SACJ 37(2) December 2025
Research Article

Improving greybox fuzzing with dictionary-based mutations: A systematic literature review

Enock L. Dube[a], Boluwaji A. Akinnuwesi[a], Stephen G. Fashoto[a,b], Petros M. Mashwama[a], Vusi W. Tsabedze[a]

[a] Department of Computer Science, University of Eswatini, Eswatini
[b] Department of Informatics, Namibia University of Science and Technology, Windhoek, Namibia

ABSTRACT
Detecting deep bugs that are guided by complex conditions, based on specific byte sequences of the input, often requires input structure-aware or grammar-aware fuzzing strategies. However, the grammar or specification of the input may not be readily available. In this regard, there exists anecdotal evidence that dictionary-based mutations contribute to preserving the syntactic structure of input test cases and may approximate the efficacy of grammar-aware fuzzing. It is not yet clear as to which is the best strategy for automatically extracting fuzzing dictionary tokens from the codebase of the program under test. In this study we conduct a systematic review of the impact of dictionary-based mutations on the fuzzing process. We further review strategies for automatically extracting dictionary tokens and optimizing dictionary-based mutations. Our findings are that current strategies for extracting fuzzing dictionary are not optimised for highly structured input. Furthermore, about 58% of the reviewed state-of-the art fuzzing tools rely on the random mutation operator distribution of respective baseline fuzzer. Moreover, the evaluation of these fuzzing tools report on aggregated performance of mutation operator scheduling algorithms, and not specific individual operators such as dictionary-based mutation operators.

Keywords Dictionary-based mutation, Mutational Fuzzing, Greybox fuzzing, Software vulnerability, Software testing

Categories • **Software and its engineering** ∼ **Software creation and management, Software verification and validation, Software defect analysis, Software testing and debugging**

Email
Enock L. Dube – eldube@uniswa.sz (CORRESPONDING)
Boluwaji A. Akinnuwesi – bakinnuwesi@uneswa.ac.sz
Stephen G. Fashoto – sgfashoto@uniswa.sz
Petros M. Mashwama – petros@uniswa.sz
Vusi W. Tsabedze – vtsabedze@uneswa.ac.sz

Article history
Received: 4 March 2025
Accepted: 15 July 2025
Online: 22 December 2025

1 INTRODUCTION

Fuzz testing, also known as fuzzing, is a software testing technique that can be used to detect faults or weaknesses such as correctness bugs and security vulnerabilities in an input-parsing

Dube, E.L., et al. (2025). Improving greybox fuzzing with dictionary-based mutations: A systematic literature review . *South African Computer Journal* 37(2), 74–103. https://doi.org/10.18489/sacj.v37i2.21430

SACJ is a publication of *SAICSIT*. ISSN 1015-7999 (print) ISSN 2313-7835 (online)

program (Liang et al., 2018). It works by repeatedly running the program under test (PUT) with mutated or fuzzed input test cases. Ever since it's conceptualisation in 1990 (Miller et al., 1990), fuzz testing remains a widely used technique, and it provides an inexpensive mechanism for eliciting faulty program behaviour. Compared to other vulnerability testing techniques, such as static analysis and penetration testing (Halfond et al., 2011; Liu et al., 2012), fuzz testing scales well to large programs (Godefroid, 2020). However, a major limitation of most fuzzing frameworks is that they often fail to deal with programs that require highly structured input data such as JavaScript and XML objects (Zalewski, 2015). In this regard, fuzzing highly structured input often generates invalid input test cases that are promptly rejected in the parsing phase of the target program under test. In addition to model-based fuzzing approaches, such as grammar-based fuzzing, dictionary-based mutational fuzzing was introduced to mitigate the limitations of the grammar-blind nature of fuzzing (Zalewski, 2015).

Dictionary-based mutation is implemented by most of the state-of-the-art fuzzers such as American Fuzzy Lop (AFL) (Zalewski, 2013), libFuzzer (Serebryany, 2015) and Honggfuz (Swiecki & Grobert, 2025). Despite anecdotal evidence (Zalewski, 2015) that dictionary-based mutation can approximate grammar-based input generation, and therefore improve efficiency of the fuzzing process, there exist a limited number of empirical studies on this approach. In this study, we conduct a systematic literature review on the impact of dictionary-based mutation on the fuzzing process. The significance of the study is that it provides a consolidated overview of dictionary-based mutational fuzzing. To the best of our knowledge, this is the first systematic literature review addressing this topic. The subsections that follow provide background information on the fuzz testing process and further discuss the dictionary-based mutation strategy in more detail.

1.1 Background of Fuzz Testing Process

Software vulnerabilities, such as memory corruption, remain a major cause of many severe threats to input parsing programs (Gan et al., 2018). Studies have shown that nation-state and independent hackers rely on the presence of software vulnerabilities to develop exploits that break down a target program execution with the intention of performing malicious actions such as control flow hijacking, information leakage, and denial of service attacks (Nagy & Hicks, 2019). Unlike independent hackers, nation-state hackers are sponsored by national governments to launch cyber-attacks that are typically aligned to political or military objectives. The target of these attacks could be an individual person, organisation or another state. Bugs and vulnerabilities in the software systems enable the execution of such attacks.

A software vulnerability is a defect that may originate from human mistake in the design of the software, and it may introduce an exploitable fault, flaw or bug in the code. A fault may result in a failure characterised by an observed deviation from expected behaviour of a program (IEEE, 1990). To identify and mitigate these vulnerabilities, software developers may employ various testing and vulnerability detection techniques, such as fuzz testing. The primary goal of this testing is to minimise the exploitability of a software system.

https://doi.org/10.18489/sacj.v37i2.21430

In principle, a vulnerability detection and analysis system should be sound and complete. A sound system never approves a vulnerable program (*no vulnerabilities are missed)* and is said to be complete if all secure programs can be approved (*no false vulnerabilities*). By extension, a vulnerability detection and analysis system is said to be both sound and complete if it can approve all secure programs and disapprove all vulnerable programs (*no missed vulnerabilities and no false vulnerabilities*) (Xie et al., 2005). In practice, however, most fuzz test techniques and strategies are neither sound nor complete (Ghaffarian & Shahriari, 2017). The goal of fuzz testing is to identify problematic program states which may be associated with security vulnerabilities (Serebryany, 2015). In recent years many fuzz testing techniques (King, 1976), tools (Böhme et al., 2019; Serebryany, 2015; Zalewski, 2013) and frameworks (Fioraldi, Maier et al., 2020) have been proposed, implemented and evaluated. These methods can be classified into three main strategic approaches: Blackbox, Greybox and Whitebox fuzzing. Figure 1 provides a workflow that summarises the common processing steps shared by these three different approaches.

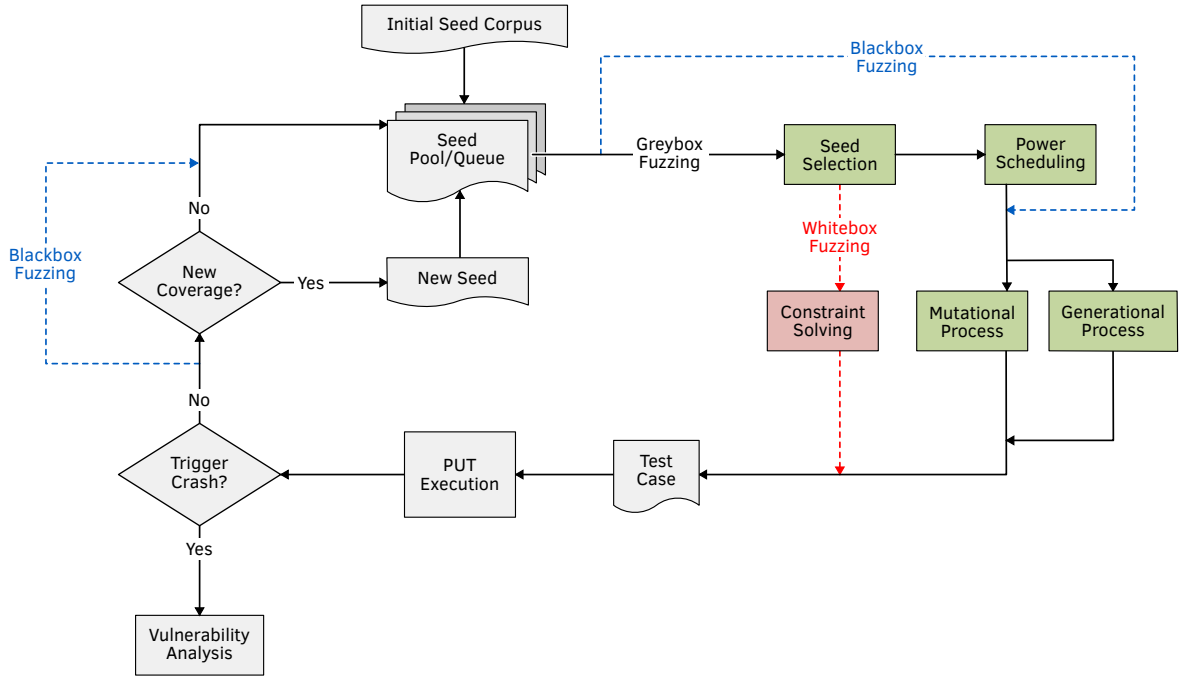

Figure 1: Workflow of Different Fuzzing Approaches[a]

[a] Adapted from P. Wang et al. (2024)

As depicted in Figure 1, the fuzzing process is a loop that commences by selecting a seed or input test case from a seed pool or queue. It uses a power schedule to determine the seed's energy. The energy specifies how many times each selected input may be mutated. The term power schedule refers to a mechanism that assigns more energy to interesting input test cases such as those that explore more code paths. The next step executes the program under test

using the newly generated input test case. The program execution is monitored to determine if it leads to a failure which must then be analysed to determine the presence of a bug or flaw in the code. As indicated in Figure 1, some steps may be skipped depending on the fuzzing strategy adopted.

All three fuzzing strategic approaches repeatedly generate input test cases and execute the program under test while monitoring the program state to detect any abnormal behaviour caused by correctness bugs or security vulnerabilities in the code. Blackbox fuzzing (indicated by blue lines in Figure 1) randomly generates new input test cases and skips the input selection and power scheduling steps. In general, Blackbox fuzzing tests the specified behaviour of a target PUT and often does not require the source code (Zeller et al., 2024). It is widely applicable without prior knowledge of internal program constructs. While random test case generation in Blackbox fuzzing is fast and scales well to large programs, its major limitation is that it may generate a lot of invalid input test cases. As a result, Blackbox fuzzing may take longer execution time to generate a new test case that triggers a failure in the target PUT. There is no guarantee that Blackbox fuzzing will identify all vulnerabilities and not raise false positives. In this regard, Blackbox fuzzing is neither sound nor complete (Ghaffarian & Shahriari, 2017).

On the other hand, Whitebox fuzzing (indicated by the red lines in Figure 1) leverages program analysis techniques, such as symbolic execution (Cadar & Sen, 2013; King, 1976) and constraint solving techniques (De Moura & Bjørner, 2008), to generate new input test cases. Symbolic execution analyses a program and determines what inputs can explore each execution path in the code. Whereas Blackbox fuzzing tests the specified behaviour of the PUT, Whitebox fuzzing tests the implemented behaviour and often requires an analysis of the source code. When applied to small code components, such as unit testing, Whitebox fuzzing is guaranteed to generate input test cases that exercise all path conditions in the program under test, increasing the likelihood to identify bugs and detect software vulnerabilities (Böhme et al., 2019). In this context, Whitebox fuzzing may be sound and complete. However, the symbolic execution and constraint solving techniques deployed in Whitebox fuzzing are computationally expensive and do not scale well to large programs (Godefroid et al., 2008).

Greybox fuzzing combines aspects of both Blackbox and Whitebox fuzzing techniques and may work on both binary and open-source code. Whereas Blackbox fuzzing generates input test cases without any internal knowledge of the program under test, Greybox fuzzing leverages program instrumentation to generate feedback information, such as code coverage, which may be used to guide the input generation process. Additionally, Greybox fuzzing incorporates some aspects of Whitebox fuzzing, such as feedback information, which provides a Greybox fuzzer the ability to acquire some internal knowledge of the target PUT. However, Greybox fuzzing may avoid or limit the use of computationally expensive techniques such as symbolic execution. Similar to Blackbox fuzzing, Greybox fuzzing is neither sound nor complete. It provides no guarantee of identifying all vulnerabilities and may still raise false vulnerabilities (Ghaffarian & Shahriari, 2017).

Greybox fuzzing approaches may be further divided into three categories: coverage-based

greybox fuzzing (CGF), directed greybox fuzzing (DGF) and regression greybox fuzzing (RGF). Whereas CGF seeks to cover all code paths in the target program under test, DGF focuses on reaching specific target locations, therefore reducing computational resource wastage on exploring unrelated code paths (Böhme et al., 2017). On the other hand, regression greybox fuzzing is an advanced technique that focuses on recently changed or frequently modified code segments (X. Zhu & Böhme, 2021). Directed greybox fuzzing has gained prominence as an efficient approach for targeted software testing in specific scenarios such as patch testing and bug reproduction (P. Wang et al., 2024). Most of the state-of-the art fuzzing tools are coverage-based greybox fuzzers. Examples in this category include AFL (Zalewski, 2013), AFLFast (Böhme et al., 2019), AFL++ (Fioraldi, Maier et al., 2020), ATTUZ (S. Zhu et al., 2024), and FIRM-COV (Kim et al., 2021). Directed greybox fuzzing tools include ALFGo (Böhme et al., 2017), GTFuzz (Li et al., 2020), and Hawkeye (H. Chen et al., 2018). Examples of regression greybox fuzzers include AFLCHURN (X. Zhu & Böhme, 2021). Greybox and Whitebox fuzzing may be combined to build hybrid fuzzing tools that leverage features of both approaches. Examples of hybrid fuzzers include Driller (Stephens et al., 2016), Vuzzer (Rawat et al., 2017), Angora (P. Chen & Chen, 2018) and Savior (Y. Chen et al., 2020).

The following section introduces a working example that is used to elaborate on the different fuzzing approaches and further clarifies some of the terminology used in this study.

1.2 Working Example: PNG Parser

Listing 1 shows sample code that may be used to parse and process a Portable Network Graphics (PNG) binary file. A PNG file starts with an 8-byte signature that identifies the file as containing a PNG image, followed by sequence of chunks or sections.

The PNG signature consists of the following hexadecimal byte sequence: `0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A`. The hexadecimal value `0x89` refers to the ordinal value of the per mile symbol (‰) in the American Standard Code for Information Interchange (ASCII) character set. The values `0x50`, `0x4E` and `0x47` are the ordinal values for character symbols *P*, *N* and *G*. Similarly, `0x0D` and `0x0A` are values for carriage return (*CR*) and linefeed (*LF*) control characters respectively. The value `0x1A` represents the substitute (*SUB*) control character. The values in the PNG signature are examples of magic bytes. The term magic byte or magic value refers to specific bytes of the input sequence that must be matched to access certain code paths in an input parsing program. As shown in Listing 1, a program that processes a PNG binary file contains conditional statements that verify that the file contains the appropriate signature and is not corrupted. Ideally, a PNG parsing program, such as the *PNG_parser* function in Listing 1, may only process a PNG file with a valid signature.

```
#define MAX_SIZE (16 * 1024 * 1024)   // 16 megabytes

bool check_png_signature(char input_buffer[], int num_bytes_to_check)
{
    // PNG signature magic values
    const char png_signature[8] = {
        0x89, 0x50, 0x4E, 0x47,
```

```
        0x0D, 0x0A, 0x1A, 0x0A
    };

    // C1: limit to bytes [1..8]
    if ((num_bytes_to_check < 1) || (num_bytes_to_check > 8)) {
        return false;
    }
    else
    // C2: check hard-coded PNG signature magic values
        return (memcmp(input_buffer, png_signature, num_bytes_to_check) == 0);
}


void PNG_parser(const char *png_input_file, int buffer_size)
{
    char *input_buffer = (char *)malloc(buffer_size);
    if (input_buffer == NULL) {
        // TODO: handle allocation error
        return;
    }

    // ...

    FILE *infile = fopen(png_input_file, "rb");
        // Handle file open failure
        // ...

    int size = fread(input_buffer, 1, buffer_size, infile);

    // C3: check magic bytes
    if (check_png_signature(input_buffer, 8) == true) {

        int chunk_start_pos = 8;     // chunk start position; after signature
        // BUG1: buffer over-read if input_buffer < 8 bytes → validate size first

        while (chunk_start_pos < size) {

            char chunk_lenbuffer[4];
            memcpy(chunk_lenbuffer, input_buffer + chunk_start_pos, 4);

            int chunk_length = get_big_endian(chunk_lenbuffer);

            char chunkbuf[5];
            int chunk_type_start_pos = chunk_start_pos + 4;

            // BUG2: buffer over-read if input_buffer < 13 bytes → validate first
            for (int index = 0; index < 4; index++) {
                chunkbuf[index] = input_buffer[chunk_type_start_pos + index];
            }

            chunkbuf[4] = '\0';

            // process chunk here
            // ...
        }
    }

    // close file and free buffer
    // ...
}
```

Listing 1: Sample code to parse a PNG file

https://doi.org/10.18489/sacj.v37i2.21430

The PNG signature is followed by a sequence of chunks/sections. Each chunk is indicated by a chunk code or type such as *IHDR*, *IEND*, *IDAT*, and *PLTE*. The codes are 4-byte sequences consisting of hexadecimal representation of each character symbol in the code name. For instance, the hexadecimal values of characters *I*, *D*, *A*, *T* in the *IDAT* code. The chunk types/codes may also be considered magic bytes or values that are often used in complex conditional statements and therefore, guard the execution of certain blocks of code in a PNG file parsing program.

A PNG file begins with an *IHDR* chunk and ends with an *IEND* chunk. Between *IHDR* and *IEND* other chunk types, such as *IDAT* and *PLTE*, may appear multiple times. Figure 2 shows the format of each chunk.

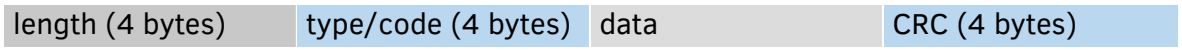

Figure 2: PNG chunk format

Each chunk starts with a 4-byte length field which specifies the size of the PNG image. The length field is followed by a 4-byte type field (for instance *IHDR*, a data field, and a 4-byte Cyclic Redundancy Check (CRC). The value of the length field depends on the size of the image represented in the data field. The CRC is a checksum value that is calculated based on byte sequences in the chunk type field and data field. In this regard, when parsing a PNG file, the CRC may be used to validate that the data is not corrupted. The RFC 2083 PNG specification (Boutell, 1997) contains a more detailed discussion of the PNG file format.

The *PNG_parser* function in Listing 1 attempts to first read and validate the PNG signature, and then proceeds to extract information from the collection of chunks. The code contains buffer over-read bugs or vulnerabilities (indicated as *BUG1* and *BUG2* in Listing 1). A buffer over-read is a type of vulnerability that occurs when code execution tries to read data beyond the end of a buffer in memory.

The *check_png_signature* function in Listing 1 takes an input buffer, representing a PNG file content, and validates the signature. Since the signature refers to the first 8 bytes of the input buffer, the first conditional statement (*C1*) limits the values for the second argument, *num_bytes_to_check*, to between 1 and 8.

In the context of Blackbox fuzzing, a randomly generated value for *num_bytes_to_check* will frequently generate a lot of values that are either greater than 8 or less than 1. In other words, the probability of randomly generating a 4-byte integer value between 1 and 8 is small $(8/2^{31})$. In this regard, the *check_png_signature* function often evaluates to false, and code execution fails to reach the validation code in conditional statement *C2*. Blackbox fuzzing will also take a long time to randomly generate an input that directs code execution to go beyond conditional statement *C3* in the *PNG_parser* function. Inevitably, it will take a longer time to identify the bugs or vulnerabilities (*BUG1* and *BUG2*) guarded by *C3*. The conditional statements *C1*, *C2* and *C3* are examples of sanity check. The term sanity check refers to security measures that are used to verify the validity of the input and help ensure that the code processes it correctly.

https://doi.org/10.18489/sacj.v37i2.21430

On the other hand, a Whitebox fuzzing approach may use symbolic execution to reach and explore the vulnerable code segments much faster than Blackbox fuzzing. A symbolic execution of the *check_png_signature* function identifies a symbolic value, such as a_1 , and assigns it to variable *num_bytes_to_check*. Upon reaching the conditional statement *C1*, it would evaluate the path constraint: $(a_1 < 1) \bigvee (a_1 > 8)$. At this point of the code execution, a_1 could take any value, and symbolic execution can fork into two paths and proceed along both branches of the if-else conditional statement *C1*. Each path gets assigned a copy of the program state at the branch instruction as well as a path constraint. In this example, the path constraint is $(a_1 < 1) \bigvee (a_1 > 8)$ for the *IF* branch and its negation, $(a_1 \geq 1) \bigwedge (a_1 \leq 8)$, for the *ELSE* branch. Both paths can be symbolically executed independently of one another. When each path's execution terminates, symbolic execution computes a concrete value for a_1 by solving the accumulated path constraints on each path. These concrete values can be thought of as concrete input test cases. In this example, the constraint solver would determine that in order to reach the else-branch of conditional statement *C1*, a_1 should be a value between 1 and 8. This branch executes the validation code in conditional statement *C2*. In this regard, Whitebox fuzzing generates valid input test cases significantly faster than Blackbox fuzzing and is often able to systematically explore the vulnerable code segments (*BUG1* and *BUG2*) guarded by conditional statement *C3*. However, for large programs with nested conditional statements, the path constraints may increase in length and complexity, making it difficult for a constraint solver to resolve.

1.3 General Weaknesses of Blackbox, Greybox, and Whitebox Fuzzing

The general weaknesses of Blackbox, Greybox, and Whitebox fuzzing lie in their limitations regarding scalability, accuracy, and computational cost. Blackbox fuzzing, while fast and scalable to large programs, suffers from inefficiency in identifying bugs, as it often generates a high volume of invalid input test cases. As illustrated in Section 1.2, randomly generated input test cases often fail sanity checks. It may take a lot of trials to generate a test case that reaches and executes the vulnerable code segments. This may contribute to a slower rate of vulnerability detection. Whitebox fuzzing, which is exhaustive in exploring program paths through symbolic execution, faces scalability issues in larger programs that contain complex path constraints. The constraints solver may fail to resolve the path constraints. Greybox fuzzing represents a middle ground between Blackbox and Whitebox fuzzing and uses code coverage feedback to guide input test case generation. However, Greybox fuzzing is prone to missing certain vulnerabilities and may raise false alarms, as it lacks the completeness of Whitebox fuzzing.

1.4 Mutational vs Generational Fuzzing

Mutational or mutation-based fuzzing treats program inputs as a sequence or array of bytes, and repeatedly applies byte-level mutation operators to generate new input test cases. These

include bitflip & byteflip operators, arithmetic increment & decrement operators, byte insertion, deletion & overwrite operators, and dictionary-based mutation operators. These operators characterise which bytes in the input byte-array to mutate and how to mutate them. For instance, the *check_png_signature* in Listing 1 takes two arguments: An *input_buffer* array and *num_bytes_to_check* integer value. As shown in Figure 3, mutational fuzzing views these two values as a single sequence or array of bytes. In this view, the *num_bytes_to_check* bytes are concatenated at the end of the *input_buffer* bytes, to form one byte sequence.

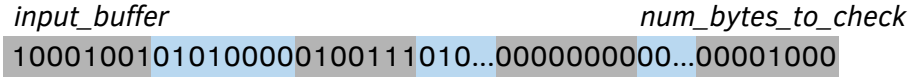

Figure 3: Sample representation of input as sequence of bytes

Mutational fuzzing repeatedly applies mutation operators to the input byte sequence. In most mutational fuzzing tools such as AFL (Zalewski, 2013), the fuzzing process begins with a deterministic stage followed by a havoc (non-deterministic) stage. The deterministic stage is executed only once for each seed input. It starts at the beginning of each seed input byte sequence, sequentially selects each byte and applies each available mutation operator. The deterministic stage may also be used to assign initial energy to each seed in the corpus. On the other hand, the havoc stage randomly selects a mutation operator and applies it to a randomly selected byte location in the input byte sequence. Some fuzzers, such as libFuzzer (Serebryany, 2015) do not implement the deterministic stage and are often referred to as non-deterministic fuzzers. However, a majority of mutational fuzzers implement both stages and apply byte-level mutations to generate new input test cases.

Byte-level mutations are often syntax blind and therefore may generate many invalid input test cases. For instance, randomly mutating the *input_buffer* bytes (in Figure 3) may alter the signature resulting in the file content not recognisable as a PNG image. For instance, a single bitflip operation on the second byte in Figure 3 may change it from `01010000` to `01010001` resulting in a signature that contains the character sequence QNG instead of PNG. This operation generates an invalid input that is rejected by the parsing process since it fails the validation check. In this regard, whereas byte-level mutations may be successful for simple formats, they generate a high number of invalid test cases for highly structured input formats such PNG files. As another example, randomly mutating the *input_buffer* (in Figure 3) may distort the structure of the PNG chunks. For instance, a number of bitflip operations that changes the byte representation for character ***R*** in the ***IHDR*** chunk code from `01010010` to `00100011` will mutate the chunk type/code to be ***IHD#***. Since the CRC checksum is calculated based on the chunk type field and chunk data field, this mutation results in a corrupted IHDR chunk with an incorrect CRC checksum. In this example, unless the *PNG_parser* code has sufficient sanity checks to validate the syntactic structure of the PNG chunks, it may fail to correctly recognise ***IHDR*** header chunk or may compute the wrong checksum. For a target parsing program with strong sanity checks, many of the syntactically invalid inputs may be rejected or may fail to explore and test the vulnerable code segments. To mitigate this challenge, a syntax-aware

approach such as generational or grammar-based fuzzing may be used.

Generational or generation-based fuzzing (Eberlein et al., 2020; Godefroid et al., 2008), sometimes referred to as grammar-based fuzzing, generates input test cases from an input language specification such as grammar or a set of rules. Its main advantage is that it guarantees that all generated test cases are syntactically valid hence the fuzzing process does not waste computational resources parsing invalid inputs. However, since most programs lack a formal grammar of well-formed inputs, mutational fuzzing techniques are widely adopted.

Most fuzzing tools adopt mutation-based strategies and repeatedly apply syntax-blind byte-level mutations. As already noted, mutational fuzzing may generate a high number of invalid test cases. To mitigate this challenge, dictionary-based mutation was introduced to make-up for the grammar-blind nature of AFL (Zalewski, 2015). Subsequently, dictionary-based mutation has been implemented in other state-of-the-art fuzzers such as libFuzzer (Serebryany, 2015) and Honggfuzz (Swiecki & Grobert, 2025). In Greybox fuzzing, the feedback provided by the compile-time instrumentation makes it possible to identify syntax tokens in some types of files, and further detect that certain combinations of terms constitute a valid grammar of the input test cases. In this regard, fuzzing dictionaries have been shown to enable the fuzzer to rapidly reconstruct the grammar of highly verbose languages such as XML, SQL and JavaScript (Zalewski, 2015).

A fuzzing dictionary is a text file that contains a collection of commonly occurring keywords, strings, interesting byte sequences and magic bytes/values. As discussed in the working example in Section 1.2, magic bytes such as the PNG signature, `0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A` and chunk codes (***IHDR***, ***IEND***, ***IDAT***, ***PLTE***), may be extracted to a fuzzing dictionary. The Google Fuzzing Dictionaries (Google, 2025) repository provides examples of fuzzing dictionary for different file formats.

1.5 Dictionary-Based Mutation

A fuzzing dictionary consists of a list of basic syntax tokens. These tokens may be manually extracted or automatically identified during the fuzzing process. Most fuzzing frameworks implement two types of dictionary-based mutation operators: *insert* and *overwrite*. Whereas other fuzzing tools may use different naming conventions, in AFL these operators are called *user extras* operator and *auto extras* operator respectively. The AFL user extras operator selects a token from a fuzzing dictionary and inserts its bytes into the input test case. The AFL auto extras operator overwrites bytes in the input test case with a dictionary token recognized and automatically extracted by AFL during deterministic stage. Other coverage-based mutational fuzzers such as libFuzzer and Honggfuzz implement similar mechanisms for dictionary-based mutation. While AFL performs dictionary-based operations during the deterministic stage as well-as havoc stage of fuzzing, libFuzzer is non-deterministic and performs dictionary-based mutation only during the random havoc stage. The general pseudocode for dictionary-based mutation is shown in Algorithm 1 which is adapted from Shastry et al. (2017).

```
function DICTIONARY_FUZZ(input_bytes, dictionary, deterministic)

    dict_token ← selectRandomTokenFrom(dictionary)

    if deterministic = TRUE then
        for each offset in input_bytes do
            FUZZ_TOKEN_OFFSET(input_bytes, dict_token, offset)
        end for
    else
        offset ← selectRandomOffsetPosition(sizeOf(input_bytes))
        FUZZ_TOKEN_OFFSET(input_bytes, dict_token, offset)
    end if

end function



function FUZZ_TOKEN_OFFSET(input_bytes, dict_token, offset)

    overWriteFromOffsetPosition(input_bytes, offset, dict_token)
    runProgramUnderTest(program, input_bytes)

    insertTokenAtOffsetPosition(input_bytes, offset, dict_token)
    runProgramUnderTest(program, input_bytes)

end function
```

Algorithm 1: Pseudocode for Dictionary-based Mutation

Algorithm 1 is implemented in most Greybox fuzzers such as AFL, Honggfuzz and libFuzzer. During dictionary-based mutation, a token is selected from the dictionary and inserted between bytes or written over byte sequences of the same length. Whereas such dictionary-based mutations can generate syntactically valid input test cases, they can also destroy the syntax structure of the input tests case. In this regard, a study (J. Wang et al., 2019) conducted in 2019 proposed an enhanced dictionary-based algorithm that identifies an ideal location to insert or overwrite bytes sequences and therefore preserve syntax structure. The results of the study provide evidence that dictionary-based mutations help improve validity of generated input test cases, resulting in improved performance of fuzz testing. The performance of fuzzing may be measured in terms of effectiveness and efficiency. The term effectiveness refers to the total number of vulnerabilities discovered, and the term efficiency refers to the rate at which vulnerabilities are discovered. Whereas counting the number of discovered vulnerabilities is an ideal metric for fuzzing effectiveness, it is not always feasible. For instance, a fuzz testing run that does not discover any unknown bugs or vulnerabilities is not necessarily ineffective. As a result, proxy metrics such as path or code coverage are often used as approximate measures of effectiveness. That is, a fuzzing technique that explores a high proportion of code execution paths is more likely to a discover a bug or vulnerability and therefore could be considered highly effective. Similarly, low code coverage reduces the likelihood of bug or vulnerability discovery.

The focus of this study is on dictionary-based mutation as a targeted approach to enhance fuzz testing efficiency and effectiveness. First, we interrogate the evidence that dictionary-

based mutation has an impact on the fuzzing process. We further investigate the impact of mutation operator selection and scheduling. The study objective and research questions are presented in Section 1.6.

1.6 Objective and Research Questions

The primary objective of this study is to investigate the role and impact of dictionary-based mutations in generating valid input test cases for fuzz testing. It reports on strategies and techniques that have been employed in previous studies to extract fuzzing dictionary tokens from the codebase of the program under test. Furthermore, the review examines how dictionary-based mutations have been used to improve the effectiveness and efficiency of the fuzzing process in the context of other optimisation techniques such as mutation operator selection and scheduling. In order to achieve this objective, the review process is guided by the following research questions (RQ):

RQ1: How does dictionary-based mutation impact fuzz testing effectiveness and efficiency?

RQ2: What approaches and techniques can be used to extract fuzzing dictionary tokens from the codebase of the program under test?

RQ3: What are the limitations of dictionary-based mutations in fuzz testing?

RQ4: What strategies can be used to optimize dictionary-based mutation operator selection and scheduling?

RQ5: How can alternative augmentation strategies be deployed to resolve the limitations of dictionary-based mutations?

We chose **RQ1** to examine the impact of dictionary-based mutations on the fuzz testing process. **RQ2** expands from **RQ1** and focuses on different approaches, and the techniques, which have been applied to extract fuzzing dictionary tokens. **RQ3** seeks to identify limitations of dictionary-based fuzzing and articulate research gaps. In **RQ4** we investigate strategies that have been used to optimize dictionary mutation operator selection and scheduling. We further critically analyse how dictionary-based mutations may be scheduled more optimally to positively impact the fuzzing efficiency and effectiveness. **RQ5** seeks to explore alternative strategies that may be deployed to address the limitations of dictionary-based mutation.

The rest of the paper is structured as follows: Section 2 presents the research methodology. A discussion of the results follows in Section 3, and we present future works and conclusion in Sections 4 and 5 respectively.

https://doi.org/10.18489/sacj.v37i2.21430

2 METHODOLOGY

A Systematic Literature Review (SLR) was conducted following the guidelines proposed by Kitchenham (2007). It is guided by the objective and the research questions. The SLR processes are discussed in this section. Furthermore, this section details the search strategy along with the inclusion and exclusion criteria used to identify the relevant literature.

2.1 Search Query

In the systematic review of literature, relevant primary studies were extracted based on the following search terms: **fuzzing**, **fuzz testing**, **mutation**, and **dictionary**. We combined the search terms to create the following search string: **("fuzzing" OR "fuzz testing") AND "mutation" AND "dictionary"**. Inclusion and exclusion criteria were used to filter the query results.

2.2 Information Sources

The primary sources used for the literature search included the following databases: Association of Computing Machinery (ACM) Digital Library, IEEE Explore and ScienceDirect. The Google scholar database search engine was used to find additional publications from proceedings of top-level conferences on security and software engineering such as the USENIX Security Symposium, and the Network and Distributed System Security Symposium (NDSS). It is worth noting that a significant number of publications on fuzz testing are discussed in the USENIX Security symposium.

2.3 Inclusion and Exclusion Criteria

The main inclusion criterion is that only peer reviewed publications are considered for this review. In addition, the publication must be written in English, its content must cover the scope of the search terms and must have been published between the years 2010 and 2024. The year range is based on the observation that a significant large number of publications on fuzz testing appeared during that period. In the same period, current state-of-the-art fuzzing tools such as AFL and libFuzzer were developed. Furthermore, only primary studies appearing in journals and conference papers were included in this review. Therefore, book chapters, literature review papers, editorials, comments, conference keynotes and short papers (less than 4 pages) were all excluded.

2.4 Screening of Publications

In addition to the inclusion and exclusion criteria, each paper publication was screened based on the title and abstract to determine its relevance with regards to the objective of the literature review. Publications with irrelevant content were removed. The focus was on application

software file-based fuzzing publications, as opposed to other fuzzing domains such as kernel fuzzing, network protocol fuzzing and parametric fuzzing. In this regard, we excluded publications focusing on fuzzing interfaces on hardware, central processing unit (CPU), embedded devices, Internet-of-Things (IoT), Industrial control systems, Android systems, kernels and network protocols.

2.5 Assessment of Literature search

The quality assessment criteria (QAC) that were used to screen each publication is shown in Table 1. It is based on the publication, methodology and whether it addresses specific aspects of dictionary-assisted mutational fuzzing. The overall design of the assessment form was adapted from a study conducted by Raharjana et al. (2021) by formulating specific questions appropriate for this review.

Table 1: Quality Assessment Form

Item	Quality Assessment Criteria (QAC)	Score and Description
QAC1	Is the goal of the research study clearly stated?	-1: NO, 0: Partially, 1: YES
QAC2	Is the research methodology described in detail?	-1: NO, 0: Partially, 1: YES
QAC3	Does the study address specific aspects of dictionary-based mutation?	-1: NO, 0: Partially, 1: YES
QAC4	Is the proposed dictionary-based mutation strategy or intervention proven to work or sufficiently evaluated?	-1: NO, 0: Partially, 1: YES
QAC5	Do other scholarly publications reference the study?	-1: NO, 0: Partially, 1: YES

A Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) (Page et al., 2021) flow diagram was used to summarise the results of the screening process and assessment criteria, as shown in Figure 4.

Based on the inclusion and exclusion criteria, the database search engines (ACM Digital Library, IEEE Xplore, and ScienceDirect) were configured to retrieve only Journal and conference papers. The initial ACM Digital Library search resulted in 203 papers, the IEEE Explore search resulted in 211 papers, and ScienceDirect search resulted in 31 papers. The papers were read in more detail, and some were removed because the content did not match the main objective and scope of this review. For instance, some publications did not address any aspect of dictionary-based mutation.

Of the retrieved studies, twenty-five (25) met the inclusion criteria and were judged suitable to address the research questions. Each of these publications articulated a clear research objective, described the methodology in detail, and were cited in subsequent scholarly literature publications. Seven of the twenty-five studies specifically investigated methods for extracting a fuzzing dictionary from a target program's codebase.

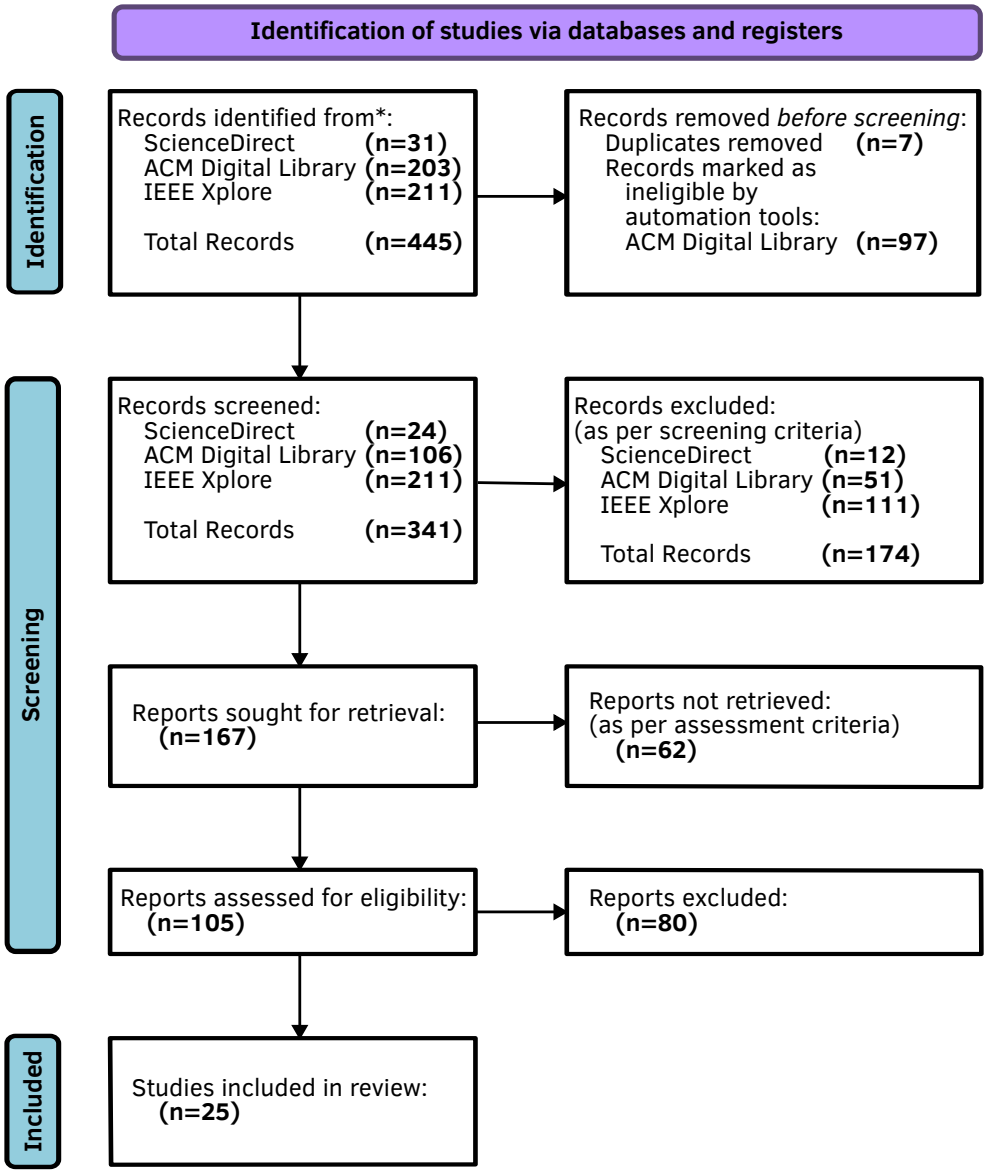

Figure 4: PRISMA Flow Diagram for Literature Search Results

2.6 Literature Data Extraction and Synthesis

To facilitate a synthesis of the publications, literature data was extracted and summarised using a data extraction form that summarises information appropriate to answering each of the review questions. For each reviewed publication, we recorded information such as the research focus, method, design, evaluation metric and limitations identified.

https://doi.org/10.18489/sacj.v37i2.21430

3 RESULTS AND DISCUSSION

Using the research questions (RQ) as a guide, and based on data extracted in Section 2.6, this section provides an analysis and synthesis of the identified publications.

3.1 Impact of dictionary-based mutation on fuzz testing effectiveness and efficiency (RQ1)

State of the art fuzzing tools such as AFL (Zalewski, 2015) have provided anecdotal evidence that user defined fuzzing dictionaries can improve code coverage. The implementation of both AFL and libFuzzer provide a collection of predefined general-purpose fuzzing dictionaries. However, these dictionaries are manually extracted, by application domain experts, from the codebase of the program under test. The Google Dictionaries project (Google, 2025) also maintains a collection of general-purpose fuzzing dictionaries and makes them available as open source. A few studies have evaluated the impact of target specific fuzzing dictionaries that are automatically extracted from the codebase of the program under test (Ebrahim et al., 2022; Shastry et al., 2017; Wu et al., 2025). An evaluation of results from these studies has shown that target specific fuzzing dictionaries are more effective than general-purpose dictionaries.

The FuzzingDriver (Ebrahim et al., 2022) is one of the latest frameworks proposed to automatically extract fuzzing dictionary tokens. It can be executed before the fuzzing process runs, and therefore, does not add any computational overhead to the fuzzing process. The FuzzingDriver is generic and therefore, may be used to extract dictionary tokens from any target program. An evaluation of the FuzzingDriver extracted dictionaries, against general-purpose Google fuzzing dictionaries (Google, 2025), showed improved code coverage. According to the developers of the FuzzingDriver framework, the efficacy of the FuzzingDriver on bug coverage is yet to be assessed.

On the same note, the Customized Dictionary Fuzzing (CDFUZZ) (Wu et al., 2025) tool evaluated the impact of target specific dictionaries extracted using the FuzzingDriver. The study concluded that the dictionary-based mutation strategy improved fuzzing effectiveness and performed better than other fuzzing exploration strategies such as input-to-state correspondence proposed in REDQUEEN (Aschermann et al., 2019) fuzzing tool, the QSYM (Yun et al., 2018) SMT-solver based concolic engine, and gradient-based algorithm proposed in Angora (P. Chen & Chen, 2018).

The Orthrus framework (Shastry et al., 2017) proposed a mechanism that uses static compile-time analysis to extract fuzzing dictionary tokens. On evaluation of tcpdump network protocol, the Orthrus dictionary showed up to 10% improvement on code coverage compared to baseline AFL, and up to 8% improvement compared to baseline AFLFast (Böhme et al., 2019). Similarly, an evaluation on Deep Network Traffic Inspection (nDPI) showed up to 15% and 5% improvement compared to baseline AFL and AFLFast respectively. The results of the evaluation also showed improved bug coverage. Although Orthrus was originally tested

only on network protocol messages, the authors indicate that its design could be extended to support file format parser applications such as PNG parsers.

In a study conducted in 2020, Mathis et al. proposed a fuzzing tool called LFuzzer that extends Dynamic Taint Analysis (DTA) to not only automatically infer fuzzing tokens, but also generate seed input test cases. DTA is a data flow tracking technique that is widely used for vulnerability analysis. It tracks tainted data during program execution and therefore, provides insight into how data flows through the program at runtime. The set of tokens inferred while using LFuzzer can be added to a fuzzing dictionary. The study also evaluated the LFuzzer extracted dictionary on highly structured input formats such as JSON and LISP. The findings indicate that, on average, the proposed approach achieves up to 17% more code coverage compared to baseline AFL.

Another study (Bundt et al., 2021) was conducted to explore the difference between fuzzing synthetic bugs and real-world bugs. The study concluded that coverage-guided fuzzers can effectively discover synthetic bugs in a LAVA (Dolan-Gavitt et al., 2016) generated synthetic dataset with techniques such as dictionary-based mutation and comparison splitting. The study utilised a dictionary of constant values parsed from a dis-assembly of the target program.

In 2021 Metzman et al. proposed and implemented a benchmarking platform and evaluated several fuzzing techniques. On the aspects of fuzzing dictionary, the results reflected minor differences between fuzzing with or without dictionary-based mutation enabled.

As highlighted in this section, there exist a limited number of research studies that provide empirical evidence of the impact of dictionary-based mutation on the efficiency and effectiveness of the fuzzing process. Moreover, a few studies provide details of how to automatically extract dictionary tokens from the codebase of the program under test. Section 3.2 looks at different strategies that have been used to extract fuzzing dictionary tokens.

3.2 Strategies for extracting fuzzing dictionary tokens from the program under test code base (RQ2)

The task of generating a dictionary from the codebase of the target PUT is strenuous and time consuming (Ebrahim et al., 2022). As a result, a lot of fuzzing tools and frameworks use predefined general-purpose dictionaries such as those provided by Google Fuzzing Dictionaries (Google, 2025). Furthermore, the source code release of AFL contains generic dictionaries for many common file formats which can be used in fuzzing runs. Moreover libFuzzer and Honggfuzz support AFL dictionaries and OSS-fuzz (Ding & Goues, 2021) integrates dictionaries in some of its fuzzing projects. While these generic dictionaries may improve code coverage, they lack target-specific information, and the impact is not optimised (Ebrahim et al., 2022). Additionally, manual extraction of a fuzzing dictionary requires expert knowledge. Therefore, in real world applications, there is a need to automatically generate target specific fuzzing dictionaries. Table 2 summarises the different approaches used to extract dictionary tokens.

The most basic strategy to create a fuzzing dictionary is to scrape through the source code of the target PUT and extract constants and string literals to a fuzzing dictionary. Some stud-

Table 2: Summary of Token Extraction Methods

EM#[a]	Method / Strategy	Fuzzing Tool	S/B[b]
EM_01	Domain expert manually extracts dictionary tokens	AFL libFuzzer	S
EM_02	Use byte and string comparison to identify tokens	AFL	B
EM_03	Auto-detect tokens during deterministic bitflip stage by looking for groups of bits that always produce the same coverage even after mutation	AFL	B
EM_04	Scrape the application binary code to create a dictionary	AFL & Thompson Sampling Deep Reinforcement Learning CONFETTI	B
EM_05	Syntactically extract input fragments from program abstract syntax tree (AST) and semantically extract conjunctions of input fragments from the program control flow graph (CFG) by statically analysing program data flow and control flows.	Orthrus	S
EM_06	Use CodeQL language queries to extract valuable information such as commonly occurring keywords, strings, and arguments of comparison functions	FuzzingDriver	S
EM_07	Use context sensitive data-flow analysis; leverage Guard Tokens; Extract strings from comparison functions and variable definitions	GTFuzz	S
EM_08	Auto-tokens – extracts tokens from instructions and comparison function	AFL++ libAFL	S
EM_09	Extract tokens from comparison instructions and functions with immediate values	libAFL	B
EM_10	Extracts all strings from binary codebase and selects strings used as arguments to library API and add to dictionary	FIRM-COV	B
EM_11	Learning input tokens; Explores branches of lexical analysis and extract dictionary tokens;	LFuzzer	B
EM_12	Uses extended dynamic tainting of implicit data transformation to produce/infer dictionary tokens and seed input test cases.	LFuzzer	B
EM_13	Leverage Input-to-state(I2S) correspondence and add all values that contain non-zero or non-0xff bytes to a specific fuzzing dictionary	FairFuzz REDQUEEN	B
EM_14	During compilation, extract all constant string comparison function parameters to a file; Need link-time optimisation (LTO) to pass auto-tokens with no overhead;	AFL++ libAFL	S
EM_15	Compile subject into human readable bit code format, and then extract string literals by iterating over the global values of the bit code file and writing the global strings to fuzzing dictionary	LFuzzer	B
EM_16	Use Linux strings tool to extract printable strings from a binary file and use output as dictionary	Deployed in older fuzzing frameworks	B
EM_17	Using ANGR binary analysis framework to help fuzzer identify possible magic values in the program under test	T-Fuzz	B
EM_18	Using hexadecimal editors, such as 010Editor, to extract interesting byte sequences (tokens) from input file format	AFLSmart	B

[continued ...]

[a] EM = Extraction Method
[b] Source / Binary

https://doi.org/10.18489/sacj.v37i2.21430

Table 2: Continued...

EM#	Method / Strategy	Fuzzing Tool	S/B
EM_19	Use AFL_LLVM_DICT2FILE = /absolute/path/file.txt and during compilation all constant string compare parameters will be written to this file and later used as fuzzing dictionary.	AFL++	S
EM_20	Using clang-sdict that performs a front-end pass on source code collecting constant string tokens used in potentially data-dependent control flow.	Statistical Evaluation	S
EM_21	Use CodeQL to extract dictionary for each seed instead of overall dictionary from all seeds.	CDFUZZ	S

ies (Böttinger et al., 2018; Karamcheti et al., 2018; Kukucka et al., 2022) deployed this strategy. The dictionary generated in this way may contain less useful strings such as comments in the source code. In cases where source code is not available, another approach is to scrape dictionary tokens from the application binary code (Kukucka et al., 2022). In addition to manual methods, strategies for extracting fuzzing dictionary tokens range from scraping constant strings from input files, to using binary analysis frameworks such as ANGR[1], 010Editor[2], and Linux strings[3] tool.

Auto-token detection strategies are used in fuzzers such as AFL, libFuzzer, AFL++ (Fioraldi, Maier et al., 2020) and libAFL (Fioraldi et al., 2022). Whereas AFL auto-detects tokens during deterministic bitflip mutation, other fuzzers auto-detect tokens from instructions and comparison functions (Fioraldi, Maier et al., 2020).

In an experimental framework called Orthrus, Shastry et al. (2017) used static syntactic analysis to automatically identify input fragments from the target program abstract syntax trees (AST). These input fragments can be added to a fuzzing dictionary. Furthermore, the framework used static semantic analysis to identify conjunctions of input fragments from the target PUT control flow graph (CFG). These were also added to the fuzzing dictionary.

A study conducted by Kim et al. (2021) outlined a detailed algorithm that first extracts all readable string constants from the data section of the binary input file, then identifies and constructs a list of addresses of the extracted strings. This is followed by a series of steps that include reference mining (identifying instructions that reference a particular address), fine-grained instructions (identifying usage of referenced instruction), and library function analysis (checking if referenced string was used in custom defined functions). Based on the library function analysis, all strings used in the custom function are added to a fuzzing dictionary.

Another approach is to instrument comparison functions like *strcmp* in the target program to extract and add tokens to a dictionary. This approach is deployed in fuzzing tools such as AFL++, libFuzzer, Entropic (Böhme & Manès, 2020) and Honggfuzz.

A couple of other studies (Ebrahim et al., 2022; Li et al., 2020; Wu et al., 2025) proposed

[1] https://github.com/angr/angr
[2] https://www.sweetscape.com/010editor/
[3] https://www.linux.org/docs/man1/strings.html

more advanced strategies and algorithms to automate token extraction. The FuzzingDriver (Ebrahim et al., 2022) is one of the latest dictionary token generation tools for coverage-based grey box fuzzers. It uses CodeQL[4] queries that can be executed before the fuzzing process begins and therefore, does not add any overhead to the fuzzing process. An evaluation of the FuzzingDriver extracted dictionaries, against Google dictionaries, shows improved code coverage.

In another study, the CDFUZZ (Wu et al., 2025) fuzzer used a similar approach to FuzzingDriver, however, instead of extracting overall dictionary from all seeds, a customised dictionary can be extracted for each seed. Compared to FuzzingDriver extracted dictionaries, the CDFUZZ dictionaries showed up to 18.4% improvement in code coverage.

In another experimental study, Mathis et al. (2020) proposed the LFuzzer technique that extends dynamic tainting to track explicit data flows, as well as taint implicitly converted data while minimizing associated path explosions. The technique introduces a mechanism that makes it possible to infer a set of tokens which can be added to the fuzzing dictionary. In addition, the implementation of the proposed technique makes it possible to infer seed input from source code of the target program under test.

Based on the concept of guard tokens, an experimental study conducted by Li et al. (2020) proposed and implemented a fuzzer called GTFuzz, which may be used to extract Guard Tokens (GTs) to direct fuzzing towards specific target locations. Its evaluation outperforms state-of-the-art fuzzers in reaching target location and exposing bugs.

Other state of the art fuzzers (Fioraldi, Maier et al., 2020; Fioraldi et al., 2022; Serebryany, 2015; Zalewski, 2013) deploy different mechanisms to learn input tokens during the fuzzing process. Whereas AFL (Zalewski, 2013) implementation supports user-specified dictionaries, the fuzzer also provides mechanisms to auto-detect dictionary tokens during bitflip operator mutation, in the deterministic stage. This is achieved by looking for groups of bits that, when mutated, always produce the same code coverage (Fioraldi et al., 2023). This may indicate that they are a significant part of a magic byte/value. Once these values are detected, AFL proceeds in the havoc stage to mutate the input test cases by replacing and inserting tokens from the user-specified and the auto-generated list of tokens. Some fuzzing frameworks extract tokens at compilation or instrumentation time (Fioraldi et al., 2022).

The libAFL framework (Fioraldi et al., 2022) proposed an auto tokens technique that can only be used by instrumenting the PUT with a link-time optimisation (LTO) pass. The pass extracts tokens from comparison instructions and functions and encodes them. A libAFL based fuzzer can then extract these tokens and add them to a fuzzing dictionary.

Sections 3.1 and 3.2 highlighted the impact and benefits of dictionary-based mutation, and the section that follows summarises its observed limitations.

[4] https://codeql.github.com/

3.3 Limitations of dictionary-based mutational fuzzing (RQ3)

The paper on AFLSmart (Pham et al., 2021) notes that both dictionary-based and taint-based mutation approaches fail to address the problem of how to mutate a high-level representation of an input file, such as an abstract syntax tree (AST), rather than its bit-level representation. Consequently, a strategy combining both bit-level and chunk-level mutations was proposed.

The paper on WEIZZ fuzzing tool (Fioraldi, D'Elia & Coppa, 2020) observes that comparison patterns such as magic values/bytes and checksum are difficult to overcome through random and blind byte-level mutations. While format-specific dictionaries may help with magic values/bytes, the fuzzer still needs to determine where to insert such values when applying a dictionary mutation operator.

In a study by You et al. (2019), it was observed that while some sanity checks can be satisfied by inserting dictionary tokens during random mutation, other sanity checks are very difficult to resolve using this approach. In this regard, alternative methods such as gradient mutation (P. Chen & Chen, 2018) were recommended. The study further proposes a seedless mutational fuzzing method that aims to mitigate the shortcomings of other approaches, such as dictionary mutation and gradient mutation, by generating input test cases from scratch without any seeds. This approach was evaluated, and it demonstrated effectiveness in resolving some but not all sanity check comparisons.

In an experimental study Even-Mendoza et al. (2023) developed a coverage-based mutational fuzzer for *C* compilers and parsers, it was reported that the use of fuzzing dictionaries could be effective in preserving static validity of mutated code. However, dictionary-based mutational fuzzing for strongly typed languages produces a high rate of invalid programs. In this regard, generational or grammar-based fuzzing and hybrid fuzzing approaches are more appropriate for language compilers.

Whereas mutational fuzzing tools such as AFL have demonstrated the benefits of dictionary-based mutation when dealing with path constraints, this approach fails to address verbose input files (Pham et al., 2021).

Also, the impact of dictionary-based mutation on fuzzing performance may be confounded by other optimisation parameters such as operator selection and scheduling. Section 3.4 provides an overview of these fuzzing parameters.

3.4 Optimisation of dictionary-based mutation operator selection and scheduling (RQ4)

On each fuzz iteration, a mutational fuzzer uses a mutation scheduler to select operators from a predefined set. Typically, this set includes dictionary-based mutation operators such as *Insert* and *Overwrite* operators. Instead of directly returning a mutation operator, the mutation scheduler yields a probability distribution of the number of interesting test cases generated by each predefined operator. During the havoc stage, the fuzzer prioritises operator selection following this distribution (Lyu et al., 2019). That is, operators that consistently generate input test cases that increase code coverage are given higher priority than those with minimal

impact on code coverage. Many baseline fuzzers, including AFL and libFuzzer, assume a uniform probability distribution, and they grant each operator an equal chance to be randomly selected for the next mutation. Several studies have focused on the optimisation of mutation scheduling to improve fuzzing efficiency and effectiveness. A few of these studies are briefly summarised in this section.

The MOPT fuzzer (Lyu et al., 2019) proposed a customised particles swarm optimisation algorithm to establish an optimal selection probability distribution of operators. The evaluation of MOPT showed up to 170% improvement in finding new vulnerabilities than baseline AFL.

Another experimental study (Karamcheti et al., 2018) proposed a machine learning approach and used a Thompson Sampling bandit-based optimisation algorithm to adaptively learn the probability distribution over mutation operators. The algorithm was implemented on AFL fuzzer, and its evaluation demonstrated higher code coverage than baseline AFL.

Table 3 provides a summary of some of the commonly used fuzzing tools and notes the dictionary extraction method used and mutation operator distribution proposed or adopted. Moreover, we note the baseline fuzzer and the fuzzer type: Coverage-based Greybox Fuzzer (CGF), Directed Greybox Fuzzer (DGF) and Regression Greybox Fuzzer (RGF).

Notwithstanding some evidence (Lyu et al., 2019) that mutation operator selection and scheduling can significantly improve fuzzing efficiency and effectiveness, 58% of the fuzzers identified in Table 3 (excluding the frameworks such a FuzzingDriver, AFL++, libAFL and CDFUZZ) adapt the random uniform distribution model. The remaining 42% (including MOPT, FIRM-COV, GTFuzz, Deep Reinforcement fuzzing, and AFL with Thompson Sampling) implement improved mutation operator distributions mechanisms.

The MOPT fuzzer proposed a Particle Swarm Optimisation (PSO)-based algorithm that evaluates the efficiency of candidate mutation operators and adjusts their selection probability towards the optimal distribution. This approach was adapted in other fuzzing frameworks, such as AFL++ and FIRM_COV, and the results showed general improvement in fuzzing performance. In 2018, Böttinger et al. (2018) proposed a reinforcement learning based distribution that resulted in significant improvement compared to random uniform distribution. This approach builds on other studies (Blum et al., 2017; Drozd & Wagner, 2018) that also demonstrated that mutator operator selection can benefit from automated feature engineering based on deep learning techniques. Moreover, Karamcheti et al. (2018) proposed a multi-armed-bandit based algorithm that deployed Thompson Sampling based algorithm to optimise mutation operator selection. However, the evaluation of different mutation operator distributions focused on overall performance and not necessarily specific on scheduling of dictionary-based mutation operations. As shown in Table 3, only AFL and GTFuzz fuzzing tools made a partial attempt on how to prioritise and schedule dictionary-based mutations.

https://doi.org/10.18489/sacj.v37i2.21430

Table 3: Summary of Mutation Operator Distributions

#	Fuzzing Tool	Type	Baseline Fuzzer	Dictionary Extraction Strategy[a]	Mutation Operator Distribution	Specific on dictionary-based mutation operator selection and scheduling
1	AFL	CGF	–	EM_01 EM_02 EM_03	Random Uniform Model	Partial; Deterministic stage
2	libFuzzer	CGF	–	EM_01	Random Uniform Model	No
3	Orthrus	CGF	AFL	EM_05	Same as user-specified baseline fuzzer	No
4	FuzzingDriver	Framework for CGF	Generic: user specified	EM_06	Same as baseline fuzzer	No
5	LFuzzer		PFuzzer	EM_11 EM_12 EM_15	Same as baseline fuzzer	No
6	AFL with Thompson Sampling	CGF	AFL	EM_04	Bernoulli distribution	No
7	FairFuzz	CGF	AFL	EM_13	Same as baseline fuzzer with light mutation masking strategy	No
8	GTFuzz	DGF	AFLGO	EM_07	Improve baseline distribution and applies only dictionary-related mutation on low GT	Partial
9	AFLSmart	CGF	AFL	EM_18	Same as baseline; Applies both bit-level and chunk-level mutation operators	No
10	FIRM-COV	CGF	AFL	EM_10	Particle Swarm Optimisation-based probability distribution;	No
11	MOPT	CGF	Generic: user-specified	Same as baseline fuzzer	Particle Swarm Optimisation-based probability distribution	No
12	Deep Reinforcement fuzzing	CGF	Generic: user-specified	EM_04	Reinforcement Learning-based distribution	No
13	AFL++	Framework for CGF	Generic: extends AFLFast, MOPT	EM_08 EM_14 EM_19	Same as baseline fuzzer plus Custom Mutator API; MOPT Mutator	No
14	REDQUEEN	CGF	–	EM_13	Same as baseline fuzzer	No
15	libAFL	Framework for CGF	Generic: user-specified	EM_09 EM_08 EM_14	Same as baseline fuzzer	No
16	CDFUZZ	Framework for CGF	Generic	EM_21	Same as baseline fuzzer	No

[a] from Table 2

https://doi.org/10.18489/sacj.v37i2.21430

3.5 Alternative augmentation strategies to address the limitations of dictionary-based mutational fuzzing (RQ5)

According to Gan et al. (2018) the core questions to consider when fuzzing is where to mutate and what to mutate. Regarding magic bytes and checksum values, these fundamental questions are most relevant, and fuzzing tools have used different techniques to address them. Vuzzer (Rawat et al., 2017) uses data flow and control-flow information to infer bytes to mutate, and Taintscope (T. Wang et al., 2010) uses taint-analysis to identify and fix checksum bytes. On the question of what value to use for mutation, Vuzzer employs dynamic taint analysis to infer interesting values and LAF-intel implements mechanisms to split long string or constant comparison and this enables the fuzzer to find matching mutation values. However, dynamic taint analysis is computationally expensive. Alternative strategies to address path constraints are used, and they include symbolic execution. While symbolic execution is sound and complete, it is not scalable to large programs, consequently some studies (P. Chen & Chen, 2018) have proposed techniques to address the limitations of symbolic execution.

A study by P. Chen and Chen (2018) proposed a mutational fuzzing tool called Angora that solved path constraints with symbolic execution. Angora leverages techniques such as scalable byte-level taint-tracking, context-sensitive branch count and input length exploration. To resolve path constraints, Angora avoids symbolic execution, and instead deploys a search-based gradient descent algorithm. The proposed strategy was evaluated on the LAVA synthetic bugs and demonstrated improved code and bug coverage.

4 GAPS IDENTIFIED AND FUTURE WORKS

Studies show anecdotal evidence that dictionary-based mutation approximates the syntactic structure of input test cases, and therefore, improves validity of generated input test cases. However, there is limited empirical evidence to quantify the proportion of any observed subsequent improvement in fuzzing efficiency that can be attributed to the dictionary-based mutation strategy. Such improvement may be a result of other confounding factors and optimisation strategies such as seed prioritization, power scheduling and mutation operator scheduling. Whereas fuzzing effectiveness and efficiency are widely used evaluation metrics, the random nature of fuzzing makes it difficult to attribute any improvement to one optimisation strategy. Based on the literature reviewed, very few studies report on adequate statistical methods to evaluate the impact of these different optimisation strategies. In the case of mutation operator scheduling, most studies report on general improvement resulting from the proposed mutation operator selection algorithms such as particle swarm optimisation-based algorithms and reinforcement learning based algorithms. In this regard, it is not clear what proportion of the improvement may be attributable to dictionary-based mutation operators.

To address the evidence gap, we therefore, recommend more empirical studies that are supported by sound statistical evaluation. These studies would seek to identify and assess the interplay between different optimisation strategies despite the random nature of the fuzzing

process. Some studies (Shastry, 2018; Wu et al., 2025) have recommended non-parametric statistical test methods, such as the non-parametric Mann–Whitney U test and Vargha (MacFarland & Yates, 2016) and Delaney's A12 statistic (Arcuri & Briand, 2014), which may be adopted and used to evaluate the impact of different optimisation strategies while accounting for the random nature of fuzzing.

5 CONCLUSION

In this study, we conducted a systematic review on the impact of dictionary-based mutation strategy on the Greybox fuzz testing process. To the best of our knowledge, this work constitutes the first systematic review addressing this topic. The reviewed literature agrees on anecdotal evidence that the dictionary-based mutation strategy preserves the syntactic structure of mutated test cases and therefore, improves the validity of generated input test cases. Subsequently, more valid input tests cases contribute to improved fuzzing performance. In the context of other optimisation strategies such as mutation operator scheduling, we further observe the paucity of empirical evidence to determine what proportion of the observed improvement can be attributed to the dictionary-based mutation strategy alone. We, therefore, conclude that there is a need to conduct more experimental studies that perform statistically sound evaluation on the impact of dictionary-based mutation strategy, considering the inherent random nature of fuzzing and accounting for other optimisation strategies.

References

Arcuri, A., & Briand, L. (2014). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, *24*(3), 219–250. https://doi.org/10.1002/STVR.1486

Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., & Holz, T. (2019). REDQUEEN: Fuzzing with input-to-state correspondence. *Proceedings 2019 Network and Distributed System Security Symposium.*, *19*, 1–15. https://doi.org/10.14722/ndss.2019.23371

Blum, W., Rajpal, M., & Singh, R. (2017). Not all bytes are equal: Neural byte sieve for fuzzing [Accessed 6 December 2025]. https://www.microsoft.com/en-us/research/publication/not-all-bytes-are-equal-neural-byte-sieve-for-fuzzing/

Böhme, M., & Manès, V. J. (2020). Boosting fuzzer efficiency: An information theoretic perspective. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 678–689. https://doi.org/10.1145/3368089.3409748

Böhme, M., Pham, V., & Roychoudhury, A. (2019). Coverage-based Greybox fuzzing as Markov chain. *IEEE Transactions on Software Engineering*, *45*(5), 489–506. https://doi.org/10.1109/TSE.2017.2785841

Böhme, M., Pham, V., Nguyen, M., & Roychoudhury, A. (2017). Directed Greybox fuzzing. *CCS '17: 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2329–2344. https://doi.org/10.1145/3133956.3134020

Böttinger, K., Godefroid, P., & Singh, R. (2018). Deep reinforcement fuzzing. *2018 IEEE Security and Privacy Workshops (SPW)*, 116–122. https://doi.org/10.1109/SPW.2018.00026

Boutell, T. (1997). *PNG (Portable Network Graphics) specification Version 1.0*. https://doi.org/10.17487/RFC2083

Bundt, J., Fasano, A., Dolan-Gavitt, B., Robertson, W., & Leek, T. (2021). Evaluating synthetic bugs. *ASIA CCS 2021 - Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, *15*, 716–730. https://doi.org/10.1145/3433210.3453096

Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Communications of the ACM*, *56*(2), 82–90. https://doi.org/10.1145/2408776.2408795

Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., & Liu, Y. (2018). Hawkeye: Towards a desired directed Grey-Box fuzzer. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2095–2108. https://doi.org/10.1145/3243734.3243849

Chen, P., & Chen, H. (2018). Angora: Efficient fuzzing by principled search. *2018 IEEE Symposium on Security and Privacy, SP2018*, 711–725. https://doi.org/10.1109/SP.2018.00046

Chen, Y., Li, P., Xu, J., Guo, S., Zhou, R., Zhang, Y., Wei, T., & Lu, L. (2020). SAVIOR: Towards bug-driven hybrid testing. *2020 IEEE Symposium on Security and Privacy*, 1580–1588. https://doi.org/10.1109/SP40000.2020.00002

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In C. Ramakrishnan & J. Rehof (Eds.), *Tools and algorithms for the construction and analysis of systems. TACAS 2008. Lecture notes in computer science* (pp. 337–340, Vol. 4963 LNCS). Springer-Verlag Berlin Heidelberg. https://doi.org/10.1007/978-3-540-78800-3_24

Ding, Z. Y., & Goues, C. L. (2021). An empirical study of OSS-Fuzz bugs. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR 2021*, 131–142. https://doi.org/10.1109/MSR52588.2021.00026

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., & Whelan, R. (2016). LAVA: Large-scale automated vulnerability addition. *2016 IEEE Symposium on Security and Privacy, SP 2016*, 110–121. https://doi.org/10.1109/SP.2016.15

Drozd, W., & Wagner, M. D. (2018). FuzzerGym: A competitive framework for fuzzing and learning. *ArXiv*. https://doi.org/10.48550/arXiv.1807.07490

Eberlein, M., Noller, Y., Vogel, T., & Grunske, L. (2020). Evolutionary grammar-based fuzzing. In A. Aleti & A. Panichella (Eds.), *Search-based software engineering* (pp. 105–120). Springer International Publishing. https://doi.org/10.1007/978-3-030-59762-7_8

Ebrahim, A. A., Hazhirpasand, M., Nierstrasz, O., & Ghafari, M. (2022). FuzzingDriver: The missing dictionary to increase code coverage in fuzzers. *2022 IEEE International Confer-*

ence on Software Analysis, Evolution and Reengineering (SANER), 268–272. https://doi.org/10.1109/SANER53432.2022.00042

Even-Mendoza, K., Sharma, A., Donaldson, A. F., & Cadar, C. (2023). GrayC: Greybox fuzzing of compilers and analysers for C. *ISSTA 2023 : Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1219–1231. https://doi.org/10.1145/3597926.3598130

Fioraldi, A., D'Elia, D. C., & Coppa, E. (2020). WEIZZ: Automatic grey-box fuzzing for structured binary formats. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1–13. https://doi.org/10.1145/3395363.3397372

Fioraldi, A., Maier, D., Eißfeldt, H., & Heuse, M. (2020). AFL++ : Combining incremental steps of fuzzing research. *Proceedings of the 14th USENIX Workshop on Offensive Technologies*. https://dl.acm.org/doi/proceedings/10.5555/3488877

Fioraldi, A., Maier, D. C., Zhang, D., & Balzarotti, D. (2022). LibAFL: A framework to build modular and reusable fuzzers. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, 1051–1065. https://doi.org/10.1145/3548606.3560602

Fioraldi, A., Mantovani, A., Maier, D., & Balzarotti, D. (2023). Dissecting American Fuzzy Lop: A FuzzBench evaluation. *ACM Transactions on Software Engineering and Methodology*, *32*(2), 52. https://doi.org/10.1145/3580596

Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., & Chen, Z. (2018). CollAFL: Path sensitive fuzzing. *2018 IEEE Symposium on Security and Privacy (SP)*, *SP2018*, 679–696. https://doi.org/10.1109/SP.2018.00040

Ghaffarian, S. M., & Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys*, *50*(4), 1–36. https://doi.org/10.1145/3092566

Godefroid, P. (2020). Fuzzing: Hack, art, and science. *Communications of the ACM*, *63*(2), 70–76. https://doi.org/10.1145/3363824

Godefroid, P., Kiezun, A., & Levin, M. Y. (2008). Grammar-based whitebox fuzzing. *ACM SIGPLAN Notices*, *43*(6), 206–215. https://doi.org/10.1145/1379022.1375607

Google. (2025). *Google fuzzing dictionaries* [Accessed 18 November 2025]. https://github.com/google/fuzzing/tree/master/dictionaries

Halfond, W. G. J., Choudhary, S. R., & Orso, A. (2011). Improving penetration testing through static and dynamic analysis. *Software Testing, Verification and Reliability*, *21*(3), 195–214. https://doi.org/10.1002/stvr.450

IEEE. (1990). *IEEE standard glossary of software engineering terminology*. https://doi.org/10.1109/IEEESTD.1990.101064

Karamcheti, S., Mann, G., & Rosenberg, D. (2018). Adaptive grey-box fuzz-testing with Thompson sampling. *Proceedings of the ACM Conference on Computer and Communications Security*, 37–47. https://doi.org/10.1145/3270101.3270108

Kim, J., Yu, J., Kim, H., Rustamov, F., & Yun, J. (2021). FIRM-COV: High-coverage Greybox fuzzing for IoT firmware via optimized process emulation. *IEEE Access*, *9*, 101627–101642. https://doi.org/10.1109/ACCESS.2021.3097807

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385–394. https://doi.org/10.1145/360248.360252

Kitchenham, B. (2007). *Guidelines for performing systematic literature reviews in software engineering* (tech. rep.). Department of Computer Science, University of Durham, Durham, UK. https://www.researchgate.net/publication/302924724

Kukucka, J., Pina, L., Ammann, P., & Bell, J. (2022). CONFETTI: Amplifying concolic guidance for fuzzers. *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*, 438–450. https://doi.org/10.1145/3510003.3510628

Li, R., Liang, H. L., Liu, L., Ma, X., Qu, R., Yan, J., & Zhang, J. (2020). GTFuzz: Guard token directed Grey-Box fuzzing. *Proceedings of 25th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 160–170. https://doi.org/10.1109/PRDC50213.2020.00027

Liang, H., Pei, X., Jia, X., Shen, W., & Zhang, J. (2018). Fuzzing: State of the art. *IEEE Transactions on Reliability*, *67*(3), 1199–1218. https://doi.org/10.1109/TR.2018.2834476

Liu, B., Shi, L., Cai, Z., & Li, M. (2012). Software vulnerability discovery techniques: A survey. *Proceedings of the 2012 Fourth International Conference on Multimedia Information Networking and Security (MINES '12)*, 152–156. https://doi.org/10.1109/MINES.2012.202

Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W. H., Song, Y., & Beyah, R. (2019). MOPT: Optimized mutation scheduling for fuzzers. *Proceedings of the 28th USENIX Security Symposium*, 1949–1966. https://dl.acm.org/doi/10.5555/3361338.3361473

MacFarland, T. W., & Yates, J. M. (2016). *Introduction to nonparametric statistics for the biological sciences using R*. Springer International Publishing. https://doi.org/10.1007/978-3-319-30634-6

Mathis, B., Gopinath, R., & Zeller, A. (2020). Learning input tokens for effective fuzzing. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*, 27–37. https://doi.org/10.1145/3395363.3397348

Metzman, J., Szekeres, L., Simon, L., Sprabery, R., & Arya, A. (2021). FuzzBench: An open fuzzer benchmarking platform and service. *Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, *21*, 1393–1403. https://doi.org/10.1145/3468264.3473932

Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, *33*(12), 32–44. https://doi.org/10.1145/96267.96279

Nagy, S., & Hicks, M. (2019). Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, *2019-May*, 787–802. https://doi.org/10.1109/SP.2019.00069

Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., Shamseer, L., Tetzlaff, J. M., Akl, E. A., Brennan, S. E., Chou, R., Glanville, J., Grimshaw, J. M., Hróbjartsson, A., Lalu, M. M., Li, T., Loder, E. W., Mayo-Wilson, E., McDonald, S., … Moher, D. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *British Medical Journal*, *372*. https://doi.org/10.1136/bmj.n71

Pham, V. T., Böhme, M., Santosa, A. E., Caciulescu, A. R., & Roychoudhury, A. (2021). Smart Greybox fuzzing. *IEEE Transactions on Software Engineering*, *47*(9), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681

Raharjana, I. K., Siahaan, D., & Fatichah, C. (2021). User stories and natural language processing: A systematic literature review. *IEEE Access*, *9*, 53811–53826. https://doi.org/10.1109/ACCESS.2021.3070606

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., & Bos, H. (2017). Vuzzer: Application-aware evolutionary fuzzing. *Network and Distributed System Security Symposium (NDSS)*, 1–14. https://doi.org/10.14722/ndss.2017.23404

Serebryany, K. (2015). *libFuzzer* [Accessed 18 November 2025]. https://llvm.org/docs/LibFuzzer.html

Shastry, B. (2018). *Statistical evaluation of a fuzzing dictionary* (tech. rep.). https://bshastry.github.io/2018/10/01/Evaluating-Dictionary-For-Fuzzing.html

Shastry, B., Leutner, M., Fiebig, T., Thimmaraju, K., Yamaguchi, F., Rieck, K., Schmid, S., Seifert, J.-P., & Feldmann, A. (2017). Static program analysis as a fuzzing aid. *Research in Attacks, Intrusions, and Defenses*, 26–47. https://doi.org/10.1007/978-3-319-66332-6_2

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., & Vigna, G. (2016). Driller: Augmenting fuzzing through selective symbolic execution. *Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2016.23368

Swiecki, R., & Grobert, F. (2025). *honggfuzz: Security oriented software fuzzer* [Accessed 18 November 2025]. https://github.com/google/honggfuzz

Wang, J., Chen, B., Wei, L., & Liu, Y. (2019). Superion: Grammar-aware Greybox fuzzing. *Proceedings – International Conference on Software Engineering*, *2019-May*, 724–735. https://doi.org/10.1109/ICSE.2019.00081

Wang, P., Zhou, X., Yue, T., Lin, P., Liu, Y., & Lu, K. (2024). The progress, challenges, and perspectives of directed Greybox fuzzing. *Software Testing, Verification and Reliability*, *34*(2), e1869. https://doi.org/10.1002/stvr.1869

Wang, T., Wei, T., Gu, G., & Zou, W. (2010). TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. *2010 IEEE Symposium on Security and Privacy*, 497–512. https://doi.org/10.1109/SP.2010.37

Wu, M., Xiang, J., Chen, K., Di, P., Tan, S. H., Cui, H., & Zhang, Y. (2025). Tumbling down the rabbit hole: How do assisting exploration strategies facilitate Grey-Box fuzzing? *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2036–2048. https://doi.org/10.1109/ICSE55347.2025.00044

Xie, Y., Naik, M., Hackett, B., & Aiken, A. (2005). Soundness and its role in bug detection systems. *BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*. https://www.cs.umd.edu/~pugh/BugWorkshop05/papers/12-xie.pdf

You, W., Liu, X., Ma, S., Perry, D., Zhang, X., & Liang, B. (2019). SLF: Fuzzing without valid seed inputs. *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*, 712–723. https://doi.org/10.1109/ICSE.2019.00080

Yun, I., Lee, S., Xu, M., Jang, Y., & Kim, T. (2018). QSYM : A practical concolic execution engine tailored for hybrid fuzzing. *Proceedings of the 27th USENIX Security Symposium*, 745–761. https://dl.acm.org/doi/10.5555/3277203.3277260

Zalewski, M. (2013). *AFL: American Fuzzy Lop – a security-oriented fuzzer* [Accessed 18 November 2025]. https://github.com/google/AFL

Zalewski, M. (2015). *afl-fuzz: making up grammar with a dictionary in hand* [Accessed 18 November 2025]. https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html

Zeller, A., Gopinath, R., Böhme, M., Fraser, G., & Holler, C. (2024). *The fuzzing book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/

Zhu, S., Wang, J., Sun, J., Yang, J., Lin, X., Wang, T., Zhang, L., & Cheng, P. (2024). Better pay attention whilst fuzzing. *IEEE Transactions on Software Engineering, 50*(2), 190–208. https://doi.org/10.1109/TSE.2023.3338129

Zhu, X., & Böhme, M. (2021). Regression Greybox fuzzing. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2169–2182. https://doi.org/10.1145/3460120.3484596