Computers & Security 120 (2022) 102813

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Fuzzing vulnerability discovery techniques: Survey, challenges and future directions

Craig Beaman, Michael Redbourne, J. Darren Mummery, Saqib Hakak*

Canadian Institute for Cybersecurity, Faculty of Computer Science, University of New Brunswick, Fredericton, Canada

ARTICLE INFO

Article history:
Received 17 August 2021
Revised 22 June 2022
Accepted 27 June 2022
Available online 28 June 2022

Keywords:
Vulnerability
Fuzzing
Software Security
Fuzzers
Software Vulnerability
Vulnerability accessment
Static code analysis
Security

ABSTRACT

Fuzzing is a powerful tool for vulnerability discovery in software, with much progress being made in the field in recent years. There is limited literature available on the fuzzing vulnerability discovery approaches. Hence, in this paper, an attempt has been made to explore the recent advances in the area of fuzzing vulnerability discovery and to propose a refinement to the classification of fuzzers. Furthermore, we have identified key research challenges and potential future areas of research that might provide new insight to researchers.

1. Introduction

Vulnerability assessment of software is a critical part of software security (Liu et al., 2012). However, manually finding vulnerabilities in software can be challenging and time-consuming, especially as software complexity increases. There are also cases where direct access to the software source code is not available, which makes manual review extremely difficult, if not impossible. Hence, there is a need for intelligent tools capable of automatically finding vulnerabilities in software. One such automated method is fuzzing, which is an efficient and effective approach to vulnerability discovery (Hu et al., 2021; Lin et al., 2020; Ozment, 2007; Zeng et al., 2020).

Fuzzing is the creation of input, called fuzz, where this input is passed to a program in order to test that program for issues. This fuzz is created to find unexpected program behavior, such as bugs or vulnerabilities, and as such, it is often created in a pseudo-random way. Some examples of the types of vulnerabilities that can be found through fuzzing include cross-site scripting (Hydara et al., 2021), buffer overflow (Mihailescu and Nita, 2021), and denial of service vulnerabilities (Onl, 2010; Salim et al., 2020). The generated fuzz input can take many forms, from files and strings to network communication data and executable binaries (Li et al., 2018). The fuzzed input is typically ideal if it is not immediately rejected by the program, as this allows for deeper code coverage and more thorough testing.

Fuzzing was first introduced in the early 1990s as the result of a class project at the University of Wisconsin (Takanen et al., 2018). This project wanted to see how UNIX programs were able to cope with randomly generated input. What that project found was that roughly one-third of the applications would crash while being fuzzed. Such a result suggested a real need for this sort of testing to be incorporated into the standard suite of software testing procedures. This early form of testing was relatively simple as it did not take into account the types of programs it was testing, it simply randomly created values that would then be entered into the software. Since then, fuzzing has evolved in many ways to become far more sophisticated and powerful.

1.1. Examples of fuzzers

A great example to illustrate the power of fuzzing is Google's open-source project, OSS-Fuzz (Serebryany, 2017), which supports C, C++, Rust, GO, and Python-based projects. OSS-Fuzz can utilize various fuzzing engines and operates in a distributed environment to automatically find and report software vulnerabilities. Since its creation, it has found thousands of such vulnerabilities in hundreds of various open-source projects (Aizatsky, 2016).

There are hundreds of different fuzzer's out there, some of which are described in literature, Banks et al. (2006);

* Corresponding author.
E-mail address: saqib.hakak@unb.ca (S. Hakak).

https://doi.org/10.1016/j.cose.2022.102813
0167-4048/

Table 1
A list of various popular fuzzing programs, along with their type.

Name of fuzzer	Knowledge of application structure
BFF	Black-box
Radamsa	Black-box
GLADE	Black-box
zuff	Black-box
AFL	Grey-box
AFLFast	Grey-box
LibFuzzer	Grey-box
BuzzFuzz	White-box
Chopper	White-box

Blotsky et al. (2018); Chen et al. (2018a); Jayaraman et al. (2009); Veggalam et al. (2016); Wang et al. (2020a); Wüstholz and Christakis (2020). Each fuzzer program has a different feature set and differing objectives. A list of a few popular fuzzer's and their types were found in Manès et al. (2019) and are listed in Table 1.

There is limited literature available on recent developments in fuzzers. Although, there are a few surveys as highlighted in Table 2, but most of the work is outdated. In this article, we have tried to explore more recent advancements in the area of vulnerability discovery approaches.

The main contributions of this article are as follows:

- Recent advances on fuzzing vulnerability discovery approaches are presented.
- A refined classification of fuzzers compared to past surveys is provided.
- Potential research challenges are highlighted and future research directions are identified.

The rest of the paper is organized as follows: Section 2 describes the various ways in which a fuzzer can be classified. Section 3 presents recent advances in vulnerability discovery approaches. Section 4 provides a discussion on the findings from the literature review, and Sections 5 and 6 go over some of the identified research challenges and future research directions. Section 7 concludes the paper.

2. Classification of fuzzers

In general, fuzzers can be classified based on a few different aspects, which are: whether test case generation is updated based on program execution, knowledge of application structure, approach to code exploration, and the fuzzer's method of generating new inputs (Li et al., 2018). It is to be noted that these classifications are not necessarily independent from each other. There may be some cross-over between terms e.g. for a generation-based fuzzer, knowledge of input structure is required (Li et al., 2018). Hence, a generation-based fuzzer cannot also be a black-box fuzzer. These classifications simply serve as a guide for the common terminology seen when classifying fuzzers. Fig. 1 illustrates the different classifications of fuzzers.

A brief description of all categories is as follows:

2.1. Test case feedback

Whether test case generation is updated based on program execution or not leads to two possible classifications. These classifications are dumb and smart fuzzers. Dumb fuzzers simply pick a test case generation approach and stick with that. The first generation of fuzzer's tend to fall into this category. Dumb fuzzers have the advantage of not being specific to any given program and hence can be easily applied to many different programs with relative ease. The disadvantage of dumb fuzzers is that their test cases will less often lead to bug or vulnerability discovery, which in turn increases testing time. Today, dumb fuzzers are still used, but research has mostly focused on smart fuzzing. Smart fuzzers make more intelligent choices for the inputs as to best test a particular program. This is done by modifying the input generation process according to program behavior (Li et al., 2018). For example, if it is noted that input seeds containing the string "abc" reach further into the program, then the fuzzer may make it more likely for future seeds to contain this string. A dumb fuzzer in this case would not change its seed generation approach based on this information.

2.2. Knowledge of application structure

Knowledge of Application structure refers to what test program information is utilized by the fuzzer. This can be divided into three categories which are white-box, grey-box, and black-box types. In particular, these categories are based upon the amount of source code or execution state information that is needed by the fuzzer to complete its tests (Manès et al., 2019).

Black-box fuzzers do not have any knowledge of the application's structure and many early types of fuzzers fall into this category. These types of fuzzers simply look at the input and corresponding output of the test program. No analysis of the code execution paths within the program is made to help guide the fuzzer. On the other hand, white-box fuzzers are based on a technique known as symbolic execution (Cadar and Sen, 2013). Using symbolic execution, program analysis of the tested program is done to systematically increase the code coverage. This guides the fuzzer by looking at how different inputs affect the internal program state (Boehme et al., 2021; Li et al., 2018). Grey-box fuzzers fall somewhere in-between white-box and black-box fuzzers in that they may use program source code or execution information, but this in a more limited way than with white-box fuzzers. For example, grey-box fuzzers can use program analysis to help guide the fuzzing progress. Program analysis includes things like code coverage information from the execution state of the test program to help guide the fuzzing process (Manès et al., 2019).

Table 2
A summary of previous surveys on fuzzing.

Reference	Year	Description
Li et al. (2018)	2018	Gives a detailed background on fuzzing classification and alternatives. Classifies research based upon the challenge that the research addresses. Also looks at some of the fuzzing research directed towards specific domains.
Manès et al. (2019)	2019	Developed a general algorithm to describe fuzzers and from that categorized 63 different fuzzing programs. Also categorized some fuzzing research based upon black-box, white-box, or grey-box structure and then subdivided from there based upon the area where the research is applicable.
Saavedra et al. (2019); Wang et al. (2020b)	2019, 2020	These studies look into how machine learning has been applied to fuzzing.
Böhme et al. (2020)	2020	Provides an in-depth overview of the challenges in fuzzing and raises a number of important questions which ought to be answered by future studies.
Eceiza et al. (2021)	2021	Provides an in-depth review of fuzzing for embedded systems, such as for IoT devices. Analyzes 41 fuzzers in terms of their ability to effectively fuzz embedded systems.

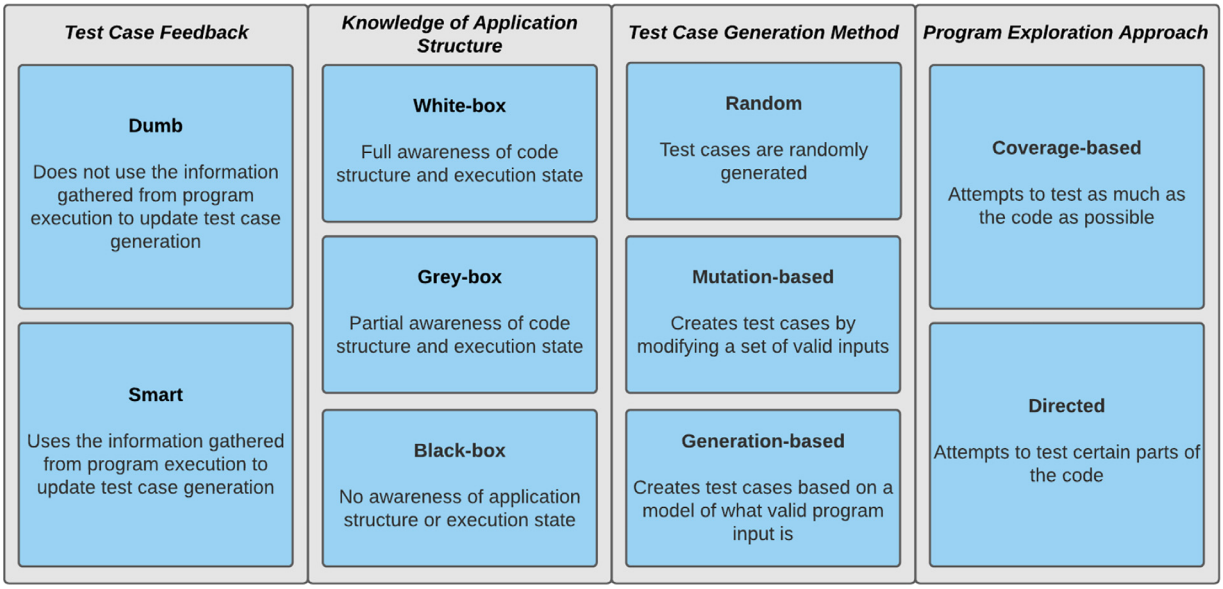

Fig. 1. The various ways in which a fuzzer can be classified.

2.3. Test case generation method

Another aspect of consideration when classifying fuzzers is based upon how they generate their test cases. This includes classifications like mutational-based, generational-based, and random fuzzers. Mutational-based fuzzers generate their test cases by mutating a set of initially valid inputs to the program, which are called "seeds". An example seed could be a valid MP3 file that contains a song. The mutations are applied to the seed and include actions such as randomly flipping bits or shifting parts of the input around. Generation-based fuzzers use a blueprint (input model, such as a grammar) in order to build valid input. For example, if the test program takes a JPG image file, then a generation-based fuzzer could be fed a configuration file for how to generate valid JPG files. The final approach to test case generation is randomly generating test cases. This approach is less often used as it is more likely to be immediately rejected by a program (Hsu et al., 2018).

2.4. Program exploration approach

How a fuzzer goes about exploring a program leads to two classifications. These classifications are directed and coverage-based fuzzers. Directed fuzzers attempt to generate test cases such that the test cases reach specific parts of the program. Coverage-based fuzzers create test cases as to test as much of the code as possible. Both are similar in that they use code-coverage metrics, but differ in their focus. A comparison can be made between search algorithms and program exploration approaches. Coverage-based fuzzers can be thought to use a "breadth first search" approach while directed-based fuzzers use a more "depth first search" approach. Directed fuzzers tend to be faster than coverage-based fuzzers to reach a particular area of code, but coverage-based fuzzers can potentially find a greater proportion of the total bugs in a software system (Li et al., 2018).

Coverage-based fuzzers can use different metrics for keeping track of their progress and for guiding seed generation. These metrics include line, basic block, basic branch, N-gram branch, context-sensitive branch, and memory-access-aware branch coverage (Wang et al., 2019b). Line coverage tracks progress based on whether or not the fuzzer is triggering new lines of code. Basic block coverage tracks whether or not the fuzzer is triggering new blocks, where one block is defined by the fuzzer (some small amount of code). Basic branch coverage takes basic block coverage one step further and considers not only what blocks have been visited, but also the previously visited blocks. This tuple of the current and previously visited block are tracked and used to measure progress. N-gram branch coverage further generalizes this metric and looks at the last N blocks visited by the fuzzer, not just the last two. When N is 0, this metric is equivalent to basic block coverage as only the current block is considered. When N is large, the complete path of visited blocks is considered and hence this metric is often called a path coverage metric. Such metrics are often used in white and grey-box fuzzers and the information is often obtained via code instrumentation. An example of code instrumentation is given in Fig. 2.

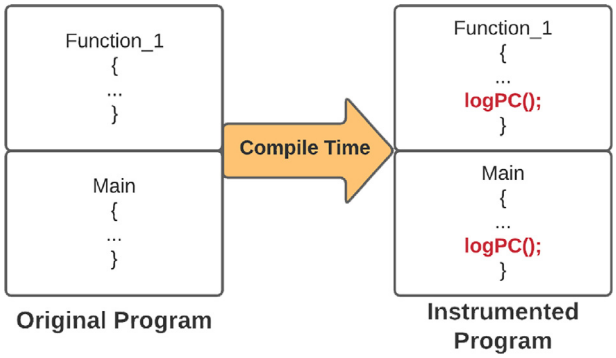

Fig. 2. A simple example of instrumentation. In this example, the "logPC" function is the instrumentation code and this logs the value of the program counter. This allows for a fuzzer to track code coverage as the program executes for each fuzz value.

The other two main metrics for coverage guided fuzzing are context-sensitive branch and memory-access-aware branch coverage (Wang et al., 2019b). Context-sensitive branch takes into consideration both the branches covered by the fuzzer, as well as the data present in the call to that branch. This extra information increases the sensitivity of the fuzzer to otherwise missed details that could potentially lead to more discovered vulnerabilities. Finally, memory-access-aware branch coverage strictly uses memory access or state information to track progress, which can help to identify things such as memory corruption vulnerabilities.

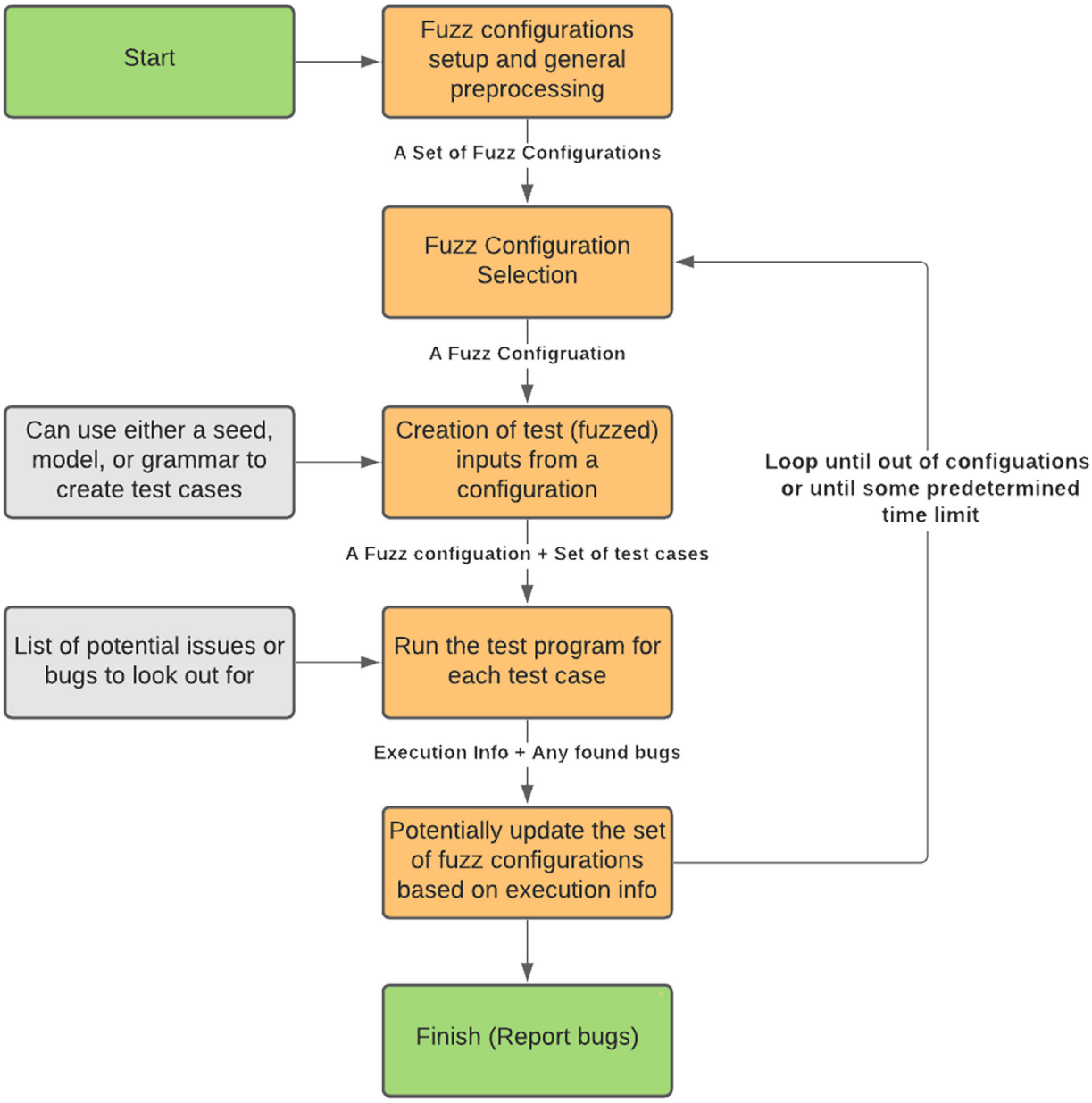

Fig. 3. The general steps involved in fuzzing a program.

An overview of the steps involved in fuzzing can be found in Fig. 3. This model is general such that it can be applied to either white-box, grey-box, or black-box fuzzers. A more detailed look into the steps used by fuzzers can be found in Manès et al. (2019).

3. Literature review

In this section, an overview of recent research on fuzzing is presented. Research is divided into different subsections based upon the goal of the research. The four identified research areas are improved code coverage (general), improved code coverage (complex input), improved directed code coverage, and improved domain-specific fuzzing. Improved code coverage (general) refers to research that improved fuzzing in the general sense; papers in this category did not focus on a specific test program input format. Improved code coverage (complex input) includes papers that sought to improve the performance of fuzzers that are fuzzing test programs that take complex input types, such as PDF readers. Improved directed code coverage research improved how fuzzers can target specific areas of code in a test program. And finally, domain-specific fuzzing includes research that did not look to improve fuzzing in general but instead sought to improve fuzzing for some specialized fields, such as IoT devices, for example.

The main contributions of these studies are highlighted in Tables 3–6 respectively.

3.1. Improved code coverage (general)

3.1.1. Angora

Chen and Chen (2018) created Angora, a mutation-based fuzzer that is capable of solving path constraints without symbolic execution. Path constraints refer to the set of conditions in the program inputs that are needed to get from one program state to another along some specified path. Symbolic execution meanwhile refers to a procedure to solve path constraints by using symbolic values

Table 3
The surveyed literature is classified according to the goal of the research. The applied techniques of each paper are also mentioned, along with the domain where the research applies and the main results. Better code coverage (general) refers to fuzzing research that attempted to increase the code coverage of fuzzing, but without a specific input format in mind.

Better code coverage (general)					
Main techniques	Use case	Programs compared Against	Tested programs/datasets	Main results	Paper
Markov chains	Grey-box fuzzing	AFL, Klee	N/A	9 new bugs	Böhme et al. (2017b)
New edge detection algorithm	Grey-box fuzzing	AFL, AFL-Fast	24 popular applications	+20% path coverage 260% more bugs	Gan et al. (2018)
Binary Instrumentation	Mutation-based fuzzing	Valgrind	28 popular applications, Lava-M	3x faster 80% more bugs	Dinesh et al. (2020)
context-sensitive branch coverage byte-level taint tracking gradient descent searching	Mutation-based fuzzer	Steelix	LAVA-M	103 new bugs in the LAVA-M dataset	Chen and Chen (2018)
Particle swarm optimization	Mutation-based fuzzers	AFL	13 open-source programs	+170% bugs +350% Crashes)	Lyu et al. (2019)
Static analysis Binary instrumentation	Mutation-based fuzzing	VUzzer, AFL-lafintel	LAVA-M	3x improvement	Li et al. (2017)
Program modification (Removes input checks)	Mutation-based fuzzing	Driller	DARPA CGC	45 new bugs 61 new bugs	Peng et al. (2018)
Human Code annotation	Open-Source Programs	AFL	N/A	Improved capabilities of tested fuzzers	Aschermann et al. (2020)

Table 4
The surveyed literature is classified according to the goal of the research. The applied techniques of each paper are also mentioned, along with the domain where the research applies and the main results. Better code coverage (complex input) refers to research that specifically focuses on increasing code coverage for programs that take complex input types.

Better code coverage (complex input)					
Main techniques	Use case	Programs compared Against	Tested programs/datasets	Main results	Paper
Improved seed generation, PCSG	Generation-based fuzzers	N/A	XSLT and XML engines (Sablotron, libslt, libxml2), Internet Explorer 11	+20% line coverage +15% function coverage 51 new bugs	Wang et al. (2017)
Machine learning (RNN)	Generation-based fuzzers	N/A	Edge PDF parser	Created PDF input with ... 97% acceptance rate Increased code coverage Found 1 new bug ... in Microsoft Edge	Godefroid et al. (2017)
Grammar aware mutations Abstract syntax trees	Mutation-based fuzzers	AFL	XML engine (libplist), JavaScript engines (WebKit, Jerryscript, ChakraCore)	+16.7% line coverage +8.8% function coverage 34 new bugs	Wang et al. (2019a)
Grammar inference	Coverage-guided fuzzer	Grammar Based Coverage Guided Fuzzers	Real World Programs	19 new bugs 11 new CVEs	Blazytko et al. (2019)

Table 5
The surveyed literature is classified according to the goal of the research. The applied techniques of each paper are also mentioned, along with the domain where the research applies and the main results. Better directed fuzzing refers to works that attempt to improve on directed fuzzing techniques.

Better directed fuzzing					
Main techniques	Use case	Programs compared against	Tested programs/datasets	Main results	Paper
Markov chains	Grey-box fuzzing	Katch, BugRedux, AFL	Katch and BugRedux benchmarks, LibPNG, Binutils, 7 security-critical libraries	up to 11 times faster than AFL	Böhme et al. (2017a)
Memory Consumption Guided Fuzzing	Grey-box fuzzing	AFL, AFLfast, PerfFuzz, FairFuzz, Angora, QSYM	14 programs (nm, nasm, openjpeg, bento4, libsass, etc.)	15 new CVEs	Wen et al. (2020)
Dynamic Taint Analysis	Mutation-based fuzzers	AFL, VUzzer, Angora, CollAFL, Honggfuzz, and QSYM	LAVA-M and 19 open source applications (readelf, libtiff, libwpd, libncurses, libsndfile, cflow, libcaca, libsass, etc.)	2.12X Program Paths,.3.09X Bugs 1.2X Program Paths, ... 1.52X Bugs	Gan et al. (2020)
Directed path exploration	White-box fuzzing	Hercules, Peach	13 applications (video players, document readers, music players, and image editors)	3 new bugs,.9/13 bugs found ... (unguided) 7 new bugs, ... 13/13 bugs found ... (guided)	Pham et al. (2016)
Compiler sanitizers	Grey-box fuzzing	Angora, AFLGo	15 programs (libssh, libxml2, boringssl, etc.)	+37% faster than Angora +288% faster than AFLGo	Österlund et al. (2021)

Table 6
The surveyed literature is classified according to the goal of the research. The applied techniques of each paper are also mentioned, along with the domain where the research applies and the main results. Impove fuzzing (for a specific domain) refers to work that aim to improve fuzzing in a specific context, such as operating system kernels, concurrency testing, or file systems.

Improve fuzzing (for a specific domain)					
Main techniques	Use case	Programs compared against	Tested programs/datasets	Main results	Paper
Randomized thread orderings Static analysis	Concurrency testing	N/A	6 real world programs	2 new bugs	Liu et al. (2018)
Static analysis Thread-aware instrumentation Reusing seeds for concurrency bugs	Concurrency testing	MAFL, AFL, MOPT	11 programs (lbzip2, ImageMagick, libvpx, x264, etc.)	8 new concurrency vulnerabilities 19 new concurrency bugs	Chen et al. (2020)
Protocol-guided fuzzing	IoT devices	N/A	17 real-world devices	8 new bugs	Chen et al. (2018b)
Seed image mutations Context-aware file operations	File systems	Syzkaller	8 file systems	90 new bugs	Xu et al. (2019)
Static analysis	Kernel drivers	N/A	7 Android phones	32 new bugs	Corina et al. (2017)
Hybrid fuzzing	Linux kernel	Moonshine, Syzkaller, kAFL	Linux Kernels	24 new vulnerabilities	Kim et al. (2020)
Seed recombination	SMT solvers	N/A	7 SMT solvers (CVC4, MathSAT5, STP, Yices2, etc.)	29 new bugs	Mansur et al. (2020)

for program inputs, rather than using specific program inputs. The benefit of using symbolic execution is that it can produce better quality input seeds, however, it also slows down the fuzzing process compared to randomly generated input seeds. Angora avoids symbolic execution and instead solves path constraints another way: Angora uses a combination of context-sensitive branch coverage, scalable byte-level taint tracking, gradient descent searching, input length exploration, and type and shape inference. These techniques allow for path constraints to be solved in an efficient manner. The main observation of their work is that generally, only a few bytes of the input have an impact on the solution to path constraints. This allows for only those few bytes to be mutated, reducing the size of the exploration space dramatically. The tracking of input bytes and their impact on program state is known as taint tracking. Various other works have also been found to utilize taint analysis (Rawat et al., 2017) or an approximation to taint analysis (Aschermann et al., 2019; Choi et al., 2019; Fioraldi et al., 2020).

3.1.2. AFLFast

Böhme et al. (2017b) created a coverage-based grey-box fuzzing tool, called AFLFast, that uses Markov chains. The motivation behind their technique is based on the fact that certain execution paths are much more commonly explored than others while fuzzing. Hence, their technique focuses more time and effort on fuzzing seeds that lead to those less frequently seen execution paths. This strategy leads to higher code coverage and improves the chances of finding deep vulnerabilities.

The use of Markov chains is what guides the creation of test cases/fuzz. Each state in the Markov chain corresponds to a seed that led to a new or unique execution path in the test program. The transition probability in the Markov chain between two states *i* and *j* represent the probability that fuzzing state *i's* seed will lead to a new seed that leads to another new and unique code path, represented by state *j*.

Initially, the seeds are manually passed to the model (mutational fuzzer). From there, each seed is fuzzed to create more seeds. New seeds are only kept and added to the Markov chain if they lead to new code paths while testing the program. This process is repeated for each seed and loops through each seed one by one until a predetermined time limit is reached. Each state in the Markov chain is also given an energy value, which determines the amount of fuzz to be created from that state's seed. The higher the energy, the more mutated seeds are created, and hence more time is spent on that state. The energy values are how the fuzzer is able to spend more time on less frequently seen states. Each time a state is fuzzed, the amount of energy is increased exponentially, and since each state is visited one by one, each state should have equal energy. However, the amount of energy is also divided by the number of times a given state has been seen, leading to frequently seen states being fuzzed less often than less frequently seen states.

3.1.3. CollAFL

Gan et al. (2018) created CollAFL, which is an extension to AFL that increases code coverage. One major problem with traditional grey-box techniques is inaccurate code coverage information. Inaccurate code coverage information leads to path collisions and to worse choices for seeds in the seed generation phase. A path collision is when a fuzzer confuses two different code execution paths of a test program as the same path. For example, AFL uses hash values to track different code paths. However, AFL stores these hashes in a 64KB array, which over time partially fills with hashes, which then results in some path collisions. CollAFL addresses this problem by using an improved edge detection algorithm, while still maintaining the low instrumentation overhead of AFL. In addition to this, it also uses the improved code coverage information to better guide seed generation towards unexplored paths in the test program. These improvements led to 20% more program paths being covered and 260% more bugs being found in a test program when compared to regular AFL after 200 h of testing.

3.1.4. MOPT

In the work of Lyu et al. (2019), the authors observed that different fuzzers select predefined mutation operators to mutate seed test cases. The purpose of mutation operators is to characterize where (which bytes to mutate) and how to mutate (e.g. replacing the bytes, deletion, etc.). As different mutation operators have varying efficiencies, mutation schedulers which follow a predefined distribution of mutation operators are inefficient. To address this issue, the authors have developed a new mutation-based fuzzing scheme, namely MOPT. MOPT selects the next optimal mutation operator using a particle Swarm Optimization (PSO) algorithm (Eberhart and Kennedy, 1995). The algorithm is designed to work with several off-the-shelf fuzzers, such as AFL, AFLFast, and VUzzer. The proposed scheme was evaluated on 13 real-world applications where it outperformed existing fuzzers such as AFL, AFLFast, and VUzzer. On average, MOPT outperformed standard AFL algorithms by finding 170% more in vulnerability searching and found 350% more crashes.

3.1.5. Steelix

According to the authors (Li et al., 2017), coverage-based fuzzing is an effective technique for achieving good code coverage, however, it suffers from the inability to penetrate deep into an application. Further, coverage-based fuzzing is not efficient when it comes to executing down code paths protected by magic bytes. Magic bytes refer to the bytes used in string comparisons (e.g., string equality comparisons). To break down magic bytes comparison, several fuzzing techniques have used heavy-weight program analysis. However, heavy-weight program analysis is expensive and not easily scalable. To circumvent this issue, Steelix was developed. Steelix is a program-state-based binary fuzzing approach proposed by Li et al. (2017). Steelix aims to improve upon coverage-based and mutation fuzzing technologies. Similar to MOPT, Steelix extends the abilities of AFL, which can be seen when Steelix is evaluated over two well-known datasets, LAVA-M, DARPA CGC, and 5 other binaries. Steelix consists of three main components i.e., static analysis, binary instrumentation, and a fuzzing loop.

In static analysis, a given binary application is broken down into its source code (or assembly). Steelix tracks all comparisons and tests, maintaining a list of the instruction address and type (memory address, register, immediate value). In a second list, it tracks strcmp, strncmp and memcmp, tracking the function call and name. This information is passed onto the binary instrumentation. Binary instrumentation generates comparison progress information during runtime by instrumenting the program under test and from that it obtains the actual value of comparison operands. The final step is the fuzzing loop. The fuzzing loop takes the instrumented binary and executes it. Using the coverage, comparison, progress, and location information, it mutates the seed and then inputs the new seed back in as test input. This process of taking the output of an execution cycle repeats until an exit condition is met and executed.

3.1.6. T-Fuzz

T-Fuzz, proposed by Peng et al. (2018), is similar to Steelix and focuses on deeply embedded vulnerabilities that are hidden the logical comparisons within conditional checks. T-Fuzz consists of three main components i.e., a fuzzer, a program transformer, and a crasher. It uses off-the-shelf fuzzers, such as AFL, to fuzz the program.

An off-the-shelf fuzzer will first run its analysis. After it is no longer able to find vulnerabilities, T-Fuzz moves in and uses a dynamic, lightweight, trace-based application that is used to discover logical comparisons that the original fuzzer failed to satisfy. These logical comparisons are classified into two groupings; either non-critical sanity checks (NCCs) or critical sanity checks (CCs). Sanity checks simply refer to checks that a program performs on input. Both NCCs and CCs are logical checks on input, where NCCs are defined such that they can be removed without introducing new bugs, while this cannot be said of CCs.

The program analyser traces the program under test and filters out any NCCs candidates; this is the transformer component of T-Fuzz. This is done since it allows the fuzzer to proceed further into the program without introducing new bugs by doing so. Any crashed input found during the transformer phase is analysed by the crash analyser which uses a symbolic-execution technique to filter false positives. T-Fuzz, therefore, excels in two fields. The use of lightweight tracing, combined with the limited application of heavyweight analysis means that T-fuzz runs quickly. Additionally, the transformation step allows the fuzzer to bypass hard-coded conditional checks that otherwise would be difficult to bypass, allowing the fuzzer to peer deep into applications.

T-Fuzz found vulnerabilities in 166 of the 296 tested binaries in the DARPA Cyber Grand Challenge (DARPA CGC) dataset. Specifically, T-Fuzz outperformed Driller and AFL by 45 and 61 binaries respectively. T-Fuzz would also go on to outperform Steelix and VUzzer in the LAVA-M Dataset where hard conditional checks were present.

3.1.7. SLF

You et al. (2019) proposed a novel fuzzing technique in 2019. The proposed novel solution features the ability to generate seed inputs. SLF adds to AFL by further classifying inputs checks and input fields. SLF further classifies according to relations, such as arithmetic, object offset, and data structure length. Fundamentally, they developed a multi-goal algorithm to apply class-specific mutations to a seed. SLF was tested against 20 existing benchmark programs, Google Fuzzing Suite. It was tested against AFL, AFLFast, KLEE (Sym. Execution), S2E (Sym. Execution), and the hybrid tool, Driller. In almost all cases, SLF significantly outperforms all other testing suites and applications.

3.2. Improved code coverage (complex input handling)

3.2.1. SkyFire

Wang et al. (2017) developed a generation-based method called SkyFire to fuzz programs that take highly structured files as input. Such input normally gets processed in stages, which include, syntax parsing, semantic checks (i.e., does the input make sense?), and application execution. To effectively and fully fuzz such a program, the fuzz needs to pass the first two stages, but this is difficult for complex input types. For example, standard mutational-based fuzzing will often fail at the syntax parsing stage. Meanwhile, generation-based fuzzing approaches often fail at semantic checks. Therefore, conventional approaches struggle to fuzz the application execution stage, potentially leaving vulnerabilities undiscovered.

SkyFire gets around these issues by using probabilistic context-sensitive grammar (PCSG) to generate good input seeds. Generation-based fuzzers often use context-free grammars. These can generate valid syntax rules, but fail to account for semantic rules. A PCSG is different from context-free grammars in that it accounts for both syntax rules and semantic rules, where it uses a probability to determine if different production rules apply, given a particular context. The PCSG is automatically created from a passed collection of input files, such as XML files, for example. Well-distributed seeds are then generated by repetitively applying the PCSG's production rules. SkyFire was tested in conjunction with AFL against several XSLT and XML engines. From this, there was a 20% increase in line coverage and a 15% increase in function coverage, on average. The JavaScript and rendering engines of Internet Explorer 11 were also tested, and from this SkyFire discovered 19 new memory corruption bugs and 32 new DoS-related bugs.

3.2.2. Grimoire

Blazytko et al. (2019) created a coverage-guided fuzzer named Grimoire that uses grammar inference. Grammar inference is the automatic determination of the input structure (i.e., the grammar) that a program accepts as valid input. Normally, expert domain knowledge is needed to successfully create a grammar for a program. The advantage of Grimoire is that this expert knowledge is not needed, but it still reaps the benefits of having a grammar specific to the program under test. To infer grammar automatically, some previous works (Bastani et al., 2017) have used techniques such as converting known valid inputs into a regular expression and generalizing new inputs from there. Other approaches have used taint tracking and symbolic execution to infer grammars (Gopinath et al., 2018; Hoschele and Zeller, 2017). However, such approaches have been shown to have drawbacks. Grimoire developed a new approach that uses standard coverage-guided fuzzing with an additional step in the fuzzing progress to infer grammar.

This additional step takes note of input seeds whenever they increase code coverage. Then, variations of this input seed that can be made without changing the coverage are referred to as generalizations. These generations form the basis of the inferred grammar. Gramioire found 19 new memory corruption bugs in real-world applications and 11 new CVE's.

3.2.3. Machine learning

The application of machine learning to fuzzing is a fairly new area of research, but Godefroid et al. (Godefroid et al., 2017) created Learn&Fuzz, an applied neural-network-based statistical machine-learning techniques to automate the generation of a complex input format (PDF) from input samples. This was utilized to test the PDF parser embedded in Microsoft's Edge browser. Learn&Fuzz utilized a recurrent neural network-based character-level language model (char-RNN) to learn a generative model of sequences to generate PDF objects from a sample of 63,000 PDF objects from 534 well-formed PDF files.

During their testing of three different sampling strategies (NoSample, Sample, and SampleSpace), they observed an opposition between the coverage and the pass rates. A pure learning algorithm with an almost perfect pass rate, like SampleSpace, will almost always generate well-formed objects. This results in poorer coverage of the error-handling code of the program. A noisier learning algorithm, like Sample, with a lower pass-rate, can generate quite a few well-formed objects, but can also generate some additional ill-formed ones. This will result in better coverage of the program's error-handling code. This generalizes to a conflict between the learning and fuzzing goals of the program: learning will ultimately want to capture the structure of well-formed inputs, while fuzzing will want to do the opposite and break that structure to explore unanticipated code paths and find more bugs. Ultimately, they didn't have enough data to conclude whether any particular approach is best outside of the opposition observed.

3.2.4. Superion

Fuzzers like American Fuzzy Lop (AFL) currently have problems when it is used to process structured inputs like XML and JavaScript since the trimming and mutation strategies used are grammar-blind. Wang et al. (2019a) created an extension to AFL called Superion. Superion utilizes grammar-aware trimming and mutation strategies to augment AFL so that it can more effectively process structured inputs. The grammar-aware trimming strategy implemented utilizes abstract syntax trees (ASTs) to trim the test inputs at the tree level. Two mutation strategies are implemented for Superion: a tree-based mutation, which replaces subtrees with ASTs of the parsed test inputs, and an enhanced dictionary-based mutation.

Superion was evaluated using XML engine libplist and three JavaScript engines (WebKit, Jerryscrupt, and ChakraCore). The results showed improved code coverage, with 16.7% inline and 8.8% function coverage, and bug-finding compared to just AFL or jsfunfuzz. 34 new bugs were ultimately discovered, 22 were previously undiscovered vulnerabilities with 19 CVEs generated and 3.2k USD bug bounty awards collected for them.

3.3. Improved directed code coverage

3.3.1. Directed grey-box fuzzer

Böhme et al. (2017a) developed a directed grey-box fuzzer. Previously, grey-box fuzzers were only coverage-based as they lacked the internal information needed to effectively guide the fuzz towards a specific goal. However, using a similar approach to Böhme et al. (2017b), this is achieved by fuzzing seeds that are closer to the target more often. Their idea was implemented through a modified AFL, which they called AFLGo. AFLGo consists of four separate components which are the Graph Extractor (GE), Distance Calculator (DC), instrumentor, and a fuzzer. The GE creates both a call graph and the related control-flow graphs based on function signatures and source file information. The DC uses the call graph to determine the distance between basic blocks in the code. The instrumentor uses the basic blocks of the code and injects assembly code to keep track of the program execution state. Finally, the fuzzer, which is AFLFast, fuzzes the instrumented binary file. AFLGo was found to outperform both directed symbolic execution-based white-box fuzzing and undirected grey-box fuzzing.

3.3.2. MoWF directed white-box fuzzing

Traditional white-box fuzzing is not perfect. It suffers from two challenges, namely path explosion, and seed dependence. Path explosion occurs when a fuzzer gets bogged down in an exponential number of paths that exercise invalid inputs. On top of this, white-box fuzzing relies on seed generation. The seed used by white-box testing is assumed to be of valid input. Often this isn't the case for any number of reasons, notably data chunks can either be missing or in the incorrect order. Pham et al. (2016) proposed a solution to fix this.

Model-based White-box Fuzzing (or MoWF for short) proposes a marriage of white-box and black-box fuzzing. The black-box model searching aspect of MoWF allows it to efficiently search the space of valid inputs. The white-box approach allows MoWF to cover white-box code. Using both the black-box information, and white-box information, MoWF is capable of generating its own sample input given some information such as crash reports, sample data, and a file model for the data. One interesting part of MoWF is that on top of not needing a seed, it can use data from other sample files.

One of the examples given by Pham et al. (2016) is a VLC binary that has a bug in it. Taking in a crash report, a set of existing PNG files, and the PNG model, MoWF will first leverage a black-box method. This input will allow MoWF to generate sample data near a given crash location, as detailed in the crash report, and the data from PNG images will allow it to explore any related paths. The sample data (PNGs) are broken into segments and added to a pool, where it can be used as a donor. On top of this, a user will mark certain fields of a data structure as modifiable, or "symbolic". This allows the application to selectively fuzz one or more fields for a given file type. As an example, PNG images may allow the height and width of an application to be modified, but not a CRC32 checksum.

In the second phase of MoWF, it will attempt to explore paths. This is accomplished by manipulating data chunks, adding, moving, or removing, for a given input file. Branches are marked as crucial given the behavior of the input file based on an enumerable data field. Once the path has been explored, MoWF will use taint analysis to identify the bytes responsible for the branch execution. If the data type is enumerable, and if it's not executed in both directions, then it's considered crucial. MoWF will then add or remove data chunks, using the donor files supplied.

In the third step, selective symbolic execution allows for local search space of semi-valid input, starting from a negated branch. Potentially invalid files are fixed in the file repair stage. Once a target location is reached, the traditional whitebox fuzzer will produce crashing input, if the conjunction is satisfiable. If it is not, the unsatisfiable core guides path exploration and checks again.

The final step is repeating. MoWF will use generated files as new seeds to continue the next iteration of fuzzing, starting back at step one.

MoWF's results seem promising. With no given seed, relying on generated seed values from the application, MoWF was able to identify 9 vulnerabilities. With more guided input, MoWF exposed all 13 vulnerabilities.

3.4. Improved domain-specific fuzzing

3.4.1. Concurrency fuzzing

Liu et al. (2018) created two concurrency vulnerability discovery fuzzer tools for C programs. These tools work by randomizing the order of operations taken by various threads and can detect concurrency buffer overflow, double free, and use-after-free vulnerabilities. One of the fuzzers is called the interleaving exploring fuzzer and the other is called the vulnerability detection fuzzer. Both tools expand AFL by allowing fuzz to include not only random input but also random thread interleavings.

The interleaving exploring fuzzer's job is to randomize the order of thread operations. This is achieved by randomly modifying thread priorities and by causing threads to sleep for random amounts of time. This helps to identify issues caused by particular thread interleavings. Six different multi-threaded C programs were tested, which included boundedbuff (a producer-consumer module), swarm (a parallel programming framework for multicore processors), bzip2smp (a multi-threaded version of the ZIP compression program: BZIP2), pfscan (a file scanner), ctrace (a tracing and debugging library), and qsort (a quick sort implementation). The interleaving exploring fuzzer produced three new crashes that the standard AFL fuzzer did not.

The vulnerability detection fuzzer works in conjunction with a static analysis tool and uses the static analysis to determine thread priorities that stand a good chance of discovering vulnerabilities. The static analysis tool works by searching for areas of code where concurrency issues may be an issue. It then uses this information to help guide the fuzzer to target those areas with the correct thread priorities set to trigger the suspected issue. The static analysis tool was created by manually looking for patterns common to concurrency vulnerabilities in C and writing a tool to automate the search. The vulnerability detection fuzzer found two new vulnerabilities that the standard AFL fuzzer missed.

The main limitations of the study are that the tools are only for C programs using POSIX multi-thread functions and that the static analysis system does not scale well to programs larger than around ten thousand lines long.

3.4.2. IoT Fuzzing

IoT devices are becoming ubiquitous, but the testing of IoT devices for security vulnerabilities is frequently solely restricted to the binary analysis of the device's firmware. This limitation makes it difficult for security researchers to find critical bugs and flaws before attackers can exploit them. Chen et al. (2018b) created a unique automatic fuzzing framework called IoTFuzzer. IoTFuzzer is utilized to find memory corruption vulnerabilities in IoT devices and does so without access to their firmware images. This is accomplished by leveraging the official app which is used in conjunction with the device. These apps contain significant information on the protocol used to communicate with IoT device's firmware. IoTFuzzer will automatically analyze traffic from the IoT app and seek to identify protocol fields that are then mutated by the fuzzer. The IoT device is then monitored remotely and detects any crashes that might have been instigated by the mutated protocol field.

IoTFuzzer was ultimately evaluated on 17 IoT devices and found 15 different memory corruption vulnerabilities, 8 of which were previously unknown.

3.4.3. File system fuzzing

Fuzzing can be a good fit for testing file systems. However, three challenges exist for this application: mutating a large performance degrading image blob, generating image-dependent file operations, and reproducing found bugs. Xu et al. (2019) presented JANUS, a feedback-driven fuzzer that explores the two-dimensional input space of a file system. The fuzzer addresses the first two challenges by functioning on a two-dimensional input space which will mutate the metadata of a large image while generating image-directed file operations. JANUS also addresses the final challenge by relying on a library OS rather than on VMs for fuzzing. Since JANUS can load a fresh copy of the OS with minimal overhead, this aids with the reproducibility of bugs. JANUS was implemented as a variant of AFL.

Xu et al. (2019) evaluated JANUS on eight different file systems (ext4, XFS, Btrfs, F2FS, GFS2, HFS+, ReiserFS, and VFAT) and found 90 bugs in the upstream Linux kernel, 62 of which have been confirmed as previously unknown. Of the found bugs, 43 of the bugs were fixed, with a total of 32 CVEs generated for them. Compared to Syzkaller, JANUS achieves a higher code coverage after fuzzing for 12 h. JANUS ultimately visits 4.19x more code paths for Btrfs and 2.01x for ext4 compared to Syzkaller. Due to the use of a library OS, JANUS can reproduce 88–100% of the crashes it encounters versus Syzkaller, which is unable to reproduce any.

3.4.4. DIFUZE: kernel fuzzing

The complex input data structures of device drivers pose some challenges for fuzzing. This has led to a dearth of research within the sphere of kernel driver security. Corina et al. (2017) created DIFUZE, an automated ioctl interface-aware fuzzing tool that retrieves the interface for the device driver, generates valid inputs, and triggers the execution of Linux kernel drivers. When crashes are detected within the kernel drivers, the inputs that lead to these crashes are logged so that they can be analyzed to locate bugs within the program. DIFUZE is ultimately used to detect bugs within the device drivers of the Android operating system.

DIFUZE was evaluated on 7 different Android smartphones. The results gathered showed that DIFUZE can effectively identify kernel driver bugs and found 36 bugs, 32 of which were previously unknown vulnerabilities.

3.4.5. HFL

Kim et al. (2020) also looked into the problem of fuzzing a kernel. They developed the HFL fuzzer with the aim of fuzzing the Linux kernel. In order to do this effectively, hybrid fuzzing was used which combines traditional fuzzing with symbolic execution, concolic execution, or taint tracking (Blazytko et al., 2019). Hybrid fuzzing attempts to minimize the problems associated with traditional fuzzing and symbolic execution. HFL is not the only fuzzer to make use of hybrid fuzzing (Borzacchiello et al., 2021; Pak, 2012; Yun et al., 2018; Zhao et al., 2019), but it is the first to apply the technique to kernel fuzzing. They noted some domain-specific problems with fuzzing a kernel and developed various solutions to address those. For example, the effect of a particular system call often depends on the current system state, meaning it is difficult to determine the sequence of system calls needed to trigger a vulnerability. To solve this problem, HFL determines the correct system call sequences by using static pointer analysis. Static pointer analysis is a method to map pointers to variables and or memory locations. HFL was tested and it found 24 new vulnerabilities in the Linux kernel.

3.4.6. Blackbox mutational fuzzing for SMT solvers

SMT solvers are a significant component within software verification, systematic test generation, and program synthesis. These programs are frequently complex, and because of this, they can contain critical bugs that influence the correctness of the results. Mansur et al. (2020) created STORM, an original black-box mutational fuzzing approach used for revealing critical bugs in SMT solvers. A critical bug in this context is defined as when the solver returns unsatisfiable for satisfiable instances. The importance of this is because, in the case of a software verifier or a test case generator, this failure will result in the inability to uncover significant bugs.

This new fuzzing technique has three steps: seed fragmentation, formula generation, and instance generation. Seed fragmentation takes a seed instance as input and breaks down the formulas in the instance into sub-formulas. This in turn is used in the second phase when these sub-formulas are recombined to generate new formulas. For the final stage, the generated formulas from step two are used to create a new, satisfiable SMT instance. Since the generated instance is satisfiable, there is no reason to run multiple solvers for differential testing. This gives STORM an advantage compared to existing solvers like FuzzSMT or StringFuzz.

STORM was evaluated on seven tried and true solvers that supported the SMT-LIB input format (Boolector, CVC4, MathSAT5, SMTInterpol, STP, Yices2, and Z3). This resulted in STORM finding 29 previously unknown critical bugs within these solvers. STORM was also used to test new features of solvers: including the new bit vector theory in Yices2's MCSAT solver, a new arithmetic solver from the developers of Z3, and Z3's debug branch. STORM-generated string instances were also provided to help test Z3str3 from the same developers of StringFuzz. The feedback they received for STORM from the solver developers was quite positive.

4. Discussion

With software projects becoming significantly more complex in recent years, the techniques that software and QA engineers use to uncover vulnerabilities must also evolve to meet that complexity. The evolution of the software testing methodology is an ongoing project, with creative methods to reverse engineer, debug, and test new software systems.

The works highlighted in this paper show massive improvement over their traditional fuzzer predecessors. Existing applications such as AFL have been proven to be highly adaptable to new forms and algorithms to further fuzz-based testing. For example, works such as SkyFire, CollAFL, AFLFast, Superion, JANUS, and many others, have all built upon AFL, expanding its capabilities in many different directions to meet noted limitations.

The radical improvement in fuzzing potential will assist in making the operating systems and applications we use significantly more secure. It is, however, important to preface that by pointing how that none but the simplest of systems will ever be completely free of vulnerabilities; meaning fuzzers must continue to evolve to find more and more vulnerabilities that previous iterations of fuzzers failed to capture.

From the literature survey, it was found that research regarding grey-box fuzzers was very common. This is likely due to grey-box fuzzing, when done right, can have the benefits of both black-box and white-box fuzzing, but with a much-reduced overhead when compared to white-box fuzzing. Research into black-box fuzzing was nowhere to be seen, with research in that area dying out in recent years. White-box fuzzing was found to be in the middle-ground, being a somewhat popular research area still.

In terms of research goals, better code coverage for complex inputs seems to be the most popular now. Being able to fuzz programs that take complex inputs has proven to be a challenging problem due to the problem of fuzzed input being rejected before it can reach deep into the program. However, the ability to fuzz complex inputs will open the door to detecting bugs in a much wider array of test programs. Machine learning has become especially popular for this task within the past year.

5. Research challenges

5.1. Deep bug coverage

An important issue for fuzzers is trying to find new code execution paths, as otherwise time is being wasted on already discovered paths. Some work has been done on this, such as the work done by Böhme et al. (2017b); Gan et al. (2018). However, fuzzers still tend to struggle with complex input formats as this tends to lead to large numbers of input being rejected before they can reach deep parts of the program. Also, as program size increases, so does the number of paths, which is called path explosion (Pham et al., 2016). This all makes it very difficult for fuzzers to find the precise format of fuzz needed to reach deep parts of the code and reach high levels of code coverage.

5.2. Limited bug variety

As mentioned in Böhme et al. (2020), a good amount of current fuzzers focus on simple indicators for determining if a bug has occurred or not. Perhaps the most common approach to this is using program crashes as an indicator that a bug has occurred. Not all bugs show up as program crashes however, some bugs are "silent" in that it is not readily apparent that anything has gone wrong. For example, for embedded devices, crashes are not common behavior, even if memory corruption has occurred (Muench et al., 2018). Hence, developing more sophisticated methods to detect security-relevant bugs is needed.

5.3. Computing resources

Another important step for researchers is creating algorithms that are both effective at finding bugs and efficient with respect to time and computing resources. The computing resources of a given environment are finite. Once the computing resource limits have been reached, it can cause false positives or false negatives as the application begins to discard imperative states, objects, and data. Therefore, it's critical that any designed algorithm must maintain or marginally with respect to modern systems, increase compute resource usage. Some works, like ClusterFuzz (Aizatsky, 2016), have helped solve resource limitations by providing an environment for certain fuzzers to operate in a distributed manner. Other works (Pham et al., 2016) have addressed computing limitations by adopting solutions that minimize typical performance bottlenecks, like path explosion, to improve fuzzing speeds.

5.4. Initial input seeds

Another challenge for researchers in some cases was obtaining large numbers of valid inputs for the test program. For example, for complex input types, it may be difficult to obtain a well-represented sample set that contains many different valid input cases. Without a good set of initial input seeds, mutation-based fuzzers may be less successful, or at least may take longer to find particular vulnerabilities.

The choice of the set of non-crashing seed inputs referred to as a corpus is extremely important to the overall performance of the fuzzer. Herrera et al. (2021) examined the impact different corpus selections made and found that a minimized corpus is always better due to the faster iteration rate. They also recommend that an

additional campaign with the empty seed be conducted to quickly weed out shallow bugs, except when conducting industrial-scale fuzzing campaigns where the empty seed should never be used.

5.5. Difficulty comparing fuzzers

Klees et al. (2018) conducted a survey of 32 papers and found many do not follow a consistent methodology that can lead to misleading or weakened conclusions. Among the problems found was that many papers did not perform multiple runs and account for varying performance by using a statistical test, fuzzer performance was not counted using distinct bugs, but unique crashes instead which can over-count the number of bugs, the usage of short timeouts which can be problematic since performance can vary during a run, the lack of consideration of seed choices on algorithm performance, and the wild variety of choice of target problems. All of these difficulties make comparing different fuzzers difficult. Similarly, Hazimeh et al. (2020) attempted to address this problem by constructing Magma: a ground-truth fuzzing benchmark that would enable uniform fuzzer evaluation and comparisons. An advance of note is the differentiation of bugs reached, triggered, and detected which would allow different fuzzers to be classified and compared across multiple dimensions.

5.6. Limited knowledge

One final challenge that we've identified for fuzzing research is that it is often difficult to know how many vulnerabilities persist in software after fuzzing is performed. Because of this, it is hard to know how long to run experiments for. Should experiments continue for a day, a week, or longer? This makes it more difficult for different fuzzers to be compared, as they may find vulnerabilities at different rates and their rate of vulnerability discovery may decrease at different rates compared to other fuzzer programs.

6. Future research directions

6.1. Seed generation for complex input

As mentioned in the previous section, seed generation for complex input types is a difficult problem to deal with. In recent years, some progress has been made, like for example, SkyFire (Wang et al., 2017), which tries to deal with this problem by automatically generating a grammar from valid input files. This grammar has some understanding of both syntax and semantic rules. However, this could be furthered by applying more powerful techniques, such as machine learning. Machine learning algorithms could be passed large numbers of valid input as training data, create fuzz for a fuzzer to test, and update itself accordingly based on how well the fuzzer performed (perhaps with deep neural networks). Some research has looked into this already, but more work should be focused on this area as challenges still exist. For example, (Wang et al., 2020b) describes how machine learning has been applied to improve generation-based fuzzing for both programs accepting PDFs and HTML blocks as input. However, in terms of a machine learning model capable of working with arbitrary complex file types, there is no good solution currently available, with the best solutions for this only being able to extract basic syntax rules (Wang et al., 2020b). Other works, such as Grimoire (Blazytko et al., 2019), have found success with automatically inferring grammar. However, this area of research can still be expanded and improved upon.

6.2. Automatic remediation suggestions

Once a fuzzer discovers vulnerabilities in software, development time is needed to first understand the cause of the issue and then additional time is needed to actually fix the issue. It would therefore be helpful to have fuzzers give insight into the problem (maybe citing common coding oversights or errors, if relevant) and offer potential solutions that do not impact the proper functioning of the program.

6.3. Expand bug variety

Most fuzzers are unable to detect more subtle types of bugs that do not readily appear as a program crashes (Böhme et al., 2020). As such, more research should be done on how to detect a wider variety of bugs. For example, privilege escalation and remote code execution vulnerabilities would be excellent and very relevant bugs to be able to directly test for using fuzzing, from a security perspective.

6.4. Standardization of fuzzing terms

While researching the various classifications of fuzzers, it was found that some sources define things slightly differently. For example, in Li et al. (2018) smart fuzzers are defined as "there is a feedback between the monitoring of program execution state and test-case generation" [sic], while in Gorbunov and Rosenbloom (2010), they are defined as "[has] knowledge of the [input] implemented by its targets". In addition to that, certain classifications do not have clear definitions, such as for grey-box fuzzers. It could therefore be useful for there to be a standard set of definitions to refer to.

6.5. Further expansion into IoT

While Chen et al. (2018b) did create a framework for finding memory corruption vulnerabilities in IoT devices requiring firmware images. Their research had some limitations with respect to scope, communication protocol, and the absence of a cloud relay that is utilized by some apps. More research to expand this up-and-coming area and address these weak points is a focus for further research.

6.6. User-provided grammars

A limitation encountered by Wang et al. (2019a) is the necessity to provide grammars for their Grammar-aware fuzzer Superion. Because of this, Superion was limited in its ability to only cover publicly documented formats. Undocumented and proprietary grammars could not be used due to this limitation. They did note, however, that an area to focus on for future work would be to integrate an automatic grammar inference technique to overcome this limitation.

6.7. Generalized fuzzers

One thing that was noted while researching this paper is that there are a number of papers specifically focused on expanding fuzzing to new domains. For example, papers were found regarding the areas of file systems, kernels, and concurrency testing. Perhaps one area of future research would be to look into how fuzzers can be quickly and easily adapted to a wider variety of test programs and applications, all without major overhauls.

7. Conclusion

In conclusion, researchers are making headway into improving fuzzing techniques across multiple fuzzing techniques and domains. While current improvements are notable, more efforts are needed to secure modern software systems.

From the inception of fuzzing in the early 1990s, we can see the improvements in fuzzing techniques. More importantly, though, we can see the improvement of techniques and algorithms compared to modern methods. These improvements may be viewed under our classification tables where each research article is broken down into their goal, techniques used, any applicable use cases, and the results of the research team's improvements. Depending on the goal of the research article, they are classified within the same category. This allows for a quick comparison of similar research projects, with differentiating techniques, use cases, and results. This would allow researchers looking to implement a new algorithm to quickly compare and contrast which one option would best suit their requirements.

There are significant challenges in developing effective fuzzing techniques. Researchers must maintain a relatively low time and resource cost, while also improving the reliability and effectiveness of the fuzzing techniques. The problem of efficiency and efficacy is discussed in these papers. We must utilize algorithms that promise better results while minimizing the cost or resource intensity. This is important in reducing the amount of time required to manually find vulnerabilities in an application.

The AFL fuzzer in particular is quite promising. Many researchers are extending AFL's capabilities with augmented algorithms and fuzzing techniques. This is leading to marketable increases in the quantity and quality of software bugs found, with marginal increases in time, resources, and cost.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Aizatsky, M., 2016. Announcing OSS-Fuzz: continuous fuzzing for open source software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

Aschermann, C., Schumilo, S., Abbasi, A., Holz, T., 2020. Ijon: exploring deep state spaces via fuzzing. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1597–1612.

Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T., 2019. Redqueen: fuzzing with input-to-state correspondence. In: NDSS, vol. 19, pp. 1–15.

Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., Vigna, G., 2006. Snooze: toward a stateful network protocol fuzzer. In: International Conference on Information Security. Springer, pp. 343–358.

Bastani, O., Sharma, R., Aiken, A., Liang, P., 2017. Synthesizing program input grammars. ACM SIGPLAN Notices 52 (6), 95–110.

Blazytko, T., Aschermann, C., Schlögel, M., Abbasi, A., Schumilo, S., Wörner, S., Holz, T., 2019. GRIMOIRE: synthesizing structure while fuzzing. In: 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA, pp. 1985–2002. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V., 2018. Stringfuzz: a fuzzer for string solvers. In: International Conference on Computer Aided Verification. Springer, pp. 45–51.

Boehme, M., Cadar, C., Roychoudhury, A., 2021. Fuzzing: challenges and reflections. IEEE Softw. 38 (3), 79–86.

Böhme, M., Cadar, C., Roychoudhury, A., 2020. Fuzzing: challenges and reflections. IEEE Softw..

Böhme, M., Pham, V.-T., Nguyen, M.-D., Roychoudhury, A., 2017. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2329–2344.

Böhme, M., Pham, V.-T., Roychoudhury, A., 2017. Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Softw. Eng. 45 (5), 489–506.

Borzacchiello, L., Coppa, E., Demetrescu, C., 2021. Fuzzolic: mixing fuzzing and concolic execution. Comput. Secur. 108, 102368.

Cadar, C., Sen, K., 2013. Symbolic execution for software testing: three decades later. Commun. ACM 56 (2), 82–90.

Chen, H., Guo, S., Xue, Y., Sui, Y., Zhang, C., Li, Y., Wang, H., Liu, Y., 2020. {MUZZ}: thread-aware grey-box fuzzing for effective bug hunting in multi-threaded programs. In: 29th {USENIX} Security Symposium ({USENIX} Security 20), pp. 2325–2342.

Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., Liu, Y., 2018. Hawkeye: towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2095–2108.

Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W.C., Sun, M., Yang, R., Zhang, K., 2018. Iotfuzzer: discovering memory corruptions in IoTthrough app-based fuzzing. NDSS.

Chen, P., Chen, H., 2018. Angora: efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 711–725.

Choi, J., Jang, J., Han, C., Cha, S.K., 2019. Grey-box concolic testing on binary code. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 736–747.

Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., Vigna, G., 2017. Difuze: Interface aware fuzzing for kernel drivers. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138.

Dinesh, S., Burow, N., Xu, D., Payer, M., 2020. Retrowrite: statically instrumenting cots binaries for fuzzing and sanitization. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1497–1511.

Eberhart, R., Kennedy, J., 1995. A new optimizer using particle swarm theory. In: MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science. IEEE, pp. 39–43.

Eceiza, M., Flores, J.L., Iturbe, M., 2021. Fuzzing the internet of things: a review on the techniques and challenges for efficient vulnerability discovery in embedded systems. IEEE Internet Things J. 8 (13), 10390–10411.

Fioraldi, A., D'Elia, D.C., Coppa, E., 2020. Weizz: automatic grey-box fuzzing for structured binary formats. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1–13.

Gan, S., Zhang, C., Chen, P., Zhao, B., Qin, X., Wu, D., Chen, Z., 2020. {GREYONE}: data flow sensitive fuzzing. In: 29th {USENIX} Security Symposium ({USENIX} Security 20), pp. 2577–2594.

Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z., 2018. Collafl: path sensitive fuzzing. In: IEEE Symposium on Security and Privacy (SP), pp. 679–696.

Godefroid, P., Peleg, H., Singh, R., 2017. Learn&fuzz: Machine learning for input fuzzing. arXiv:1701.07232

Gopinath, R., Mathis, B., Höschele, M., Kampmann, A., Zeller, A., 2018. Sample-free learning of input grammars for comprehensive software fuzzing. arXiv preprint arXiv:1810.08289

Gorbunov, S., Rosenbloom, A., 2010. Autofuzz: automated network protocol fuzzing framework. IJCSNS 10 (8), 239.

Hazimeh, A., Herrera, A., Payer, M., 2020. Magma: a ground-truth fuzzing benchmark. Proc. ACM Meas. Anal. Comput. Syst. 4 (3), 1–29.

Herrera, A., Gunadi, H., Magrath, S., Norrish, M., Payer, M., Hosking, A.L., 2021. Seed selection for successful fuzzing. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. Association for Computing Machinery, New York, NY, USA, pp. 230–243.

Hoschele, M., Zeller, A., 2017. Mining input grammars with autogram. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE, pp. 31–34.

Hsu, C.-C., Wu, C.-Y., Hsiao, H.-C., Huang, S.-K., 2018. Instrim: lightweight instrumentation for coverage-guided fuzzing. Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research.

Hu, J., Chen, J., Ali, S., Liu, B., Chen, J., Zhang, C., Yang, J., 2021. A detection approach for vulnerability exploiter based on the features of the exploiter. Secur. Commun. Netw. 2021, 1–14.

Hydara, I., et al., 2021. The limitations of cross-site scripting vulnerabilities detection and removal techniques. Turkish J. Comput. Math. Educ. (TURCOMAT) 12 (3), 1975–1980.

Jayaraman, K., Harvison, D., Ganesh, V., Kiezun, A., 2009. jFuzz: a concolic whitebox fuzzer for java. In: NASA Formal Methods, pp. 121–125.

Kim, K., Jeong, D.R., Kim, C.H., Jang, Y., Shin, I., Lee, B., 2020. Hfl: hybrid fuzzing on the linux kernel. NDSS.

Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M., 2018. Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138.

Li, J., Zhao, B., Zhang, C., 2018. Fuzzing: a survey. Cybersecurity 1 (1), 1–13.

Li, Y., Chen, B., Chandramohan, M., Lin, S.-W., Liu, Y., Tiu, A., 2017. Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 627–637.

Lin, G., Wen, S., Han, Q.-L., Zhang, J., Xiang, Y., 2020. Software vulnerability detection using deep neural networks: a survey. Proc. IEEE 108 (10), 1825–1848.

Liu, B., Shi, L., Cai, Z., Li, M., 2012. Software vulnerability discovery techniques: asurvey. In: 2012 Fourth International Conference on Multimedia Information Networking and Security. IEEE, pp. 152–156.

Liu, C., Zou, D., Luo, P., Zhu, B.B., Jin, H., 2018. A heuristic framework to detect concurrency vulnerabilities. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 529–541.

Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.-H., Song, Y., Beyah, R., 2019. {MOPT}: optimized mutation scheduling for fuzzers. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 1949–1966.

Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M., 2019. The art, science, and engineering of fuzzing: asurvey. IEEE Trans. Software Eng. 47, 2312–2331.

Mansur, M.N., Christakis, M., Wüstholz, V., Zhang, F., 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 701–712.

Mihailescu, M.I., Nita, S.L., 2021. Brute force and buffer overflow attacks. In: Pro Cryptography and Cryptanalysis with C++ 20. Springer, pp. 423–434.

Muench, M., Stijohann, J., Kargl, F., Francillon, A., Balzarotti, D., 2018. What you corrupt is not what you crash: challenges in fuzzing embedded devices. NDSS.
onl, 2010. fuzz testing (fuzzing). https://searchsecurity.techtarget.com/definition/fuzz-testing.
Österlund, S., Geretto, E., Jemmett, A., Güler, E., Görz, P., Holz, T., Giuffrida, C., Bos, H., 2021. Collabfuzz: a framework for collaborative fuzzing. In: Proceedings of the 14th European Workshop on Systems Security, pp. 1–7.
Ozment, J.A., 2007. Vulnerability Discovery & Software Security. University of Cambridge Ph.D. thesis.
Pak, B.S., 2012. Hybrid fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution. School of Computer Science Carnegie Mellon University.
Peng, H., Shoshitaishvili, Y., Payer, M., 2018. T-fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 697–710.
Pham, V.-T., Böhme, M., Roychoudhury, A., 2016. Model-based whitebox fuzzing for program binaries. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 543–553.
Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H., 2017. Vuzzer: application-aware evolutionary fuzzing. In: NDSS, vol. 17, pp. 1–14.
Saavedra, G. J., Rodhouse, K. N., Dunlavy, D. M., Kegelmeyer, P. W., 2019. A review of machine learning applications in fuzzing. arXiv preprint arXiv:1906.11133
Salim, M.M., Rathore, S., Park, J.H., 2020. Distributed denial of service attacks and its defenses in IoT: a survey. J. Supercomput. 76 (7), 5320–5363.
Serebryany, K., 2017. {OSS-Fuzz}-google's continuous fuzzing service for open source software.
Takanen, A., Demott, J.D., Miller, C., Kettunen, A., 2018. Fuzzing for Software Security Testing and Quality Assurance. Artech House.
Veggalam, S., Rawat, S., Haller, I., Bos, H., 2016. Ifuzzer: an evolutionary interpreter fuzzer using genetic programming. In: European Symposium on Research in Computer Security. Springer, pp. 581–601.
Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H., Sui, Y., 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). IEEE, pp. 999–1010.
Wang, J., Chen, B., Wei, L., Liu, Y., 2017. Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 579–594.
Wang, J., Chen, B., Wei, L., Liu, Y., 2019. Superion: grammar-aware greybox fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 724–735.
Wang, J., Duan, Y., Song, W., Yin, H., Song, C., 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID}), pp. 1–15.
Wang, Y., Jia, P., Liu, L., Huang, C., Liu, Z., 2020. A systematic review of fuzzing based on machine learning techniques. PLoS One 15 (8), e0237749.
Wen, C., Wang, H., Li, Y., Qin, S., Liu, Y., Xu, Z., Chen, H., Xie, X., Pu, G., Liu, T., 2020. Memlock: memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 765–777.
Wüstholz, V., Christakis, M., 2020. Harvey: a greybox fuzzer for smart contracts. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1398–1409.
Xu, W., Moon, H., Kashyap, S., Tseng, P.-N., Kim, T., 2019. Fuzzing file systems via two-dimensional input space exploration. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 818–834.
You, W., Liu, X., Ma, S., Perry, D., Zhang, X., Liang, B., 2019. Slf: fuzzing without valid seed inputs. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 712–723.
Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T., 2018. {QSYM}: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 745–761.
Zeng, P., Lin, G., Pan, L., Tai, Y., Zhang, J., 2020. Software vulnerability analysis and discovery using deep learning techniques: a survey. IEEE Access 8, 197158–197172.
Zhao, L., Duan, Y., Yin, H., Xuan, J., 2019. Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing. NDSS.

Craig Beaman is a graduate student at the University of New Brunswick, where he is completing a Master of Applied Cybersecurity. Craig received a B.Sc. (Honours) from the University of New Brunswick with a major in physics and minors in mathematics and computer science. His research interests include cryptography, network security, and malware detection and prevention.

Michael Redbourne is a graduate student at the University of New Brunswick, where he is completing a Master of Applied Cybersecurity.

John Darren Mummery is a graduate student at the University of New Brunswick, where he is completing a Master of Applied Cybersecurity.

Dr. Hakak is an assistant professor at the Canadian Institute for Cybersecurity (CIC), Faculty of Computer Science, University of New Brunswick (UNB). Having more than 5+ years of industrial and academic experience, he has received the number of Gold/Silver awards in international innovation competitions and is serving as the technical committee member/reviewer of several reputed conference/journal venues. His current research interests include Risk management, Fake news detection using AI, Security and Privacy concerns in IoE, Applications of Federated Learning in IoT, and blockchain technology.