzzers.

Types of Fuzzers

A fuzzer that generates completely random input and feeds it to a program is a *naïve fuzzer*. While naïve fuzzers are fairly easy to implement, they are unlikely to reach interesting program states in a timely manner. Three primary types of modern fuzzers improve on naïve fuzzers: *mutation-based*, *generation-based*, and *evolutionary*.

Mutation-based fuzzers blindly *mutate* or manipulate provided input to feed to the program. Generally mutation-based fuzzers are not aware of the expected input format or specifications, and they cannot select mutations wisely. Peach is a fuzzer that can perform mutation-based as well as generation-based fuzzing [49].

Generation-based fuzzers take information about the expected input format or protocol through specifications. Generation-based fuzzers *generate* or craft inputs based on these specifications. Generation-based fuzzers include Peach and Sulley, a Python fuzzing framework that can generate inputs for file transfer protocols, network protocols, and file formats [1].

Evolutionary fuzzers, the newest type of fuzzers, build on mutation-based fuzzers by selecting some inputs over others for mutation. Specifically, evolutionary fuzzers aim to evaluate what each input causes the program to do and change how they proceed based on that evaluation. Practically, current state-of-the-art evolutionary fuzzers rank inputs using a fitness function (often coverage) and select the best-ranked inputs to mutate. Example evolutionary fuzzers include honggfuzz [22], AFL [58], and libFuzzer [31].

For the interested reader, Vimpari's 2015 thesis evaluates the utility of a subset of free fuzzers [51]. However, we move on to explore the fuzzing process.

Fuzzing Process

Figure 1 breaks a general fuzzing process into stages. Each fuzzer takes in *program knowledge*, uses that knowledge to *generate inputs*, *select inputs* to feed to the program being fuzzed, *monitor program* actions in response to each input, and identify and export any *interesting program states* for the user to analyze. The fuzzer automatically repeats the inner stages, and it can use results from monitoring the program to *evaluate input* and inform the next input generation stage. While this process describes fuzzing in general, the different types of fuzzing exhibit nuances in generating input and evaluating inputs based on results from monitoring the program. We will next discuss these stages and their nuances.

Pre-Fuzzing: Program Knowledge. Prior to beginning the fuzzing process, a user incorporates program knowledge into the fuzzer. At a minimum, a fuzzer needs program knowledge about how to *instrument* or observe a program, including what constitutes an interesting program state, and about which input interfaces to explore. For naïve fuzzers, this knowledge is sufficient to generate randomized input and send it to the indicated interfaces. For non-naïve fuzzers, however, additional program knowledge is needed to effectively generate inputs.

Mutation-based fuzzers often require additional program knowledge in the form of an *input corpus*, or a set of program inputs to feed to the indicated interfaces, in order to effectively generate inputs. Such program inputs tend to be examples of expected inputs, e.g., TLS handshake buffers for OpenSSL [4] or memory dumps for Volatility [9]. Valid initial inputs guide the fuzzer to explore deep program states, i.e., states that are found after many branches in the program, by allowing the program to iterate on valid input, as opposed to random initial inputs that may not pass initial program input checks. Initial input corpora are generally created manually, but large sets of inputs do pre-exist for some applications (e.g., all PDFs on the internet for Adobe Reader).

Generation-based fuzzers take in additional program knowledge in the form of input specifications, such as expected file formats or protocol descriptions. Generation-based fuzzers typically achieve better code coverage and deeper program state exploration than mutation-based fuzzers [17]. These fuzzers are more difficult to set up because they require developing accurate program input specifications, which is often significantly more time-intensive than generating an input corpus. Generation-based fuzzers are limited to exploring the input space specified by these specifications.

Evolutionary fuzzers, like mutation-based fuzzers, take in additional program knowledge through an input corpus.

Typically, program knowledge is provided by the human user and is developed manually. In contrast, fuzzers automatically iterate over the next three stages of the fuzzing

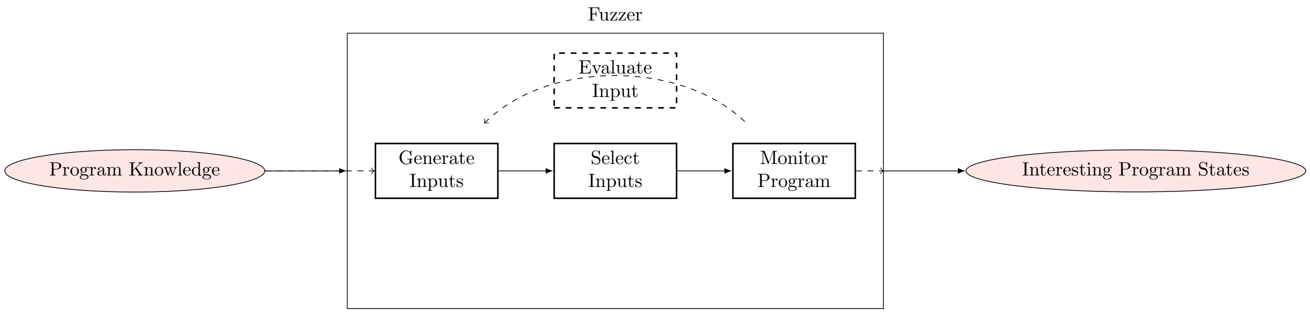

Figure 1: Fuzzing Process

process. A fuzzer will generate new inputs, down-select and order inputs to send to the program, monitor the program for interesting program states, and repeat, generally until the user terminates the fuzzer. We explore these stages next.

Stage 1: Generate Inputs. In the first stage of the fuzzing process, the fuzzer uses program knowledge to generate inputs to feed to the program through the identified interfaces. The goal of this stage is to create inputs that will explore new and interesting program states. Not all of the new inputs will be sent to the program; the next stage of the fuzzing process down-selects to the most relevant of the inputs generated here. However, the fuzzer aims here to generate the most relevant inputs.

Mutation-based fuzzers generate new inputs by manipulating previous "interesting" inputs. In the first iteration of this stage, a fuzzer manipulates input provided in its input corpus. Later, the fuzzer may adjust its collection of interesting inputs based on results from monitoring the program and evaluating input performance.

Mutation-based fuzzers mutate an interesting input by altering some portion of the input. Two decisions are necessary for such mutation: 1) where to mutate (including the length of the mutation), and 2) what new value to use for the mutation [40]. Fuzzers use many techniques to make these decisions. Common techniques include randomization (from bits to entire sections), specific bit flips, integer increments, and integer bound analysis and substitution.

Generation-based fuzzers generate inputs by creating a new input according to the input specification. Given a specification, a finite number of inputs corresponding to that specification exist [48]. Because the input search space is finite, generation-based fuzzers can explore all possible specified inputs, which allows generation-based fuzzers to accurately measure how much of the input space they have explored.

Evolutionary fuzzers, again like mutation-based fuzzers, generate new inputs by manipulating previous interesting inputs. Generally, evolutionary fuzzers either mutate one input or select two or more inputs and perform crossover, combining components of the selected inputs to make a new input; however, other randomization techniques may be used as well. These fuzzers select inputs for randomization by evaluating input performance in previous stages.

Theoretically, mutation-based and evolutionary fuzzers could generate an infinite number of inputs. The infinite input search space makes it difficult to estimate how much of the input space has been explored. Additionally, as input lengths grow, mutation and crossover techniques require more manipulations to generate each new input and thus become more computationally challenging [18]. This can slow the fuzzing process considerably and severely limit the coverage obtained by the fuzzer in a set amount of time.

Symbolic Execution is a static analysis technique that can help to generate new inputs that increase fuzzer coverage. Although symbolic execution is not a type of fuzzer, tools such as Driller [46] and SAGE [19] have combined fuzzing and symbolic execution to improve input generation. We discuss symbolic execution separately here because it can generate new inputs for mutation-based and evolutionary fuzzers. These new inputs can be added to the input corpus or otherwise force-fed to the program. We briefly describe how symbolic execution can help to generate new inputs to increase coverage.

Symbolic execution analyzes a program to find *constraints*, or limitations, on data values inside the program without making assumptions about input values fed to the program.[1] Symbolic execution works by abstracting input data into *symbolic* values, or data that might take on many *concrete* values, and stepping through the program using these symbolic values. When the program branches on a symbolic value, e.g., `if (x < 10)`, either side of the branch might be taken. In such cases, the symbolic execution engine makes

[1]Actually, constraints can be added to input values as well, but we ignore that here for simplicity.

two symbolic program states, each including a constraint met by the *taken* side of the branch, and continues symbolically executing both of those states. In our example, the branch-taken state would include the constraint `x < 10` and the branch-not-taken state would include the constraint `x >= 10`; other than that, the two symbolic states would be identical.

A symbolic state includes constraints representing the series of branch decisions necessary to reach that state, i.e., a path. A user can ask questions about the path used to reach that state. For example, what is a value of `x` that would reach this point? To answer such a question, the symbolic state is *modeled* as a set of constraints that encode, for each input variable, the range of valid values along the path. A constraint solver then *solves* these constraints, either returning a valid concrete assignment to the input variables or proving that no such assignment exists (i.e., the constraints are unsatisfiable) [39]. This assignment represents an input that would cause the program to reach the desired program state by making the same branch decisions as those in the original path.

Fuzzers can use symbolic execution to create new inputs that explore new or interesting paths by asking the solver for solutions to newly discovered symbolic states. Unfortunately, symbolic execution is computationally costly, and the large number of possible program paths makes it infeasible to symbolically execute an entire program.[2] This high cost means that symbolic execution must be used cleverly; it is typically useful for finding paths that are difficult to find through random exploration. To mitigate this cost, fuzzers pair symbolic execution with their standard input generation techniques [19]. The symbolic execution engine can limit paths by "following" an interesting input, e.g., only allowing new symbolic states to diverge from the input for a limited number of branches. However, the computational costs and path explosion remain significant hurdles. A large number of research efforts are attempting to address these research challenges in symbolic execution [25, 35, 46].

Unfortunately, fuzzers still suffer from low coverage, even when pairing their specialized input generation techniques with symbolic execution. Effective input generation remains a research challenge, and other clever applications of new techniques may help to intelligently generate inputs that increase coverage. However, even these input generation techniques create many more inputs than fuzzers have computational time to execute. In the next stage, inputs are filtered to a smaller set to be fed into the program.

Stage 2: Select Inputs. In the second stage of the fuzzing process, the fuzzer selects and orders the inputs to send to the program. Recall, a fuzzer's goal is to exercise new program paths quickly. As Böhme et al. assert, however, most inputs exercise the same few program paths [5, 6]. To combat this, a fuzzer must use its input corpus effectively and minimize the computation spent to discover new program paths.

Input test scheduling, or *seed selection*, can help combat this tendency of inputs to explore limited program paths. Input test scheduling ranks and selects inputs and input order, anticipating which new inputs are most likely to lead to new and disjoint interesting program states. For vulnerability assessment, test scheduling generally chooses inputs to maximize the number of bugs found [53]. Input test scheduling is critical to effectively explore large or infinite input search spaces, but finding the right scheduling strategy for a particular program and fuzzer remains a research challenge. Luckily, tools such as FuzzSim quickly compare input selection strategies using input performance information across many iterations of the process [53].

As mentioned previously, as the fuzzer continuous to iterate through the process, mutation-based and evolutionary fuzzers may collect many extremely large inputs. This artificially slows down the fuzzing process. To combat this, the fuzzer or user needs to perform either *corpus minimization*, reducing the number of inputs, or *input minimization*, reducing the size of each input. For example, *corpus distillation*, introduced by Ormandy in "Making Software Dumberer" [37], does this by reducing a set of inputs to the minimal set of inputs that maintain the same coverage. SEC Consult, an Austrian vulnerability lab with a fuzzing capability, ignores specific portions of the input search space identified through manual analysis as not interesting in order to limit the number of manipulations performed on each input [18].

Whether intelligently scheduling inputs, reducing the number of active inputs, or keeping inputs small, these techniques attempt to reduce the time needed for a fuzzer to find new interesting program states. However, effectively using the fuzzer's limited computational time remains a research challenge. In the next stage, the fuzzer sends the selected inputs, monitors the program's resulting actions, and identifies interesting program states.

Stage 3: Monitor Program. In the third stage of the fuzzing process, the fuzzer feeds the program chosen inputs and monitors the program to identify interesting program states. "Interesting program states" exhibit a specific program behavior in that state. In most cases, a crash represents the behavior of interest (i.e., the program fails unexpectedly). However, any behavior that is observable through program instrumentation might be used to identify interesting program states. For example, Valgrind can detect (observe) memory corruption even if the corruption does not cause a crash [50]. As another example, Heelan uses fuzzing to identify potential

[2] Paths or symbolic states are exponential in the number of branches, and each iteration of a loop results in another branch.

program memory allocators [23]. For vulnerability assessment, interesting behaviors are observable behaviors that relate to possible flaws or vulnerabilities.

As mentioned, a fuzzer requires program knowledge about how to instrument the program and what constitutes an interesting program state. A human user often provides this information, but the definition of what an interesting program state *should* be remains a research challenge. In the vulnerability assessment case that is, what observable behaviors are most relevant to identifying flaws or vulnerabilities?

Once the fuzzer has identified an interesting program state, it needs to provide descriptions of that state back to the user for analysis. Program state descriptions differ wildly by fuzzer. For example, on a crash, one fuzzer might simply provide the input causing the crash, while another fuzzer might provide a full *core dump*, a capture of program memory at the point of the failure. After evaluation (the next stage), this information might guide further fuzzing iterations.

Stage 4: Evaluate Inputs. In the fourth stage of the fuzzing process, the fuzzer evaluates how well inputs performed. Many fuzzers use code coverage to measure the utility of an input: if the input causes execution of a new part of code (generally, a new basic block), that input has increased coverage and is rated highly. The libFuzzer tool uses a similar metric, data coverage, which also rates an input highly if new data values occur at a previously explored comparison in the code. Some fuzzers use bug discovery as a metric; inputs that cause a crash are rated highly.

Evolutionary fuzzers *require* feedback about how well inputs performed [17]; that is, they must be able to evaluate inputs and rank them. These fuzzers use input rank both in generating new inputs and in selecting and sending inputs.

Mutation-based and **generation-based** fuzzers do not generally require input performance feedback, but the same metrics can help to evaluate a fuzzer's performance overall.

Practically, coverage metrics are heuristics and do not provide a complete assessment of input performance. Thus, effective and comparable metrics remain a research challenge.

Post-Fuzzing: Interesting Program States. Following the fuzzing process, the user analyzes interesting program states output by the fuzzer. This process is often highly manual and benefits from research in related areas in software engineering. For each output state, the user analyzes the state to determine the *root cause* of the interesting behavior in that state. Often the user manually observes as the program executes the associated input and hopes to identify the root cause. The user then decides whether the root cause represents a new flaw or vulnerability.

Often fuzzers output an overwhelming number of interesting program states. The user must perform *triage* to decide which interesting program states merit further investigation. Unfortunately, output program states often share the same root cause. Even worse, states that share the same root cause can appear drastically different. For example, a memory corruption vulnerability can result in crashes in many different parts of the program, with wildly different memory images, because the effect of the vulnerability is not observed immediately. Some automated tools attempt to deduplicate fuzzer outputs [23] or root causes, but these are often imperfect, and triage and root cause analysis remain research challenges.

Comparing Fuzzers

Unfortunately, despite all the research into making fuzzers more effective and efficient, it is hard to determine whether one fuzzer is "better" than another. Measuring the utility of a fuzzer is difficult for many reasons, from inability to explore the entire input search space which biases performance measures; to lack of ground truth, which makes validation a difficult and manual process; to randomness exploited in the fuzzing process, which introduces the need for statistical testing. For example, in vulnerability analysis, fuzzers are often judged by number of unique vulnerabilities; however, when a new fuzzer finds new vulnerabilities in a set of real-world programs, this simply shows that the fuzzer's core algorithm is better suited to that set of programs than other algorithms [41]. Results do not necessarily generalize well.

To help judge fuzzer utility, Klees recently proposed a framework for fuzzer comparison [24]. Klees calls for controlling comparisons by specifying: a baseline fuzzer, a benchmark suite of target programs, a performance metric, configuration parameters comparable to the baseline, and sufficient number of trials for statistical testing [24]. Small data sets with known flaws can be used to compare and validate fuzzers in this way. Unfortunately, however, these comparisons are computationally expensive, and, as mentioned, results do not necessarily generalize. Practically, until thorough benchmarks for this framework have been developed, users will have to rely on their intuition to guide decisions about which fuzzer to use for a particular program, how to configure or adapt that fuzzer, and whether that fuzzer performs well enough. Effective fuzzer comparison remains a significant research challenge.

3 APPLICATIONS OF MACHINE LEARNING TO FUZZING

In this section, we explore research applying machine learning (ML) to the fuzzing process. ML has been used to generate new inputs in the fuzzing process and, to a lesser extent, to improve post-fuzzing. Unsupervised learning has seen the most successful applications to input generation, with fuzzing tools such as AFL [28] integrating genetic algorithms into the input generation process. There are also recent applications of both supervised and reinforcement learning to input generation. Additionally, all three types of ML have been applied to symbolic execution [8, 10, 33, 42, 52, 54], primarily to reduce constraint equation solve times. Both supervised and unsupervised learning have been applied to post-fuzzing processes primarily for crash triage and root cause categorization [11, 27, 32]. Interestingly, we know of no research in three areas: input minimization, corpus minimization, and seed selection. These pieces of the fuzzing process tend not to be large bottlenecks, which may account for the lack of research.

We begin by discussing ML research for input generation. We then discuss applications of ML to post-fuzzing tasks. Finally, we conclude by discussing fuzzing tasks that have not seen ML applications. For each of these areas, we show how each type of ML offers unique methods for improvement as well as the difficulties of applying each type.

Input Generation

The most successful applications of ML to fuzzing occur in the input generation stage. In this section we will discuss how the various types of ML have been applied to input generation. We will show that unsupervised learning, in the form of genetic algorithms and deep learning, has had many successful applications to input generation. We also show the applications of reinforcement learning and supervised learning to various types of input generation, although this research tends to be more exploratory. This section will also discuss the distinct advantages offered by each ML type to the input generation problem, as well as the difficulties of applying each type.

Genetic Algorithms. The most frequently used ML technique for input generation is the genetic algorithm (GA) [13, 14, 17, 28, 30]. GAs, a type of unsupervised ML inspired by biological evolution, provide the core algorithms in evolutionary fuzzers. GAs take a small base population of inputs, make small transformations, and keep the transformations that improve upon the previous input. This iterative nature allows GAs to take an input seed and mutate pieces of it to explore the code space. GAs have been successful for input generation due to this ability to build off previous successful inputs to further explore the code space.

As mentioned in Section 2, evolutionary fuzzers use a fitness function to rank inputs for selection and mutation. The choice of fitness function can have a tremendous impact on 1) the performance of the fuzzer, 2) the ability of the fuzzer to identify certain types of bugs, and 3) the tendency to get stuck in a local minimum. Thus, particular care must be taken when choosing a fitness function. While code coverage is the most common metric used for the fitness function, more advanced heuristics such as the Dynamic Markov Model heuristic have also been used [45]. This heuristic allows the fuzzer to take into account previous mutations which are used to guide the fuzzer along a path to a suspected vulnerable region. In contrast to fitness functions which use code coverage, the Markov Model heuristic steers the fuzzer towards regions of the code that exhibit vulnerabilities. In this way, we see that the fitness function determines the overall behavior and objectives of the fuzzer. However, different users may desire different fuzzer behaviors. Thus, a promising area of research may be the development of new fitness functions which create the desired fuzzer behaviors.

Deep Learning: Unsupervised and Supervised Applications. Deep learning (DL) has also been applied to input generation, specifically to improve generation-based and mutation-based fuzzers. Addressing an often manual and time-intensive task, Microsoft automatically generated the input grammar for generation-based fuzzers using Long-Short Term Memory (LSTM) networks, neural networks often used for sequence analysis [20]. Separately, Microsoft compared four different DL architectures, each attempting to increase mutation-based fuzzer coverage by identifying promising input bytes to mutate [3]. This research compared a standard LSTM; a bidirectional LSTM, which processes input sequences both forwards and backwards; a sequence-to-sequence model known as Seq2Seq, which transforms a sequence from one form to another; and a variant of Seq2Seq that uses an attention mechanism to focus on the most important parts of an input. While all models increased code coverage in some situations, LSTMs slightly outperformed the other models overall. Their method also provides an alternative to GAs for mutation of inputs for evolutionary fuzzers.

While both of these studies by Microsoft show promise for DL applications to fuzzing, the ability to transfer DL models between programs remains a challenge. DL models require significant computation time for training; they are not integrated into the fuzzing process as GAs are. Re-training a model for every new program is likely too costly to be practical. Further, it is not yet clear whether a DL model trained on one type of software will transfer well to new software. Thus, DL will likely benefit fuzzing once models can be transferred effectively to other programs.

Reinforcement Learning. Several groups have also applied reinforcement learning (RL) to input generation. Becker et al. created an RL agent which learned to fuzz the IPv6 protocol by mutating messages sent to the host [2]. Bottinger et al. transformed the fuzzing problem into a Markov Decision Process. They used a deep Q-learning network to learn a grammar describing inputs for generation-based fuzzers [7].

These two applications of RL to fuzzing provide insight into effectively defining a reward function for an agent, often the most challenging aspect of RL. Often, multiple criteria must be considered when defining a reward function. For instance, Becker et al. created a multi-part reward function based on the following criteria: number of program functions called from a single input, presence of an error, and potential corruption or delay of the response message from the program [2]. Each part of the reward function played a critical role in defining the behavior of the RL agent. The presence of the error acted as the strongest signal to the agent that it had reached an interesting part of the code space. Additionally, the program response can be used to guide the agent even in the absence of the error signal. In this way each part of the reward function plays a unique role in guiding the agent. Leaving out any of these criteria can result in an ineffective agent due to incomplete information. Thus, when defining a reward function, it is important to consider the different types of program signals that can be used.

Bottinger et al., on the other hand, experimented with multiple distinct reward functions, one using code coverage, another using execution time, and a third combining code coverage and execution time. Unsurprisingly, the reward function influenced the fuzzer's exploration of the input space; for instance, when Bottinger used execution time as a reward, the agent learned inputs that caused the program to terminate quickly. In this way, we see that a user can influence fuzzer behaviors by crafting a particular reward function. Overall, reward functions must be defined carefully, taking into account at least the software program type, type of bugs being sought, and fuzzing metrics available.

Standard fuzzers do not yet implement RL, and applications remain theoretical. As in DL, understanding the transferability of an RL agent remains a research challenge. Currently, it is not clear if it is necessary to train a new RL agent for each new program that can be encountered by the fuzzer. It is likely that training a new RL agent for each new program instance would negate any benefits, therefore creating transferable agents is an important step for future research.

Machine Learning for Symbolic Execution. Here, we briefly touch on several different ML techniques currently being explored to improve symbolic execution. As discussed in Section 2, symbolic execution can help generate effective new inputs for fuzzers, but computational cost and path explosion remain significant hurdles. Several research efforts in ML explore the feasibility of improving constraint solving, which could support symbolic execution for fuzzer input generation. Unfortunately, current efforts in using ML do not rival state-of-the-art approaches using graph algorithms [25]; they are merely explorations in feasibility.

Several research efforts use supervised learning to solve constraint equations. Graph Neural Networks were used to identify features indicating whether constraint equations had valid solutions or not [8]. In another study, Wu used a combination of logistic regression and Monte Carlo methods to identify initial values that increased the probability of finding a valid solution to a constraint equation. The Monte Carlo methods were used to identify initial promising values, while logistic regression was used to indicate the validity of the solution to the constraint equation using these chosen values. Incorporating these new initial values led to decreased runtimes for the Minisat solver [54]. In another study, LSTMs (i.e., DL) were trained to solve constraint equations [42]. While the LSTMs were not able to beat state-of-the-art constraint solvers, they were able to solve constraint equations from domains they were not trained on, indicating an ability for ML models to generalize in this space. In general, each of these studies offers unique ways of solving constraint equations. While not currently state-of-the-art, they nonetheless offer a strong starting point for using supervised learning to decrease the amount of computation time required to solve constraint equations.

We know of only one research effort using RL for solving constraint equations. In this study, Mairy et al. used RL to improve local neighborhood search methods [33]. Local neighborhood search methods iteratively find solutions to a constraint equation by finding solutions to various subsets of the constraint equation and combining the subsets to form the final solution. In order to discover useful subsets, these local neighborhood search methods must intelligently explore the space of possible subsets. RL is particularly well suited for this exploration, since the RL agent can be guided to choose subsets that are more likely to lead to a valid solution. This study also indicates that RL may play a useful role for solving constraint equations in general. The search space of possible solutions is too large to search naively however, RL methods offer a way to guide the exploration and thus reduce the time needed to find a solution.

Several research efforts explored the feasibility of using GAs to solve constraint equations. GAs were used to find optimal solutions to constraint problems given a small number of feasible solutions [10, 52].[3] In contrast to this work, fuzzers prefer solutions that discover new code paths over optimal solutions for known code paths. However, this work can benefit fuzzers directly if fitness functions apply heavier weights to feasible solutions as explored by Venkatraman and Yen [52]. Finding optimal solutions does benefit fuzzers as well: optimal solutions reduce input length, thereby reducing time spent in input generation.

[3] Finding a small set of feasible solutions is usually straightforward in fuzzing. The user can provide a small set of paths using his program knowledge, or a fuzzer run for a small number of iterations can find a small set of feasible paths.

One research effort used ML to reduce the size of the search space rather than to directly solve constraint equations. Li et al. reformulate the typical collection of path constraints as an optimization problem and attempt to reduce the number of infeasible paths, i.e., paths that can never be reached due to conflicting constraints [29]. They use the ML technique RACOS [57], a technique for optimization that scales well to high dimensional problems, to solve the optimization problems. However, only very small programs of up to 335 lines were analyzed. In general, fuzzers tend to generate a large number of infeasible paths, reducing the efficiency of symbolic execution. Thus, techniques employed by Li et al. may directly benefit fuzzers by reducing the number of infeasible paths and allowing more extensive use of symbolic execution.

Initial explorations in improving symbolic execution by applying ML show promise. However, the practical utility of applying ML here remains to be seen. The current research is typically on small problems and the methods shown in this section do not beat the state-of-the-art. Thus, improving both scalability and performance of ML techniques used for symbolic execution and constraint solving are important open research challenges.

Post-Fuzzing: Interesting Program States

As mentioned in Section 2, users triage and then analyze interesting program states output by the fuzzer, often manually. Users triage program states by 1) evaluating for *uniqueness* [38], 2) analyzing triaged states to determine *reproducibility* and a root cause, and, 3) when fuzzing for vulnerability assessment, determining whether a root cause is exploitable. ML has primarily been used to categorize crashes (triage) or to categorize bugs (root cause analysis), although Yan et al. used Bayesian methods and the *!exploitable* tool to improve reliability in determining bug exploitability [55].

Dang et al. triaged crashes by grouping crashes with similar call stacks using agglomerative hierarchical clustering, an unsupervised learning technique that clusters data points with similar features [11]. They introduced their own similarity metric over call stacks, the position-dependent model, allowing them to use an unlabeled call stack data set for training. They tested their model over various Microsoft products and in many cases outperformed previous methods for crash similarity identification.

Harsh et al. experimented with root cause categorization using supervised, unsupervised, and semi-supervised techniques [27]; unfortunately, their techniques are limited as their categories are both extremely broad and system-specific. Long et al. identified root causes and automatically generated patches using probabilistic models [32].

Outside of the software domain, supervised ML techniques have been used to identify root causes more frequently [43]. Specifically, decision trees and support vector machines were used for root cause analysis on industrial production systems [12]. Neural networks were used for fault localization within industrial tank systems [44]. Support vector machines showed promise in speeding up fault localization on circuit boards [56]. While none of this research was directly applied to software, it may be extended to aid root cause analysis in the software domain.

Challenges. ML is rarely applied to post-fuzzing tasks for two reasons: 1) ML results and algorithms are often difficult to interpret, and 2) appropriate training data sets are sparse. For the first, classification techniques only return a prediction, not an explanation. This makes it particularly difficult for a user to determine whether a predicted label is correct and why that label was applied. For example, in root cause analysis, the user would find it difficult to understand where the root cause supposedly manifested in the code, either to validate the label or to correct the root cause [27]. Further, ML algorithms often build up opaque rules that are difficult to map into domain knowledge. For the second, we have very few available labeled data sets, and it is not yet clear what constitutes a strong, generalizable benchmark data set. Which of all the possible bugs should be included? Which programming languages should be represented? How should we encode a bug and its root cause for an ML algorithm, especially given that root causes are nuanced, root causes can vary from system to system, and a bug in one context may not be a bug in another? Significant research challenges remain in applying ML to post-fuzzing tasks, specifically improving interpretability of ML models and creating effective training data sets.

Input and Corpus Minimization

To our knowledge, there has not been any research into *input minimization* or *corpus minimization* using ML. There are several reasons for this lack of research. First, neither *input minimization* nor *corpus minimization* is a large bottleneck. The largest bottlenecks exist in the input generation and post-fuzzing process thus, most research tends to concentrate on these areas. Second, minimization of input or total corpus sizes does not naturally lend itself to ML techniques. While input generation and, to a lesser extent, post-fuzzing are often able to be formulated as ML problems, minimization is often achieved successfully using heuristic methods [37].

Seed selection

To our knowledge, there has not been any research in *seed selection* using ML. However, prior *seed selection* research does point to the possibility of future ML research. *Seed selection* must balance using current inputs with a known level of performance versus exploring new inputs with an unknown but potentially better performance [53]. Reinforcement learning algorithms are often applied successfully in these types of scenarios which require a balance of both exploration of new inputs and utilization of current inputs. Thus, reinforcement learning algorithms may be well suited for determining an optimal seed schedule.

4 CONCLUSION

In this survey, we discussed how machine learning (ML) has been applied to fuzzing. Because fuzzing problems lend themselves more naturally to unsupervised and reinforcement learning techniques, supervised learning is rarely used to support the fuzzing process. ML is most often used to support the *Generate Inputs* stage of the fuzzing process. Unsupervised methods currently offer the most benefit with tools such as AFL making unsupervised algorithms part of their workflow. In contrast, reinforcement learning and deep learning are being explored for possible improvements but are not yet part of any standard fuzzing tool. ML has also been applied to analyze *Interesting Program States* during post-fuzzing, helping to triage crashes and support root cause analysis. However, ML has not been applied to the *Select Inputs* stage of the fuzzing process, possibly because this stage is not a major bottleneck. Additionally ML has not been applied to evaluating reproducibility of crashes during post-fuzzing. The lack of ML research in certain portions of the fuzzing process may be due to ML being less understandable than other heuristic approaches, or it may be due to a lack of accessible training data.

While ML has played an important role in the functionality of fuzzing systems, there remain many open research challenges. Recently an influx of researchers have begun dedicating resources to address some of these challenges. Fuzzing will continue to play an important role in vulnerability assessment in the future. As research in this area grows, we expect to see the continued application of ML to address the bottlenecks of the fuzzing process.

ACKNOWLEDGMENTS

The authors would like to thank Danny Loffredo for insights into the practical application of fuzzing processes and Christopher Harrison for helpful discussions on symbolic execution. The authors thank Michelle Leger for technical edits and discussions. This work is funded in part by Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

REFERENCES

[1] Pedram Amini and Aaron Portnoy. 2015. Sulley: Fuzzing Framework. https://github.com/OpenRCE/sulley

[2] Sheila Becker, Humberto Abdelnur, Radu State, and Thomas Engel. 2010. An Autonomic Testing Framework for IPv6 Configuration Protocols. In *Mechanisms for Autonomous Management of Networks and Services*, Burkhard Stiller and Filip De Turck (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 65–76.

[3] William Blum, Mohit Rajpal, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. (November 2017). https://www.microsoft.com/en-us/research/publication/not-all-bytes-are-equal-neural-byte-sieve-for-fuzzing/

[4] Hanno Böck. 2015. How Heartbleed could've been found. https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html blog.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *23rd ACM Conference on Computer and Communications Security (CCS)*.

[7] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. [n. d.]. Deep Reinforcement Fuzzing. ([n. d.]). arXiv:1801.04589 http://arxiv.org/abs/1801.04589

[8] Benedikt Bünz and Matthew Lamm. 2017. Graph Neural Networks and Boolean Satisfiability. *CoRR* abs/1702.03592 (2017). arXiv:1702.03592 http://arxiv.org/abs/1702.03592

[9] Andrew Case, Arghya Kusum Das, Seung-Jong Park, J. (Ram) Ramanujam, and Golden G.Richard III. 2017. Gaslight: A comprehensive fuzzing architecture for memory forensics frameworks. In *2017 17th Annual DFRWS (Volume 22)*. Digital Investigation, USA, 86–93.

[10] Adam Chehouri, Rafic Younes, Jean Perron, and Adrian Ilinca. 2016. A Constraint-Handling Technique for Genetic Algorithms using a Violation Factor. *CoRR* abs/1610.00976 (2016). arXiv:1610.00976 http://arxiv.org/abs/1610.00976

[11] Yingnong Dang, Rongxin wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. (06 2012), 1084–1093.

[12] M. Demetgul. 2013. Fault diagnosis on production systems with support vector machine and decision trees algorithms. *The International Journal of Advanced Manufacturing Technology* 67, 9 (01 Aug 2013), 2183–2194. https://doi.org/10.1007/s00170-012-4639-5

[13] Jared D. DeMott, Richard J. Enbody, and William F. Punch. 2007. Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. (2007).

[14] Fabien Duchene. 2013. Fuzz in the Dark: Genetic Algorithm for Black-Box Fuzzing. In *Black Hat 2013*. Black Hat, Sao Paulo, Brazil. https://hal.inria.fr/hal-00978844

[15] Joe W. Duran and Simeon Ntafos. 1981. A Report on Random Testing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 179–183. http://dl.acm.org/citation.cfm?id=800078.802530

[16] Joe W. Duran and Simeon C. Ntafos. 1984. An Evaluation of Random Testing. *IEEE Trans. Softw. Eng.* 10, 4 (July 1984), 438–444. https://doi.org/10.1109/TSE.1984.5010257

[17] James Fell. 2017. *A Review of Fuzzing Tools and Methods*. Technical Report. https://dl.packetstormsecurity.net/papers/general/a-review-of-fuzzing-tools-and-methods.pdf.

[18] RenÃľ Freingruber. 2017. Hack the Hacker - Fuzzing Mimikatz on Windows with WinAFL & Heatmaps (0DAY). https://www.sec-consult.com/en/blog/2017/09/hack-the-hacker-fuzzing-mimikatz-on-windows-with-winafl-heatmaps-0day/ blog.

[19] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[20] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn & Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 50–59. http://dl.acm.org/citation.cfm?id=3155562.3155573

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[22] Google. 2016. Honggfuzz. https://github.com/google/honggfuzz

[23] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic Heap Layout Manipulation for Exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/usenixsecurity18/presentation/heelan

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS âĂŽ18). In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS âĂŽ18)*. ACM, Toronto, ON, Canada. https://arxiv.org/pdf/1808.09700.pdf

[25] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. 2010. Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs. In *Proceedings of the 2010 19th IEEE Asian Test Symposium (ATS '10)*. IEEE Computer Society, Washington, DC, USA, 59–64. https://doi.org/10.1109/ATS.2010.19

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[27] H. Lal and G. Pahwa. 2017. Root cause analysis of software bugs using machine learning techniques. In *2017 7th International Conference on Cloud Computing, Data Science Engineering - Confluence*. 105–111.

[28] lcamtuf. 2014. afl-fuzz: crash exploration mode. https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html blog.

[29] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 554–559. https://doi.org/10.1145/2970276.2970364

[30] Guang-Hong Liu, Gang Wu, Zheng Tao, Jian-Mei Shuai, and Zhuo-Chun Tang. 2008. Vulnerability Analysis for X86 Executables Using Genetic Algorithm and Fuzzing. In *2008 Third International Conference on Convergence and Hybrid Information Technology*, Vol. 2. 491–497. https://doi.org/10.1109/ICCIT.2008.9

[31] LLVM. 2017. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html

[32] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *SIGPLAN Not.* 51, 1 (Jan. 2016), 298–312. https://doi.org/10.1145/2914770.2837617

[33] Jean-Baptiste Mairy, Yves Deville, and Pascal Van Hentenryck. 2011. Reinforced Adaptive Large Neighborhood Search. Doctoral Program at the Interational Conference on Principles and Practice of Constraint Programming, 55–60.

[34] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[35] Jan Obdrzálek and Marek Trtík. 2011. Efficient Loop Navigation for Symbolic Execution. *CoRR* abs/1107.1398 (2011). arXiv:1107.1398 http://arxiv.org/abs/1107.1398

[36] Peter Oehlert. 2005. Violating Assumptions with Fuzzing. *IEEE Security and Privacy* 3, 2 (March 2005), 58–62. https://doi.org/10.1109/MSP.2005.55

[37] Tavis Ormandy. 2010. Making Software Dumber. http://taviso.decsystem.org/making_software_dumber.pdf Presentation.

[38] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. 2015. Hercules: Reproducing Crashes in Real-world Application Binaries. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 891–901. http://dl.acm.org/citation.cfm?id=2818754.2818862

[39] Luc De Raedt, Siegfried Nijssen, Barry OâĂŹSullivan, and Pascal Van Hentenryck. 2011. Constraint Programming meets Machine Learning and Data Mining. In *Dagstuhl Seminar*.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos (Eds.). 2017. *VUzzer: Application-aware evolutionary fuzzing*. Proceedings of the Network and Distributed System Security Symposium (NDSS). http://www.cs.vu.nl/~giuffrida/papers/vuzzer-ndss-2017.pdf

[41] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 861–875. http://dl.acm.org/citation.cfm?id=2671225.2671280

[42] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2018. Learning a SAT Solver from Single-Bit Supervision. *CoRR* abs/1802.03685 (2018). arXiv:1802.03685 http://arxiv.org/abs/1802.03685

[43] Marc Solé, Victor Muntés-Mulero, Annie Ibrahim Rana, and Giovani Estrada. 2017. Survey on Models and Techniques for Root-Cause Analysis. *CoRR* abs/1701.08546 (2017). arXiv:1701.08546 http://arxiv.org/abs/1701.08546

[44] Timo Sorsa and Heikki N. Koivo. 1993. Application of artificial neural networks in process fault diagnosis. *Automatica* 29 (7 1993), 843–849. Issue 4.

[45] Sherri Sparks, Shawn Embleton, Ryan K Cunningham, and Cliff Changchun Zou. 2007. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 477–486. https://doi.org/10.1109/ACSAC.2007.27

[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. (01 2016).

[47] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3104–3112. http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[48] Ari Takanen, Jared Demott, and Charlie Miller. 2008. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc. http://index-of.co.uk/Reversing-Exploiting/Fuzzing.pdf.

[49] Peach Tech. 2018. Peach. https://www.peach.tech/products/peach-fuzzer/

[50] Valgrind. 2017. Valgrind tool. http://valgrind.org

[51] Mikko Vampari. 2015. *An Evaluation of Free Fuzzing Tools*. Master's thesis. University of Oulo.

[52] S. Venkatraman and G. G. Yen. 2005. A Generic Framework for Constrained Optimization Using Genetic Algorithms. *IEEE Transactions on Evolutionary Computation* 9, 4 (Aug 2005), 424–435. https://doi.org/10.1109/TEVC.2005.846817

[53] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security (CCS '13)*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2508859.2516736

[54] Haoze Wu. 2017. Improve SAT-solving with Machine Learning. *CoRR* abs/1710.11204 (2017). arXiv:1710.11204 http://arxiv.org/abs/1710.11204

[55] Guanhua Yan, Junchen Lu, Zhan Shu, and Yunus Kucuk. 2017. ExploitMeter: Combining Fuzzing with Machine Learning for Automated Evaluation of Software Exploitability. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*. 164–175. https://doi.org/10.1109/PAC.2017.10

[56] F. Ye, Z. Zhang, K. Chakrabarty, and X. Gu. 2014. Board-Level Functional Fault Diagnosis Using Multikernel Support Vector Machines and Incremental Learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 2 (Feb 2014), 279–290. https://doi.org/10.1109/TCAD.2013.2287184

[57] Yang Yu, Hong Qian, and Yi-Qi Hu. 2016. Derivative-Free Optimization via Classification *(Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence)*.

[58] Michal Zalewski. 2017. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/