Tampere University

Niko Lappalainen

COVERAGE-GUIDED FUZZING OF GRPC INTERFACE

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Marko Helenius
Billy Brumley
January 2021

ABSTRACT

Niko Lappalainen: Coverage-guided fuzzing of gRPC interface
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
January 2021

Fuzz testing has emerged as a cost-effective method of finding security issues in many real-world targets. The software company M-Files Inc. wanted to incorporate fuzz testing to harden the security of their product M-Files Server. The newly implemented gRPC API was set as the target interface to be fuzzed. This thesis was requested to find a suitable fuzzing tool, and to verify that the tool could find and report issues. Another objective of this thesis was to determine a criterion for stopping fuzzing when adequate testing coverage has been achieved without having to run the fuzzer perpetually.

To select a suitable fuzzing tool, some requirements had to be defined. Requirements and selection criteria were set based on the properties of the M-Files system as well as the target interface. Next, various fuzzing tool options were gathered from different sources. These options were validated based on the set requirements to select a short list of tools that could be analysed more closely. The suitable tool was selected from these based on their ease of integration and suspected performance. The coverage-guided WinAFL was evaluated as the most suitable from the considered options.

The selected fuzzing tool was used to test M-Files Server in order to record its results. The fuzzer was able to find an actual security-critical issue, which verifies the fuzzer's ability to find and report issues. To define a stopping criterion, the fuzzer's cumulative path coverage over time was analysed. It was decided that the time interval between found distinct code paths would be used to determine when a fuzzing run should be stopped. The intervals observed in the results were studied and a maximum interval value was suggested based on when the fuzzing efficacy was observed to noticeably decrease.

Keywords: fuzzing, testing, security, coverage, API, AFL

TIIVISTELMÄ

Niko Lappalainen: gRPC rajapinnan kattavuusohjattu fuzz-testaus
Diplomityö
Tampereen yliopisto
Tietotekniikan koulutusohjelma
Tammikuu 2021

Fuzz-testaus on noussut esiin tehokkaana tapana löytää tietoturvahaavoittuvuuksia oikeista ohjelmista. Ohjelmistokehitysyhtiö M-Files OY päätti ottaa fuzz-testauksen käyttöön varmistaakseen tuotteensa M-Files Serverin tietoturvallisuutta. Fuzzauksen kohteeksi valikoitui hiljattain kehitetty gRPC rajapinta, jota monet M-Files järjestelmän komponentit käyttävät. Tämän diplomityön tarkoituksena oli löytää M-Filesin testaukseen soveltuva fuzzaustyökalu ja varmistaa työkalun kyky löytää virheitä M-Files Serveristä. M-Files OY halusi työn myös määrittelevän säännön, jonka mukaan fuzzaus voitaisiin lopettaa sen saavutettua riittävän kattavuuden.

Määriteltyyn käyttötarkoitukseen soveltuvan fuzzaustyökalun löytämiseksi määriteltiin ensin vaatimuksia fuzzerille. Seuraavaksi kerättiin lista työkaluvaihtoehtoja eri lähteistä ja verrattiin niitä vaatimuksiin. Vertauksen perusteella valittiin lyhyt lista vaihtoehtoja, joita tutkittiin tarkemmin. Näistä vaihtoehdoista pyrittiin valitsemaan paras työkalu oletetun integroimisvaikeuden ja tehokkuuden perusteella. Parhaaksi vaihtoehdoksi valiintui kattavuusohjattu WinAFL, joka integroitiin testaamaan M-Files Serveriä.

Valittua työkalua käytettiin M-Files Serverin testaamiseen, jotta sen tuloksia voitaisiin analysoida. Työkalu onnistui löytämään serveristä aidon tietoturvakriittisen vian. Tämä oli yllättävää koska testaus oli dimplomityötä varten rajattu hyvin pieneen osaan rajapinnasta. Löydös kuitenkin todisti fuzzerin kyvyn löytää vikoja M-Files-järjestelmästä. Fuzzerin aikaansaamia tuloksia käytettiin myös pysäyttämissäännön määritykseen. Sopivaksi säännöksi valittiin kahden fuzzerin löytämien eriävien koodipolkujen löytämiseen kuluva väliaika. Fuzzaus voitaisiin lopettaa valitun väliajan ylitettyä. Fuzzauksessa nähdyistä tuloksista katsottiin sopiva väliaika-arvo, jonka jälkeen fuzzaustehokkuus selkeästi laski. Tätä väliaika-arvoa ehdotettiin pysäytyssäännöksi.

Avainsanat: fuzzaus, testaus, tietoturva, kattavuus, API, AFL

PREFACE

I would like to thank Tero Piirainen and Minna Vallius for assigning me this interesting thesis topic. The work had its fair share of difficulties and frustrations, but it was a great learning experience. On that note, I would also like to thank Taneli Uusikorpi, Juha Lepola, and Mikko Rantanen for their technical support and limitless patience so far and continuing while the fuzzer is developed toward production use.

Also, thank you Marko Helenius for guiding me through the writing and course process despite my own confusions. These weird times that we live in surely set their own challenges for all of us.

Tampere, 30th January 2021

Niko Lappalainen

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAMS AND ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

AEQ	All Equivalent
AFL	American fuzzy lop
AIO	All Included in One
ART	adaptive random testing
CI	continuous integration
HTTP/2	Second version of hyper text transfer protocol
IDL	interface definition language
IPC	inter-process communication
RT	random testing
SUT	system under test

1 INTRODUCTION

Security testing is a difficult but necessary part of software development. Any software that exposes an interface to a potentially malicious users must take information security into accord in some extent. Services exposing interfaces to the Internet are at most risk, because they are accessible to a large audience of malicious actors.

Security testing is further made difficult by today's practises of iterative software development. The guiding principle of continuous integration (CI) is to build the software after every change. This includes continuous testing to ensure that the software remains stable. Test automation is used to enable continuous testing without unreasonable human resource costs. It is also important to maintain the security of the software for any versions that have any chance of being delivered to customers. For this purpose, automated security testing is a valuable tool.

Fuzz testing has emerged as a popular tool for automated security testing. In a paper by Miller et al. [2], the term fuzz was used as the name of a random string stream generator program that was used to pipe random text to tested command line programs. In more recent literature [3], the term has expanded to cover any security testing using anomalous input with the purpose of crashing the system under test (SUT). It is therefore a form of random testing (RT) or adaptive random testing (ART) [4] where the program is only considered faulty if it crashes or hangs [3].

M-Files Inc. is a global company offering an enterprise content management solution based on metadata management. Both on-premises and cloud solutions of the M-Files product (M-Files) are offered, and they are used to store customers' business critical information and documents. This information could contain valuable business secrets and its availability and integrity is critical for customers of M-Files Inc. M-Files also offers an extensibility interface, which would enable an attacker who penetrated the system to run arbitrary code on both client and server machines. These factors make M-Files an attractive target for possible attackers.

At the time of writing, M-Files Inc. is working towards replacing the current public and internal interfaces of M-Files with a single public interface that uses gRPC remote procedure call protocol. Unlike the previous solutions where some interfaces are intentionally made obscure, the new gRPC interface would expose all possible functionality of the software to all users who have authenticated to the system. It would also be fully documented and advertised to the general public. This would make it easy for malicious actors

to explore the software for vulnerabilities.

M-Files Inc. decided to look into fuzz testing for the purpose of increasing security testing coverage of the new interface. Three goals were set for this thesis:

- Find a fuzzing tool that is suitable for testing gRPC interfaces.
- Study the ability of fuzz testing to find faults in M-Files.
- Find out how long a gRPC method needs to be fuzzed to achieve adequate coverage.

For the sake of finishing this thesis on schedule, the focus was set to a single method in the gRPC interface. Extending the API coverage was left as a topic of future development. Focus was also set on M-Files Server specifically. Testing of the client software is not considered.

To find a suitable fuzzing tool, some popular open-source options were listed and compared to the limitations set by M-Files. A fuzzer tool was implemented to study its ability to find bugs by measuring coverage. Finally, the coverage of fuzz testing was compared to existing automated M-Files security testing.

2 GRPC

gRPC [5] is an open-source RPC protocol developed by Google. It was designed for efficient communication of microservices, but it now also has other use-cases such as connecting mobile and browser clients to backend services and Internet of things. The protocol uses HTTP/2 [6] to transfer binary encoded messages. The binary encoding used is based on protocol buffers, which is also developed by Google. gRPC has several advantages over some other RPC protocols, such as MSRPC which was previously used by M-Files Server.

Interfaces such as JSON APIs can end up transferring a lot of overhead data over the network due to representing the data as encoded text such as ASCII or UTF. For example, the integer 10 can be represented as a single byte in binary, however it takes 2 bytes to represent the same number in UTF-8. The overhead of text-based communication protocols may be alleviated by using compression. However, compression cannot necessarily be used in some security critical applications where confidential information is transferred. If an attacker could control some part of a compressed and encrypted response message, they could infer contents of the message by measuring response sizes. By using a binary encoding, gRPC attempts to optimize the transferred data. The binary-serialized data can also be compressed, but it would be vulnerable to the same attacks as text-based communications. Even without compression, gRPC has much less overhead. It has also been proven to have good performance in M-Files use.

The optimizations of gRPC are also partly due to HTTP/2. For example, the HTTP/2 protocol uses the HPACK [7] compression algorithm for its header data [6] that it uses by default. The compression is based on maintaining a dynamic list of headers that have been used during the current connection. Once a header field has been used and saved to the list, subsequent calls that have the same header field use the index of the saved value instead. Either the whole header field or just its name is replaced by the index. Header values are only replaced by the index if they exactly match the saved value. This makes HPACK less vulnerable to side-channel attacks than some other compression algorithms. Improved concurrency is another optimization feature that HTTP/2 provides [6]. HTTP/2 allows opening multiple streams so that there are several requests concurrently pending.

Additionally, the stream nature of HTTP/2 enables features such as server data push where the server can send a client data that the client did not explicitly request. Because the client does not need to send polling requests, less bandwidth is used. The session

still must be kept alive using other requests. An example of server push use case would be a client that subscribes to changes in a data object. If the object is changed on the server, the client has to be notified so that it can update its user interface.

The use of HTTP/2 also makes gRPC more compatible with existing network infrastructure compared to some other RPC protocols. For example, most firewalls likely support the protocol by default because HTTP is also used by web browsers. If configured to the same network ports, gRPC traffic will appear very similar to other HTTP requests, such as REST API calls or requests for HTML pages.

2.1 gRPC over HTTP/2

Specification of how gRPC implementations communicate using HTTP/2 is described in the source git repository [8]. A gRPC request consists of request headers that are transferred in the HTTP/2 HEADER and CONTINUATION frames and a length prefixed message that is delivered in one or more DATA frames.

The request metadata headers consist of key-value pairs. The following headers are used:

- *`:method = POST` gRPC methods are delivered as HTTP POST methods.
- *`:scheme = {(http / https)}` Determines wether a secure connection is used.
- *`:path = {service name}/{method name}` Specifies the gRPC service and method names.
- `:authority` Authority portion of the target URI.
- *`te = trailers` Used to detect incompatible proxies.
- `grpc-timeout = {integer value and time unit}` Timeout for the request. Can be omitted implying infinite timeout.
- *`content-type = application/grpc{("+proto" / "+json" / {custom})}` Type of the request message and how it is serialized.
- `grpc-encoding = {compression mechanism}` Compression mechanism of the request message.
- `grpc-accept-encoding = {list of compression mechanisms}` Accepted compression mechanisms of the response message.
- `user-agent` String describing the calling library.
- `grpc-message-type = {name of the proto message in the schema}` Protocol Buffers message name.

The headers marked with an asterisk (*) cannot be omitted from the request. In addition to these headers the protocol allows for custom headers that can be used in the application layer.

The request DATA frame or frames deliver the following binary data:

- *Compressed flag* that indicates whether the message has been compressed. If the flag is set to 1, the `grpc-encoding` header must not be omitted because it determines the compression mechanism. If the flag is 0, the message is not compressed, and the header can be omitted.
- *Message length* as 32 bit unsigned big endian integer.
- *Message* that is a serialized Protocol Buffers message. It can optionally be compressed.

The server will respond the client's request with a response that is also delivered over HTTP/2. The HTTP status header should be 200 indicating a success as long as there was no error on the HTTP/2 layer. The result of the gRPC method itself is defined in the header frames as follows:

- `grpc-status = {gRPC status code}` Status code where e.g. 0 indicates success.
- `grpc-message = {message string}` Status message containing e.g. error description.

The response can also include custom metadata headers defined by the application layer.

The response also includes a response Protocol Buffers message that is similarly prefixed to the request message.

2.2 Protocol Buffers IDL

The interface definition language (IDL) used to define a gRPC interfaces is the *proto* language for Protocol Buffers [9]. The interface methods and the data consumed by them are defined in files with .proto files. Protocol Buffer compiler *protoc* is used to generate source code files which implement the defined interface and the data structures. The proto language has no dependencies to gRPC and it can be used to define any other RPC protocols that are based on protocol buffer serialization.

Protocol buffers [9] is a binary data serialization protocol. The proto language is used to define messages of key-value pairs in hierarchical structures. The message structures are comparable to JSON objects and the library implementations even include features to parse the messages into JSON by default. The Protocol Buffers objects have one or several fields which all have a field number and a type. The field number functions as the key in the key-value pair. Some field types include integers, floating point numbers, booleans, character strings, raw data blocks, and another protocol buffer message. Any field can also be defined as repeated which means that the field can repeat any number of times with each repeated field having a distinct value.

In order to study fuzzing Protocol Buffers messages, knowing the binary structure of encoded messages [9] is required. Only the proto3 version is described here, because that is used by M-Files Server. The encoded message is an array of the encoded key-value pairs where the value is the field value. Different field type encodings are listed in

Table 2.1. *Protocol Buffers field type encodings [9].*

Encoding	Meaning	Data types
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

table 2.1. A varint is a way of encoding a variable-length integer. The most significant bit is set as 0 for the last byte in the varint and 1 for every other byte. The integer value is represented by the groups of 7 bits where the first group is the least significant. In addition, the signed integer types `sint32` and `sint64` use ZigZag encoding to reduce the encoded size of negative values. The encoded varint value of a negative integer n is $|n| \times 2 - 1$ and $n \times 2$ for a positive integer. The fixed-sized values do not have a specific encoding and are read as blocks of data of the specified type.

Length delimited encoding is used for strings, byte arrays, embedded messages, and repeated fields. The encoded value starts with a varint that specifies the length of the actual value. The varint is followed by a block of bytes of the specified length. The block could be for example a UTF-8 string. If more data follows the data block, that is interpreted as the next field.

The message itself is encoded as an array of encoded key-value pairs. The key is encoded as a varint of the field number that is left shifted by 3. The 3 least significant bits in the key varint specify the encoding type of the field value. The encoded field value immediately follows the key varint. Even if the field specification is unknown to the application that reads the message, the application knows the length of every field from the encoding type value in the key varint. This allows applications to ignore unknown fields, which allows version compatibility if the API specification is changed. Not all specified fields have to be present in a message and some fields may be repeated even if the field is not a repeated type field.

The proto service definitions contain of methods that are grouped together in services. A method takes a protocol buffers message as an input and it outputs another protocol buffers message. The Services are intended to be used for routing in the implementing RPC protocols. As an example of a proto file, a partial M-Files login method definition is given in program 2.1.

An encoded protocol buffers message does not contain the full metadata about the containing fields. It contains enough information for an application to ignore fields that it does not recognize, but not the exact types of fields or their names. The name of the message type is also not encoded. The application parsing the message must know its type and definition before decoding it. gRPC protocol maps each method to a URL. The receiving application reads the URL and looks for the matching method from the definition. The

```
1  syntax = "proto3";
2
3  import "m-files/structs.proto";
4
5  package MFiles;
6
7  // MF_LogIn method request structure.
8  message LogInRequest {
9
10     //!< Description of the client environment.
11     EnvironmentData environment_data = 1;
12
13     //!< Description of the client.
14     ClientData client_data = 2;
15
16     //!< Description of the desired login.
17     LoginData login_data = 3;
18
19     //!< Authentication parameters.
20     AuthDataClient authentication_data = 4;
21 }
22
23 service IRPCLogin {
24
25     //! Logs in (with authentication).
26     rpc LogIn( LogInRequest )
27             returns ( LogInResponse )
28     {
29         option ( google.api.http ).get = "/grpc/v1/log-in";
30     }
31
32 }
```

Program 2.1. *Simplified proto file for the M-Files login method.*

definition specifies the type of the input message.

3 COVERAGE CRITERIA

Testing coverage is the criterion that is used to define the adequacy of testing [10]. The absence of bugs is not easy to prove, but by systematically designing tests to cover specific areas of the target we gain a higher confidence that those areas do not contain errors. A test is adequate if its successful execution can confirm with reasonable confidence that the target contains no errors. The criteria for testing coverage and adequacy have been well studied [10].

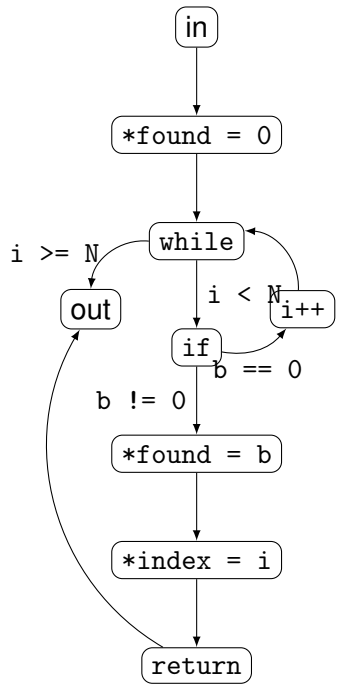
Zhu et al. [10] listed some categorizations for coverage criteria based on the testing approach:

1. *structural testing*: Testing aims to cover the program or specification structure such as unit testing for every method of a class.
2. *fault-based testing*: When testing focuses on finding faults, adequacy criteria must measure the test set's ability to detect faults.
3. *error-based testing*: Test cases should cover commonly error-prone points.

Coverage criteria have more uses than just defining a binary distinction between adequate and inadequate tests. The criteria can be used to measure a degree of adequacy which can have more varied applications [10]. For example, for product managers and system architectures the coverage measurements can be used to assess the dependability of the target. Different targets may have different requirements for testing coverage. Some types of organizations, such as medical or military, may have standardized requirements for testing and testing coverage.

Coverage criteria is also useful for developers and testers. The criteria may be used to specify the number and type of test cases required. The degree of coverage is also useful for many purposes. It can reveal areas of the code that have not been tested sufficiently. Random testing especially benefits from measuring coverage. For example, it can be used to guide the random selection of diverse test cases [11]. This method of improving test case diversity has been proven to improve failure detection of test sets [12][13][14]. Additionally, coverage criteria can be used as a stopping rule for random testing [10] [15]. Also, different randomly selected test cases can be compared using their coverage.

```
1  void Search(int arr[], int key,
2              int *found, int *index)
3  {
4      int i = 0;
5      int b;
6
7      *found = 0;
8
9      while (i < N)
10     {
11         if (b = isabsequal(arr[i], key))
12         {
13             *found = b;
14             *index = i;
15             return;
16         }
17
18         i++;
19     }
20 }
```



Program (3.1) *Search function example.*

Figure 3.1. *Function* Search *and its flow-graph model [17].*

3.1 Structural coverage

Statement coverage criterion [16] measures the number of statements in the SUT that are exercised in the execution. In a flow-graph model [10] [17], statement coverage is defined as the number of graph nodes that are contained int the execution paths of the test set. Example of a flow-graph model is given in figure 3.1. Full statement coverage is often not possible due to unreachable code. Statement coverage is also considered a weak criterion because a fully statement adequate test set may miss some control transfers. [10] Statement coverage can be reduced by grouping consecutive statements without control transfers as segments or blocks [17].

Branch coverage [16] is a slightly stronger criterion compared to statement coverage. It corresponds to the number of traversed control transfers or the edges in the flow-graph model. Because there is at least one edge for every node in the flow-graph model, full branch coverage intuitively also correlates to full statement coverage. Branch coverage adequacy also cannot be achieved if there is dead code in the tested system. [10]

Path coverage criterion requires that all possible combinations of control transfers are executed for full adequacy. In the flow-graph model, this would mean that every path from the start node to the end node are executed. Path coverage is much stronger criterion compared to statement and branch coverage, but it is also too strong in practise. Looping structures in the SUT can lead to infinite number of possible paths. Therefore, it would not be possible to reach full adequacy using a finite test set. [10]

Neither statement, branch nor path coverage criteria are finitely applicable. It is also difficult to define finitely applicable versions of statement or branch coverage because it can be difficult to determine whether a statement or branch is reachable. However, finitely applicable version can be defined for path coverage for example by requiring only simple or elementary paths with some complexity restrictions. The selection of simple and elementary paths averts paths with redundancy and therefore selects only the most important subset of paths. [10]

In addition to criteria that are based on the flow-graph model there are some structural coverage criteria that are based on program text instead [10]. An example for such criterion is the *multiple condition coverage* [18] that requires the test set to cover all possible combinations of values of atomic predicates in every control transfer condition. The atomic predicates of a condition are the predicates that are combined into a boolean expression using logic connectives like *and* and *or* [10].

3.2 Fault-based coverage

Fault-based coverage criteria aim to measure the test sets ability to find faults. One example of fault-based criteria is *error seeding*, where faults are purposefully introduced to the SUT in random locations that are unknown to the tester. The test set is then used to test the target and the number of discovered artificial faults is calculated. The higher the ratio of discovered and total artificial faults is the higher the quality of testing is considered. The seeded faults are assumed to be similar in difficulty of detection to actual faults. [10] Therefore, fault-based coverage criteria should be stronger measures of testing quality than structural coverage criteria.

In addition to evaluating the test set, error seeding can be used to estimate the number of undiscovered bugs. We can assume that the ratio of discovered and total artificial faults r is equal to the ratio of discovered and total actual faults. With that assumption we can estimate the total amount of actual faults to be $f_a = f/r$ where f is the number of actual faults discovered with the test set. [10]

The ability of error seeding to measure the quality of testing is based on how similar the artificial faults are to actual faults in difficulty of detection. According to Zhu et al. [10] it is difficult to manually create artificial faults that are close in similarity to actual faults. They state that artificially seeded errors are typically much easier to discover than actual errors. They say that mutation testing was developed to mitigate that problem by introducing faults to the target more systematically.

Mutation analysis measures a test set's ability to discover bugs by automatically seeding a bug in the target software to create a mutant of the SUT. The mutant is then tested using the test set and the outputs are compared to outputs of the original tested program. If some test case results in a different output on the mutant compared to the original, the mutant is considered to have been killed by the set. In other words, the test set would be able to discover a bug in the code. By repeating the process of creating a mutant and

attempting to kill the mutant the quality of the test set can be measured using the ratio of killed and total mutants. This ratio is called the *mutation score*. The accuracy of mutation score may be affected by the existence of mutants that produce the same output as the original program. It is not possible to decide whether a mutant is equivalent in this way. [10]

The most important issue of mutation analysis is the mutation operations used to mutate the target. A mutation operation can be for example changing a plus operation to a minus. For mutation analysis to effectively measure the adequacy of a test set the mutation operations must be close to real errors that a programmer could make. [10] Therefore, these mutation operations are usually applied on the source code level. The effectiveness of mutation analysis is based on two assumptions. First, the competent-programmer assumption [19] states that programmers normally create code that is close to being correct. Therefore, small modifications to the target should be close to possible real programming errors. Secondly, the coupling effect assumption states that a test set that kills a simple mutation would also kill a complex mutation [10].

Zhu et al. [10] state that mutation analysis is expensive because of the required resources to run mutants and the human resources of analysing mutants for equivalence. With large programs that are written in compiled languages, compilation time is also likely to be very resource demanding. This is slightly alleviated by incremental compilation since mutational operations should affect only single file at a time. The human resource cost can be entirely waived if the purpose of mutation score is to compare the effectiveness of two test sets on the same program. The storage cost of mutants can be alleviated by only storing the delta of the source code, which would be trivial using a version control program. There is no need to store the mutant binary after it has been tested with all available test sets.

3.3 Error-based coverage

Error-Based coverage criteria are based on equivalence class partitioning. Faults in a software have been observed to cluster on specific segments. For example, consider a function that calculates the sum of two unsigned integers x and y. The function produces an incorrect result if the sum causes an integer overflow. The function then has at least two segments:

$$\{x + y \leq UINT\_MAX, x + y > UINT\_MAX\}$$

These segments are behaviourally equivalent. If one test case in a segment results in a faulty result, then all inputs in that segment result in a fault. [10] These segments are also called equivalence classes.

Input partitioning can be made using either software specification of code structure [10]. Partitioning based on specification requires the specification to be formally expressed as tuples of pre- and postconditions. In program-based partitioning inputs that result in the same computation are considered to be equivalent. Computations are the same if the

path coverages are same. Program based equivalence classes are therefore based on structural path coverage.

The objective of Error-based coverage criteria is to choose how many test cases should be selected from each input subdomain and the test case positions in those domains. While path criterion is covered as long as one test case is tested for each equivalence class, error-based criteria additionally require test cases at the class boundaries and adjacent to the boundaries to be covered. [10] These inputs are considered to be prone to errors. Test cases selected in the middle of equivalence classes are intended to discover faulty implementation of input subdomain computation [10]. Test cases at and adjacent to the boundaries test for errors in selecting the input subdomains.

4 FUZZ TESTING

Takanen et al. [3] describe fuzzing as sending anomalous data to a system to crash it. The purpose of it is to find inputs that would cause an error in the system leading to a crash. A crash in the system could be caused by an error such as buffer overflow, which could possibly be exploited by an attacker to take control of the system. At the least, an error that causes the system to crash would enable denial-of-service attacks assuming that the system is an online service. Therefore, it is essential for information security to find and fix faults that result in crashes.

The software quality assurance process has developed to focus on verification and validation. The goal of finding faults has been de-prioritized in favour of validating that the software fulfils its requirement or acceptance criteria. [3] However, this focus away from finding faults is detrimental to software security. Missed security flaws are critical threats to the software and its users. In that sense, fuzz testing is a useful security testing tool because it focuses entirely on finding faults. Fuzzers will even test cases that are virtually impossible in normal use of the software. However, fuzzing alone is not enough to secure a software. The other practices of security development are still required.

4.1 Random and adaptive random testing

For an SUT that has n parameters where each parameter l_i has discrete values from set $V_i, 1 \leq i \leq n$, a test case would be represented as a tuple $c = (x_1, x_2, \ldots, x_n)$ where $x_i \in V_i$. The search space of test cases $D = V_1 \times V_2 \times \ldots \times V_n$ can grow very large when there are multiple parameters, which is typical in current software. [20] Even with few parameters, the search space can grow unmanageably large if the search spaces of the individual parameters are also large. This can be the case, for example, if the parameter is a block of data of variable length such as a file. In those cases, the size of the input space could even be considered to approach infinite.

Because the large size of the test case search space, not all test cases can be reasonably tested. Instead, a finite subset of test cases $C \subset D$ must be selected. An optimal test set would maximize the intersection $C \cap F$ where F represents the inputs that expose faults. The optimal set would also minimize the total amount test cases $|C|$ so that the test set can be tested efficiently. In manual test-case-based testing and unit testing the subset of inputs is selected by the tester. The set can also be selected automatically using strategies such as random testing (RT) where inputs are selected randomly without

any knowledge of the SUT. RT has been described as naive [2] [21] because it is likely to select test sets that are very far from optimal. However RT has been proven to be a cost-effective strategy of finding faults [20].

Adaptive random testing [4] is a proposed improvement for RT. Instead of choosing test cases with uniform probability distribution, ART focuses on the diversity of the test cases. The technique is based on the observation of equivalence classes. That is, because the input space can be divided into contiguous failure and non-failure regions, failure-causing test cases can be found by selecting cases far away from non-failure-causing cases [11]. ART aims to cover all the contiguous regions by evenly spreading the selected cases over the search space. If the distance of similarity can be properly calculated for inputs, the determining factor for the error-based coverage of ART becomes the number of selected test cases. Fuzz testing could be considered a form of random testing where the selected test cases are used to execute the SUT, but the output is not necessarily verified besides checking for crashes or hangs. Some fuzzers could also be considered to use ART, as they use some information collected from the SUT to select diverse test cases [3].

ART techniques have been proven to perform better than pure RT by multiple studies [11] [20]. Arcuri and Briand [21] questioned the results of previous empirical studies. They claimed that the empirical studies used to support the benefits of ART used test applications with too high failure rates in comparison to actual programs. However, another empirical study by Wu et al. [20] that tested real-world applications also supported the benefits of ART. This study found that ART outperformed RT also with relatively low failure rate targets, but the difference was negligible with highly constrained targets.

The test model in the study by Wu et al. [20] considers two kinds of input constraints that the target can have: hard constraints that prevent test case execution, and soft constraints that do not need to be tested. The tool used to select test cases generated random parameters until they fulfilled the constraints. This method of selecting test cases may have affected results because of its inefficiency. Also, the some or all of the constraints would likely not have been considered as actual constraints in security testing context. The programs that were tested by Wu et al. were command utilities and therefore no parameter combination would actually prevent execution. Some input combination would simply stop execution after parameter parsing. Such invalid parameter combinations are actually desired with fuzz testing. Therefore, we can say that the constraints' effect on ART's efficiency is not as notable for fuzz testing.

4.2 Fuzzer architucture

Takanen et al. [3, p. 249] as well as Rathaus and Evron [22, p. 16] describe three components that a fuzzer should implement:

1. Test case selection
2. Delivering inputs to target

3. Target monitoring

Test case selection is the central focus of fuzz testing. By using random or adaptive random methods, the fuzzer produces data that can be input to the SUT. These inputs collectively make a set of test cases. While simpler black-box fuzzers likely discard any test cases that do not result in a fault, more feature rich fuzzers can accrue a reusable set of test cases that it has found interesting [23] [24]. Test cases are selected either purely randomly from the input search space or using methods of ART. Heuristics of selecting interesting test cases have been the topic of several studies [4] [25] [26] [27].

Input delivery of a fuzzer depends of the interface of the target system. Originally fuzzers were used to test command line interfaces [2]. Today they are used to also test servers and libraries [3]. With such targets, an interface between the fuzzer and target may need to be specifically implemented. In some cases, the fuzzer runs as a separate process and communicates to the target using inter-process communication (IPC) [23]. Other fuzzers may need to be linked to the target binaries to create a fuzzing executable [24]. Some fuzzers are capable of directly fuzzing values in the target process' memory, which enables testing specific functions even without access to source code [3, p. 161].

The purpose of *target monitoring* is to find faults in the target system. A fault could be for example a hang, a crash, or a memory violation. Fuzzers can also monitor the target environment to detect issues such as unauthorized file access or network traffic. When a fault is detected, the input resulting is said fault must be saved for later analysis. A fuzzer can also monitor the target for other reasons, such as in order to guide test case selection. This is the case with coverage-guided fuzzers that monitor the executed code paths to genetically mutate distinct test cases.

4.3 Fuzzer classifications

There is a large amount of different fuzzing tools with widely different features and use cases. Manes et al. [28] have collected and compiled a genealogy of different fuzzing tools and studies that cite and refer to each other all the way to the original fuzzer by Miller, Fredriksen and So [2]. The genealogy is depicted in figure 4.1. Different fuzzers can be categorized by their input generation methods and their target interfaces [3]. One common way of classifying fuzzers is by the granularity of the monitored data. Black-box, gray-box and white-box classifications are used by multiple sources [3] [25] [28] and they refer to the level of data collected from the data.

A *black-box* fuzzer has no awareness of the code structure of the SUT and simply randomly selects many test cases very quickly [25]. It only sees the output behaviour of the target on the used inputs. They are simple to implement, but do not typically achieve as high coverage or find as many bugs as white- and gray-box fuzzers [3] [25] [29]. The coverage of black-box fuzzers can be improved by other methods, such as using structural information about expected inputs to generate test cases that have a higher chance of passing some input validations [30].

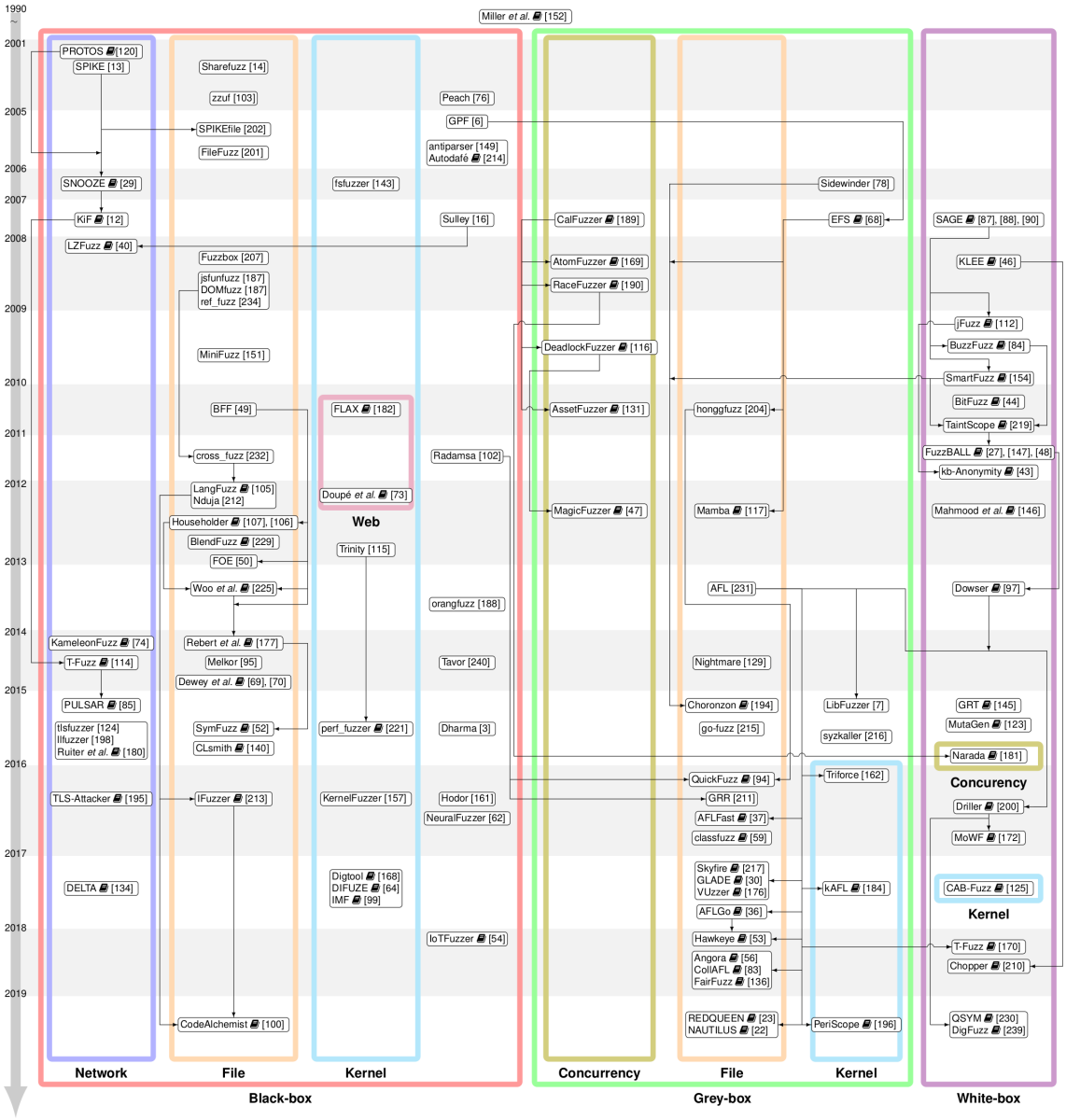


Figure 4.1. *Genealogy of fuzzers. Nodes on the same row have been published on the same year. An arrow from a fuzzer indicates that it was cited, referenced, or its techniques were used by another fuzzer. [28]*

White-box fuzzers like SAGE [29] use symbolic information of the source code to select test cases to deterministically cover every code path [25]. Other white-box methods include dynamic taint analysis [27]. This method aims to analyse how the input data is analysed by the target by marking the memory that is affected by changes in the input as tainted and following how the taint spreads. According to Manès et al. [28] the information gathered by white-box fuzzers at runtime can be used to systematically explore the state space of the target. White-box fuzzing tools like SAGE or KLEE are able to reach high path or multiple condition coverage very efficiently in terms of the number of test cases [29]. However, white-box fuzzers have a higher overhead compared to black-box fuzzers due to the methods used to analyse and monitor the target [28]. Therefore, the test-cases-per-second performance may be lower than that of black-box fuzzers.

Gray-box fuzzers could be considered a compromise between the efficiency of white-box and speed of black-box fuzzers. They use some information about the target's internals to improve their test case selection but they do not analyze the full semantics of the SUT [28]. One method of implementing a gray-box fuzzer is to use instrumentation to gather code coverage information of the SUT during testing. When a test case results in a new code path, that information is used to select the next test case. [25] American fuzzy lop (AFL) [23] and LibFuzzer [24] are both gray-box fuzzers and they take this coverage-guided approach to implementing their evolutionary test case selection algorithms. With both tools the program under test has to be compiled with specific compilers in order to instrument the binary. AFL uses a modified version of GCC and LibFuzzer is a built-in feature of clang.

There are also other basis of fuzzer classification than target analysis granularity. One way to classify fuzzers is based on their test case selection method [3] [28]. A fuzzer that can generate test cases without prior saved test cases is called a *generation-based* fuzzer. They are also called model-based, because they use some model of the expected inputs or executions. The model can be predefined by the tester or automatically inferred through various methods [28].

Another input selection class of fuzzers is *mutation-based* [3] [28]. These fuzzers are given an existing set test cases that are then modified before delivering them to the target. They do not require a model of the input, although they can use some modelled information to improve the mutation. As an example of mutation functions, a fuzzer can flip some bits in the test case data to mutate it.

Fuzzers could also be categorized based on the interfaces that they use to input the fuzzing data and the targets that they focus on. Some fuzzers feed their inputs to the target as command line parameters [2] while other may save the input to a file that the target reads [23]. Fuzzers can also test targets over some network protocol [3]. Many fuzzers are implemented specifically to test a single target and are therefore not made public [22].

A fuzzer that selects test cases without constraints is sometimes called a *dump fuzzer* [3, p. 144]. Alternatively, the fuzzer can be given some constraints for example by defining a grammar for the input. Grammar aware fuzzers are sometimes called *smart fuzzers*. The choice between dumb and smart fuzzers is made according to the SUT and the interface to fuzz. Smart fuzzers are more likely to find deeply nested bugs, but it requires more trust from input parsing than dumb fuzzing. This is because the model can be used to only produce inputs that pass the parsing. On the other hand, the parsing itself would not experience negative testing. AFL and LibFuzzer are examples of dumb fuzzers. On the other hand, Peach is a smart fuzzer and it requires a context definition [30].

4.4 Found bug types

Fuzz testing has proven to be an effective tool for finding exploitable faults in software. Some of the discovered bugs have been listed as trophies on the fuzzing tools' websites [23] [24]. These trophies include security sensitive faults in products such as PHP, OpenSSL, SQLite, and gRPC. The listed trophies include segmentation faults, memory leaks, and hangs. Unlike security scanners that are designed to look for known vulnerabilities in new targets, fuzzers focus on finding new bugs [3].

The oracle problem is a common issue with RT. An oracle is a method of determining whether an observed behaviour in the SUT is consistent with the specification and requirements. Specifically, the purpose of an oracle is to analyse if an observed output is correct for the input that invoked it. An oracle therefore determines the failure and success of a test case. The oracle problem arises when there is no oracle available for the system and implementing one would be non-trivial. Although it is possible to implement an automated testing oracle in circumstances where there are some artefacts of expected behaviour available, in lack of such artefacts an oracle must be created using human effort [31]. The oracle problem is typically not considered in fuzz testing and instead fuzzers tend to focus on behaviour that is clearly faulty regardless of the input [3].

Crashes and hangs are examples of behaviour that is clearly undesired with all input. Many other types of errors can also be detected by fuzzers, but their detection is dependent on the implemented target monitoring. Takanen et al. [3] listed several types of issues that can be discovered with fuzzing and the method of detecting them. Valid case monitoring can be used to detect catastrophic failures in the SUT by sending valid inputs to it and verifying that the output does not change. Different tools can be used to monitor the system where the target is running for arbitrary file access, unexpected network traffic, started child processes, and resource consumption. This could reveal vulnerabilities such as path traversal, information disclosure, privilege escalation, injection, and broken authentication. The target application can also be directly monitored for behaviour such as memory corruptions using debuggers, virtualization, or instrumentation using tools like Valgrind or AddressSanitizer [3, pp. 180–184]. Memory-related vulnerabilities do not necessarily result in crashes, which is why monitoring process internals using these tools is useful.

There are vulnerabilities that cannot be detected by fuzzers. Fault detection is dependent on the choise of monitoring tools. Monitors also have limitations in the types of faults that they can detect. Finding state-dependent vulnerabilities may be challenging using fuzzing. The behaviour of stateful depends on the input and the current state of the program when the input is received. Some faulty behaviour may only appear in specific program states. Fuzzing focuses on negative test cases that will not likely cause state changes. Model-based security testing is a promising alternative to fuzzing for testing finite-state machine targets, because the model enables purposeful traversal of the states [32].

4.5 Coverage-guided fuzzing

Fuzzers that use run-time collected code coverage information about the SUT to guide the selection of test cases are called coverage-guided fuzzers. Coverage-guided fuzzers have been proven effective to find bugs in comparison to black-box fuzzers. Although less effective than white-box fuzzers, coverage-guided fuzzers are simpler in implementation because they do not require analysis of the source code and its control transfers [29]. Coverage-guided fuzzers can therefore be considered a compromise between black- and white-box fuzzers and are sometimes referred to as gray-box fuzzers [28].

The instrumentation used by coverage-guided fuzzers can be relatively simple [1]. The instrumentation informs the fuzzer if the test case has experienced new code coverage. Mutational coverage-guided fuzzers use that information to determine whether a mutant test case should be mutated further. [25] Theoretically any structural coverage criteria can be used to guide the mutation. Implementing the coverage instrumentation still has to be simple enough to justify using a gray-box fuzzer instead of a white-box fuzzer. Statement or path criteria are typically used [1]. Algorithm 4.1 describes the coverage-guided random fuzzing of AFL and WinAFL.

Coverage-guided fuzzers could be considered an example of ART. The fuzzer measures the similarity of two test cases c_i and c_j by feeding them to the SUT and measuring coverage. If the test cases invoke different code paths, they can be considered to exist in different input space segments and are therefore not similar. Because measuring similarity requires to execute the target using the input, some of the tested cases will inevitably be similar to each other $(c_i \nsim c_j)$. The fuzzer starts with a corpus $C = \{c_1, c_2, \ldots, c_n\}$ of n test cases that is diverse so that all the cases are distinct $\forall c_i, c_j \in C, c_i \neq c_j \rightarrow c_i \nsim c_j$. The fuzzer then selects a case $c_i \in C$ and creates some amount of mutants of it $M = \{\hat{c}_{i1}, \hat{c}_{i2}, \ldots, \hat{c}_{im}\}$. A mutant in this set can either be similar to some existing test case in the corpus or it can invoke new code coverage in which case it will be included in the corpus on the next iteration.

This alone does not make coverage-guided fuzzing different from pure RT. The improvement to RT comes from the assumption that the target's input can be parameterized as a tuple $c = (l_1, l_2, \ldots, l_N)$ where the control transfer conditions of the target depend only on one or two of the parameters. This assumption usually holds true for programs with structured data inputs. For example, the header sections of TCP frames can clearly be represented as parameters that are mapped to specific offsets in the binary frame. On the other hand, the mutation operations are designed to modify only small parts of the input so that only one or few of the parameters are affected. The fuzzer can also use a model of the input structure to ensure that only individual parameters are affected by the mutation operations.

Program 4.2 can be used as an example of how a coverage-guided genetic algorithm progresses. We assume that the used mutation algorithm selects one of the input characters and changes it to some other random value within the constraints. The efficacy of

```
1  // Stopping criteria for fuzzing.
2  Time time;
3
4  // Queue of test cases for mutation and testing.
5  Queue queue;
6
7  // Covered code paths or nodes.
8  Path[] paths;
9
10 while( time_elapsed < time )
11 {
12     // Scheduling algorithm determines how much effort
13     // is spent mutating a single test case.
14     (Input parent, int energy) = pick_input( queue );
15     Input child = parent;
16     for( int i = 0; i < energy; i++ )
17     {
18         int n = random_integer();
19         for( int j = 0; j < n; j++ )
20         {
21             // Use a random mutation function to mutate the test case.
22             int mutation = random_mutation_function();
23             child = mutate_input( child, mutation_functions[mutation] );
24         }
25
26         // Execute the target and measure coverage.
27         Path path = execute_system_under_test( child );
28         if( path not in paths )
29         {
30             // Mutant child invokes new coverage.
31             paths.add( path );
32             queue.add( child );
33         }
34     }
35 }
```

Algorithm 4.1. *WinAFL coverage-guided mutational fuzzer algorithm [1].*

coverage-guided fuzzing depends on the number of nested control transfer conditions in the target. If the seed test case for the mutation algorithm invokes very shallow coverage, the probability of finding a mutant that passes the next nested condition is close to or even lower than the probability of finding such a test case randomly.

For a mutant test case to satisfy a condition p_i in the example program, the mutation algorithm must first select the corresponding parameter l_i in the input to mutate. Next, the algorithm has to randomly select a value that satisfies the condition. In the given example, the probability of mutating an input that passes the first condition is $(\frac{1}{4})(\frac{1}{25})$ by mutating the input `str = "AAAA"`. The other conditions cannot be satisfied by using a single-order mutation of the seed case. The probability a pure random test satisfying the first condition

```
1  /* Constraints:
2   * 1. str must have exactly 4 characters.
3   * 2. Characters in str must be in the range A-Z.
4   */
5  void Function(char* str){
6      if(str[0] == 'F')
7          if(str[1] == 'U')
8              if(str[2] == 'Z')
9                  if(str[3] == 'Z')
10                     abort();
11 }
```

Program 4.2. *Example of a program benefitting from coverage-guided fuzzing.*

is only $\frac{1}{26}$. The difference between RT and coverage-guided fuzzing becomes clear on the second condition. By mutating the input `str = "FAAA"` the probability of passing the second condition is again $(\frac{1}{4})(\frac{1}{25})$. On the other hand, passing this condition randomly would be $(\frac{1}{26})^2$.

Coverage-guided mutational algorithms are not as effective for finding new coverage if the control transfer predicates in the target depend on multiple input parameters. Mutants that do not invoke new coverage are discarded by the algorithm. If the mutants are discarded after mutating only one input parameter, it may not be possible to find a mutant that passes a predicate with multiple dependencies. The algorithms can mutate multiple parameters by using higher-order mutations before discarding the mutants. For a mutation algorithm to be able to theoretically test the entire input space, the algorithm must be designed so that an nth-order mutant can be any value in the input space and the algorithm randomly mutates a seed case n times.

Sometimes the target program contains control transfer predicates that depend on a large number of input parameters. Coverage-guided fuzzers can have difficulties testing these targets. For example, nested checksums are common roadblocks for fuzzers, although solutions for fuzzing those have been proposed [33][28]. As another example, targets that consume compressed inputs are difficult to fuzz test because changing an input parameter may require changing multiple parts of the input data. Structure-aware fuzzers can solve these problems as long as the input model is accordingly configured.

Coverage instrumentation can also be used to determine if an input has caused an execution of a specific function. This can be useful information, for example, to determine if a recently modified piece of the code has been covered by testing. It can also be used to implement a directed fuzzer. Böhme et al. [26] used a coverage distance measurement as a heuristic for determining the energy value for queued inputs. Input mutants that invoked paths that had shorter calculated distance to the target code block were given more energy for further fuzzing. This directed gray-box fuzzer was determined by Böhme et al. to outperform both gray-box fuzzers like AFL and symbolic-execution-based fuzzers like Katch. They also describe the directed gray-box fuzzer to be significantly faster than

symbolic-execution-based fuzzers because it does not require expensive analysis and constraint solving.

Even with coverage-guided methods, reaching full code coverage adequacy using fuzzing is not feasible with most targets. However, because the coverage is continuously calculated, other stopping conditions could be defined. One option proposed here takes inspiration from the elbow-method of determining the number of clusters for a k-means algorithm. The assumption is that fuzzing increases more in the beginning of fuzzing when very few code paths have been covered. As time goes on, more of the mutants end up being equivalent to either the parent test case or some other input in the queue and as a result the speed of new code paths decreases. The point at which the speed of finding new coverage decreases below some threshold could be used as a stopping condition.

On the other hand, stopping the fuzzer while new code paths are still consistently found could be considered wasteful. Any input that invokes new code paths has a potential of discovering a fault. The best fuzzing results would be achieved by letting the fuzzer run until no new coverage is invoked with any mutants. However, reaching this point may take even days depending on the target. Fuzzing takes up processing resources especially if the machines used cannot be used for other tasks without the risk of interfering with coverage measurements of the fuzzer. Therefore, it may not be cost-effective to continue fuzzing to that point.

Defining an arbitrary time allocated for fuzzing may also not be the ideal solution. Without previous knowledge of how fast the selected fuzzing tools finds coverage in the specific target it may be difficult to determine a suitable stopping time. Some feasible code coverage could be left untested if the fuzzer is stopped too soon. Then again, processing resources could be reserved for too long if the fuzzer is stopped late. There is also no guarantee that the fuzzer will reach the same coverage in the same time on two different fuzzing sessions. Changes to the target may affect the required time. Considering these issues, the elbow-method could be an efficient method of determining the stopping condition.

4.6 Cloud fuzzing

Although fuzzing has been proven to be an effective method for discovering security vulnerabilities, there is a notable cost involved. First, fuzzing requires for a suitable tool to be selected, the tool to be integrated with the SUT, and for the fuzzing environment to be built. Secondly, processing time must be allocated for fuzzing. According to the principles of CI/CD, tests are usually run on dedicated servers instead of developers' machines. Dedicating such a server for fuzzing comes with a monetary cost. Cloud fuzzing and fuzzing-as-a-service are emerging solutions that aim to mitigate especially the second issue of resource allocation.

OSS-Fuzz [34] is one good example of a cloud fuzzing service. It is designed for continuous fuzzing of open-source software using the Google Cloud Platform. Open-source

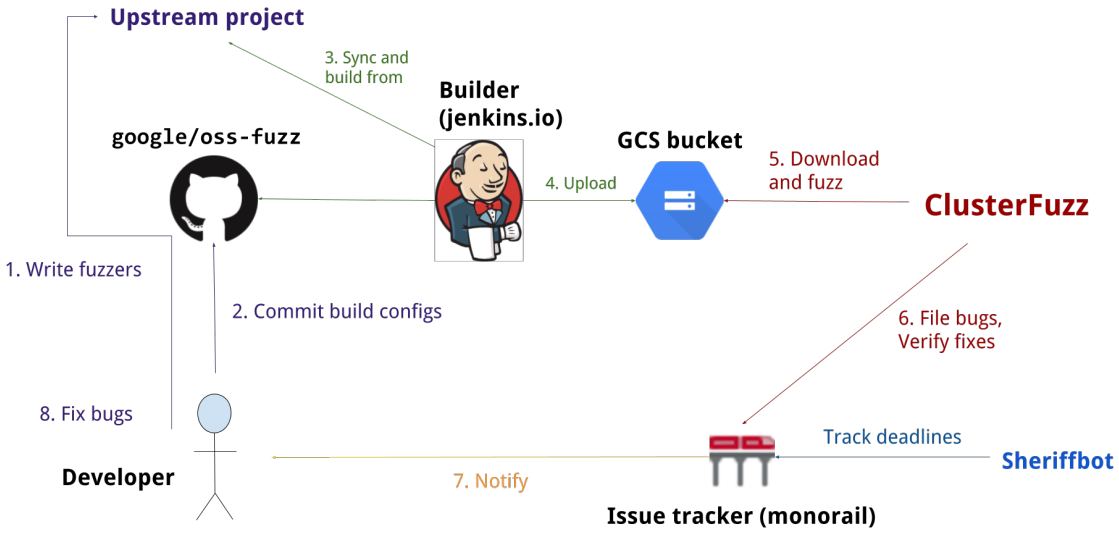


Figure 4.2. *OSS-Fuzz testing process [34].*

projects especially have an issue with resource allocation for testing. Because the ownership of an open-source project is shared, there may not be a responsible party who is willing to shoulder the costs for testing resources. OSS-Fuzz does not charge for testing resources and instead offers a monetary reward for integrated projects [34]. However, a project must be approved before it is taken into testing. The approval process requires for the project to either have many users or for the project to be critical to the global IT infrastructure. The approved projects include gRPC, Git, and OpenSSL. Google reports to finding over 200 000 bugs in different software.

OSS-Fuzz targets are compiled and run in Docker containers [34]. The target project must define some fuzzing targets which are linked with LibFuzzer [24]. Once the fuzzer target has been integrated, a new project directory is added to the OSS-Fuzz repository [34]. This directory contains metadata about the integrated project, a Dockerfile for the build environment, and a script that retrieves the project code and builds the fuzzer targets. After the project directory has been pushed to the OSS-Fuzz, OSS-Fuzz builder starts to occasionally build the included Docker images and saves them to a Google Cloud Storage bucket. The ClusterFuzz cloud service then takes the image for fuzzing. When a bug is found, a report is added to an issue tracker and the project developer is notified. The build system automatically responds to fixes in the projects based on commit messages and verifies the fixed versions. The process is depicted in figure 4.2.

ClusterFuzz depicted in figure 4.2 is the fuzzing infrastructure that is responsible for the actual fuzzing processing. It is not an actual cloud service offering, but an open-source collection of scripts, templates, and a web user interface that can be deployed to Google cloud services. Unlike OSS-Fuzz, ClusterFuzz enables cloud fuzzing of targets that are not open-source, such as commercial products. The costs are dependent on the cloud resource usage. The infrastructure offers most of the same functionality as OSS-Fuzz and it is used in similar way. It can be scaled to leverage very high computing resources

for fuzzing.

Microsoft also had a cloud fuzzing service offering called Microsoft Security Risk Detection, formerly known as Project Springfield [35]. The service used SAGE [29] for white-box fuzzing with symbolic analysis. This service was discontinued June 25, 2020 in favour of OneFuzz [36]. The open-source OneFuzz project contains scripts and templates for launching Azure resources that can be used to host fuzzing. The project enables container-based fuzzing that can be deployed to virtual machine scale sets for distributed fuzzing. The reserved Azure resources can be de-allocated when no fuzzing is taking place, which saves costs. Surprisingly, the project does not seem to offer support for containerized fuzzing. Instead, the scripts use Azure Virtual Machines to host the fuzzing jobs. Therefore, the jobs should do fuzzing in parallel in order to take full advantage of the reserved resources.

5 M-FILES PRODUCT

M-Files enterprise content management solution is the primary product offering of M-Files Inc. The M-Files Product, or just M-Files, is used to manage potentially large volumes of documents and unstructured data that is stored in one or more locations. The managed files are associated to metadata that describes the information and can be used to search it without knowing its actual location. The metadata can be either be entered by the users or automatically inferred from the associated files using artificial intelligence algorithms. The metadata consists of key-value-pair properties where property definition is the key and property value is the value. The property definitions define the type of values, name of the property, and other characteristics for the property. Although the ability to keep the managed files stored in various locations is a major selling point of M-Files, the metadata has to be stored in an SQL database.

An object is a unit in M-Files that encapsulates the metadata, files, and version data of a single managed piece of information. There can be several different types of objects, such as project, customer, contact person, or the built-in document. Every object must have the built-in title and class properties as well as some implicit non-user-managed properties such as last modified timestamps and version numbers. An object can also contain one or more files, although some object types may not allow saving any objects. The class property is used for more granular classification of objects than the object type. The type and class of the object define the required and possible property values as well as the default property values. Some property definitions can be used to represent relationships between objects. For example, a customer object could be related to several customer project objects. Objects can also contain workflows which model the customers' internal processes and access control lists that limit permissions to access and modify the object.

A vault is an M-Files abstraction of the database that contains the metadata. A single server can serve multiple vaults. Different vaults have their own objects, object types, property values, and other configurations. Separate vaults can be configured, for example, for different divisions in the customer's organization. The metadata structure of a vault can be configured to meet the specific business requirements of those divisions. A single user can be allowed access to multiple vaults depending on their position in the customer's organization. Customers are recommended to use Windows Active Directory users, but specific M-Files users can also be configured.

Perhaps the most essential feature of M-Files is searching objects based on metadata or file contents. Object metadata and files are indexed using one of the supported search

indexing engines. The index is used to search objects based on a text search that consists of keywords. Objects whose metadata or files contain the keywords are included in search results. Objects can also be searched using property value search conditions in which case the server uses database queries to find the correct objects. Using folders is avoided except for special cases. Instead, M-Files uses views that are configured with search conditions and always contain the objects that match to those conditions. The objective is to make the location of files and metadata objects irrelevant to end users.

M-Files enables modification of the base product offering features using extensions. Different extension types that are available in the catalogue include connectors for external repositories, intelligence services, and UI extensions. Connectors are required if the files managed by M-Files need to be stored on some external repositories such as network folders, OneDrive, or SharePoint. Files that are exposed by a connector extension can be searched like other objects, but no metadata is stored in the vault database until the file is promoted as a managed object. The file content remains in the external repository for the entire life cycle of the object even if the file is promoted to managed object. Intelligence services extensions are used to automatically infer property values from object contents. These alleviate the required user input to maintain accurate metadata of the objects. UI extensions are usually used to make visual representations of the vault objects. While connector and intelligence service extensions are executed on the server, UI extensions are executed on every compatible client that accesses the vault.

5.1 M-Files architecture

M-Files is a monolithic multi-tier client-server information system. The data that is managed by the system is hosted on a database server. Optionally, the files can also be stored on file servers instead of the database. When connector extensions are used, files and metadata can also be persisted in external systems. The client applications are user interfaces that allow for searching, displaying, and modifying metadata objects. M-Files Server implements the logic layer of the system. It processes the data in the system to create the metadata object abstraction. It handles access control, text searches using the index, email notifications, and scheduled operations to the managed objects. Although currently there are several different interfaces that the clients and other components can use to communicate with the server, work is ongoing to make the gRPC API the only used interface to access the server. Because of technical limitations, some clients cannot use the gRPC interface directly. Instead, they use HTTP/1.1 protocol to send JSON representations of the gRPC Protocol Buffers requests which are converted to actual gRPC requests by a separate Node.js server.

M-Files server can either be deployed to customer's on-premises environment or used as a cloud service. With on-premises deployments, M-Files customers fully control the system and the environment that hosts it. This option is usually preferred if the customer organization handles highly sensitive information that cannot be trusted to a cloud ser-

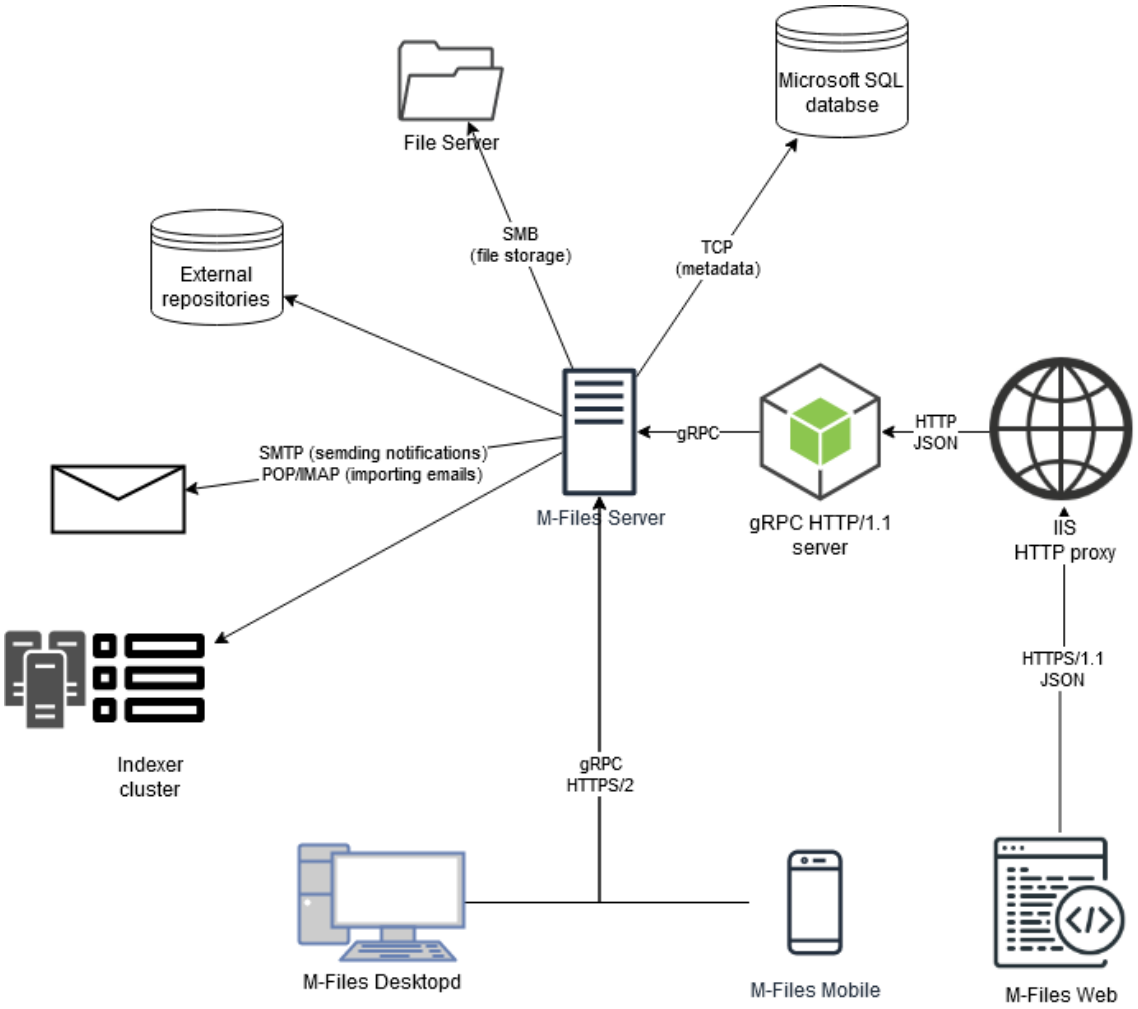


Figure 5.1. *M-Files on-premises architecture.*

vice. The architecture of an on-premises M-Files system is depicted in figure 5.1. The server is installed to a machine running a supported version of a Windows operating system. The database is usually hosted on a Microsoft SQL Server. Files managed by the system can be persisted in the database, M-Files Server's local directories, or on a separate SMB file server. A Node.js server is used on the M-Files Server machine to host the web client and to connect M-Files Web to the M-Files Server gRPC API.

In cloud deployments, M-Files Inc. or its partners manage the servers. A vault is specific to a single customer, but a single server may serve multiple vaults. In this offering model, customers cannot make server-level changes or install extension scripts, but they can control most other vault-level configurations such as the metadata structure. Customers can also choose to buy a dedicated cloud environment that is hosted in an isolated container and allows customers to use full system administrator privileges in configuring the server. The isolation allows customers to safely install extensions and scripts without risking other customers' environments. In addition to the M-Files Server interface, the cloud environment includes a separate interface for managing cloud servers and vaults. It is exposed by a Web application that is intended for administrators in the customer organizations and controls some vault-level settings and users. The M-Files cloud operations

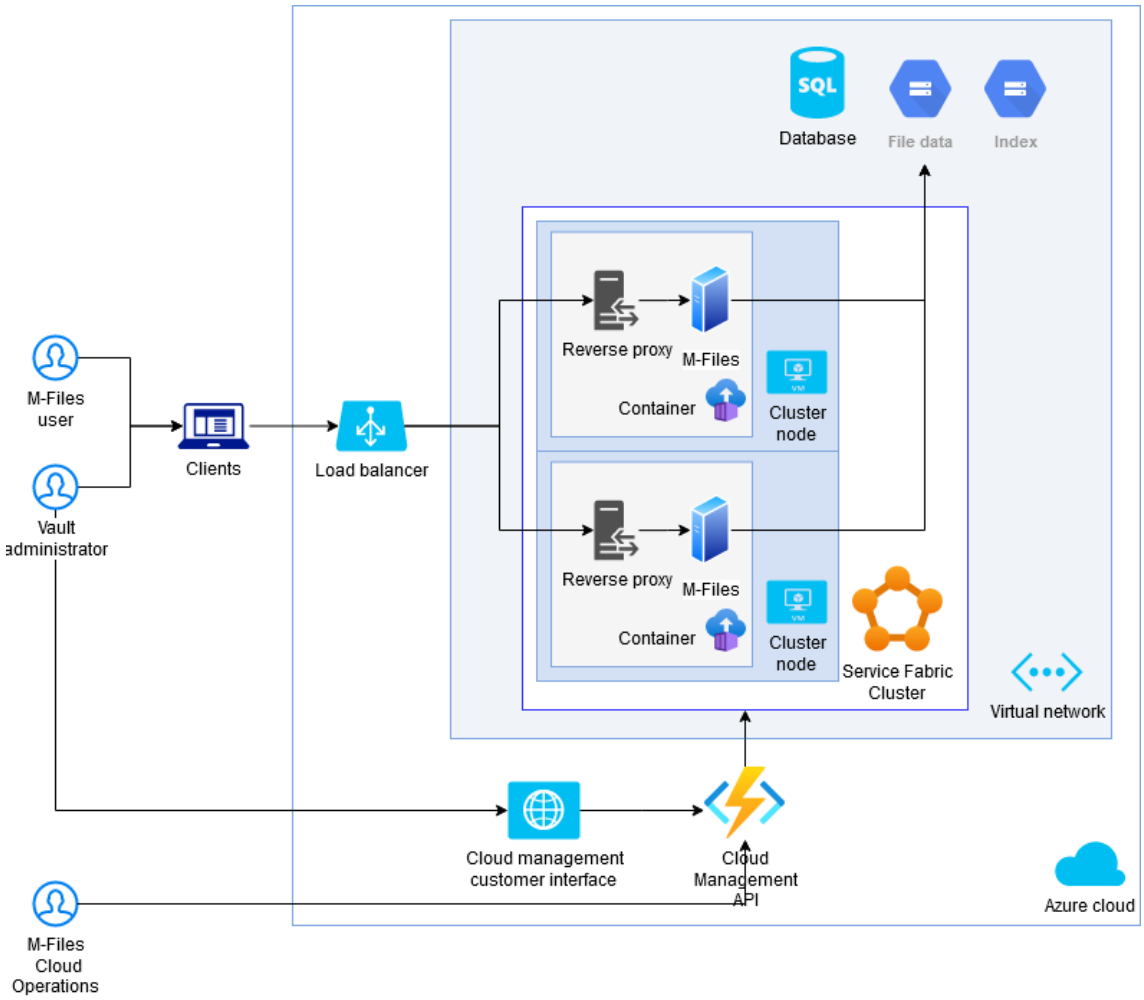


Figure 5.2. *M-Files cloud architecture.*

team also uses the API to maintain the server instances by deploying updates, creating new instances, or scaling existing ones.

At the time of writing, there are 3 types of client programs for M-Files. M-Files Desktop client is the most used and most actively developed client application. All new client features are primarily implemented for this client. It integrates with Windows File Explorer and Microsoft Office programs. Updates to the desktop program are automatically distributed from an automatic update server. M-Files Server can also be configured to host a web client. Users can access the vaults from anywhere by connecting to the server using a web browser. The files are actually hosted on a separate Node.js process that comes installed with M-Files Server and is spawned as its child on startup. The Node.js server also functions as an adapter for the M-Files Server's gRPC API because the Web client is unable to use the gRPC protocol due to technical limitations. The Node.js server exposes an interface that uses a JSON representation of the gRPC Protocol Buffers messages over HTTP/1.1 and converts them to gRPC format when communicating with M-Files Server. There is also a mobile client available for Android and iOS devices. The installations are served freely from the platforms' respective application stores.

5.2 M-Files as an attack target

The M-Files system has many assets that need to be protected. The managed files likely contain the largest volumes of sensitive information. The metadata information can be considered as sensitive as file data, because it can contain information such as credentials to external systems. The confidentiality, availability, and integrity of these two assets are the highest priority for M-Files' security. The search index may contain some individual words from both file and metadata. Although the original files can likely not be reconstructed using this index, it can still contain highly confidential information such as credit card numbers or credentials. Its availability and integrity are less of a priority because those only affect the searchability of the vault objects and the index can be easily rebuilt as long as the original information is intact. However, rebuilding the index can take a long time in large vaults with many objects, and the searchability of objects is essential for their availability. The extensibility framework causes a compromised M-Files Server to also compromise the client machines and much of the users' personal data stored on them.

M-Files Server has access to all the important assets in the M-Files system. It also has interfaces that are exposed to the Internet, some of which are even advertised to end users and publicly documented. A vulnerability in these interfaces would possibly compromise the assets of multiple organizations that use M-Files. M-Files Server is therefore an attractive target for malicious actors. Possible objectives for an attack include changing bank account information in managed invoices to redirect money payments, disruption of organization's operation by a denial-of-service attack, or ransomware attacks on the end users' personal machines. M-Files Server is a C++ application which means that some types of bugs may lead to arbitrary code execution vulnerabilities. This type of vulnerability would expose the entire system and its assets. Privilege escalation vulnerabilities would enable the same attacks because it could be used to install malicious extensions.

The JSON gRPC API that is used by M-Files Web is as exposed as the actual gRPC interface. Fuzz testing this interface would therefore also be valuable. However, this interface is only served by the Node.js server whose only purpose is to deserialize the JSON representation of Protocol Buffers requests and pass them to M-Files Server using gRPC protocol. Aside from serving the static web files, the Node.js does not do any file operations. It is also not involved in authenticating users or processing the managed data. Because it is implemented with JavaScript using an open-source library for JSON Protocol Buffers deserialization, it should be less susceptible to code execution vulnerabilities than M-Files Server. Therefore, fuzz testing of the JSON API was considered lower priority than the gRPC API and was not included in the scope of this thesis.

6 FUZZER IMPLEMENTATION

To find a suitable fuzzing tool and to investigate its ability to find bugs, a fuzzer had to be implemented for M-Files Server's gRPC interface. M-Files Inc. wanted the fuzzer to be as efficient as possible with reasonable implementing effort. It was clear that achieving that would require taking advantage of previous fuzzing research by using existing tools rather than implementing all fuzzer features in-house. Selecting the most suitable tools and integrating them with M-Files Server required careful consideration.

The first step was to define some requirements that a fuzzing tool would have to fulfil. Second step was to research different fuzzing tool options and select the most suitable tool for integration. The work done by Manès et al. [28] to collect different research papers and fuzzing tools proved to be very useful in finding suitable tools. The different options were roughly trimmed into a shortlist of fuzzers that were inspected more closely. Finally, the design choices for integrating the selected fuzzing tool had to be decided.

6.1 Fuzzer selection criteria

In order to find fuzzing tools that could be used to test M-Files Server, it was important to define the requirements that the fuzzer would have to fulfil. These requirements were set by analysing the target program and the target interface. The requirements could be used to later rule out unsuitable fuzzers to create a shortlist of fuzzers that could be further studied and compared.

Secondly, it was important to define some criteria for comparing fuzzers. It would not have been possible to integrate multiple fuzzing tools with M-Files Server in order to compare their results. Therefore, the criteria had to be defined so that they could be used to evaluate fuzzers based on just their documentation and possibly any research papers published related to them.

6.1.1 Environment requirement

M-Files Server currently only supports Windows environments. Although it is possible to run a fuzzer on a different system than where the target is running [3, p. 175], doing so complicates target monitoring. Especially if the fuzzing tool monitors target internals, the tool may not support remote monitoring. Therefore Windows support for the fuzzing framework is also required. Alternatively the fuzzer could have a separate monitoring

agent component that can be executed in Windows environments. Porting Unix fuzzers using runtime environments such as Cygwin [37] was not considered because there was no confidence of getting those methods to work with M-Files Server.

This requirement ended up excluding a surprising number of fuzzers from consideration. A majority of the analysed fuzzers seemed to focus on testing software targeted for Unix systems. In addition to the operating system, a fuzzer must support testing C++ in order to function for M-Files Server. Tools intended for testing programs in languages such as Java would not be usable with the M-Files products.

6.1.2 Initialization time requirement

The time it takes to initialize M-Files Server on start-up is substantial. The server has to spend time doing file and database operations to read configurations and to attach the managed object vaults. Therefore, a fuzzer that would restart the process between iterations would have very poor performance in terms of executed test cases per second. Therefore, the selected fuzzer should be able to deliver inputs without restarting the server process.

One method of achieving this requirement would be to rely on an input loop on the server. For a server target this should be relatively simple for a server target because sending a request and receiving a response naturally denote a single iteration of the input loop in the target. However, some fuzzers are designed to test console applications where the input is only passed as a parameter when starting the program. Another method would be to use in-memory fuzzing [3, p. 161] to reset the target's internal state to some snapshot between iterations.

6.1.3 Structure awareness requirement

A gRPC fuzzer has to be able to efficiently select input data that can be successfully de-serialized from Protocol Buffers format. If the de-serialization fails, processing time is wasted on testing Protocol Buffers parsing rather than the service implementing the API. It was assumed that a structure unaware fuzzer would be too inefficient in selecting test cases that pass Protocol Buffers parsing even if the fuzzer uses a genetic algorithm to guide input mutation. This was verified by using a coverage-guided structure-unaware fuzzer to repeatedly mutate an actual Protocol Buffers serialized message for some time.

A simple console application was implemented for the Protocol Buffers message mutation test. The application reads a file path from command line parameters and uses the Protobuf library to de-serialize the file into a Protocol Buffers message. After every de-serialization the application incremented counters for successful de-serializations and total iterations in a CSV file. On every 1000 iterations, the application added a new line with the current counters to a CSV file. Parsing the message file and updating the counter files were excluded from coverage instrumentation so that unsuccessful de-serializations

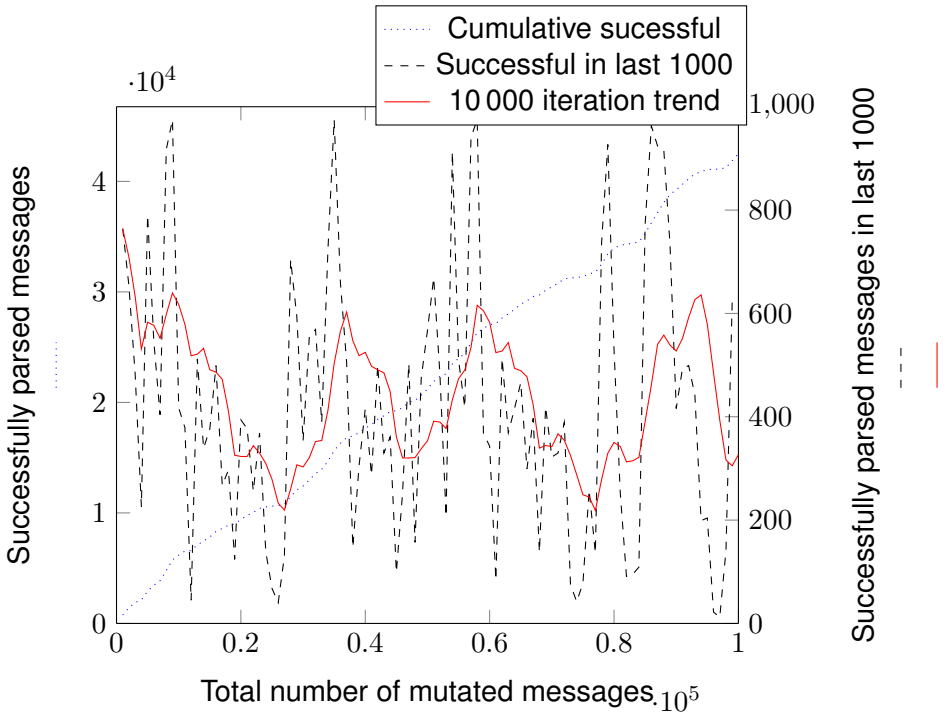


Figure 6.1. *Results of mutating Protocol Buffers messages using WinAFL.*

would always result in the same coverage. Finally, the program would use a compression library to compress the raw message data if de-serialization succeeded. The compression library was included in the instrumented modules. The purpose of this was to simulate a gRPC server program that has potential code paths to cover if parsing the messages succeeds.

WinAFL [1], a branch of AFL [23], was selected to fuzz the test application, because its dynamic binary instrumentation could easily be used to instrument the compression library. A valid M-Files Server login request message was used as the seed file for mutation. The application was fuzzed for 100 000 iterations. The results are depicted in figure 6.1. The figure shows that the ratio of successful de-serializations varies greatly over time. From the 100 000 iterations 42 504 were successful de-serializations, meaning that in total 42.5 % of mutated messages were valid. WinAFL had saved 140 inputs that it had determined to invoke new code coverage. 139 of those inputs contained valid login requests that could be de-serialized to M-Files Server login messages. Because there was only one possible code path when de-serialization failed, this suggests that the coverage measurements and the genetic algorithms were functioning correctly.

The results show a few distinct spikes in the ratio of successfully parsed and total messages in previous 1000 iterations. It is unknown whether these are caused by WinAFL selecting a specific mutation function or selecting an opportune seed input for further mutations. However, there is no notable improvement in the ratio of successfully parsed messages over time. The conclusion of the test is that if Protocol Buffers messages are mutated without structure awareness, the majority of the mutants are not valid messages. Using such a mutator to fuzz a gRPC server application would waste more than half of

the processing time testing protobuf de-serialization rather than the server implementing the interface. Therefore, a structure aware mutator would be preferred.

6.1.4 Cost requirement

At first, both free and commercial solutions were searched based on the possible protocols that they advertised to be able to fuzz. M-Files Inc. was prepared to invest some amount of money to purchase a licensed fuzzing solution. Some commercial solution providers, including Peach Tech, were contacted to query about availability of tools to fuzz gRPC interfaces. The situation changed after the COVID-19 pandemic started affecting the global economy. Investing into a new testing tool would have been difficult. The selected fuzzing tool would therefore have to be relatively inexpensive.

To solve the cost requirement issue, open-sourced solutions were the primary target of consideration. Also fuzzers with freely available pre-built binaries were considered. The licensing conditions of the found fuzzers were not verified at this stage, because there are no plans to publish the finished M-Files compatible fuzzer. In addition to free solutions, paid security analysis tools that were already in use were considered.

6.1.5 Monitoring granularity criteria

The differences of black-box, gray-box, and white-box fuzzrers in effectiveness has been well studied, but the results have sometimes been conflicting. Still, the consensus seems to be that white-box is more effective than gray-box while black-box is the least effective of the three. The purpose of fuzz testing M-Files Server is to discover vulnerabilities so that they can be fixed before any attacker has time to find and exploit them. That is why it is important to maximize the effectiveness of finding faults by selecting an efficient fuzzing tool.

Many white- and gray-box fuzzers require the source code to function [28]. The access to the source therefore gives us a unique advantage over potential attackers if the selected fuzzer is able to capitalize on it. In comparing fuzzers, white-box fuzzers would be regarded the highest with gray-box fuzzers the second and black-box fuzzers the last.

6.2 Selecting a framework

Three methods were used to find possible fuzzing tool options. First, the tools that are already in use for security testing at M-Files Inc. were considered. More specifically, the considered tools were OWASP ZAP [38] and Burp Suite [39]. There already has been work done to integrate these tools with M-Files Server Web services, and there are multiple testers who have experience using them. The Burp Suite is already licensed, meaning that there would be no extra cost. For these reasons, there would be strong arguments for using one of these tools to also fuzz test the gRPC interface.

The second method of finding fuzzing options was by going through the fuzzers listed by Manès et al. [28] and selecting the fuzzers that fit the requirements. Of the 11 open-source white-box fuzzers, 4 could not be easily found by using a search machine and 7 were not designed for Windows environments. In other words, none of the fully or partially white-box or fuzzers fulfilled the set requirements. Next, the 16 listed open-source gray-box fuzzers were roughly inspected. Of these, 7 were only for Linux or Unix-like systems, 5 were for programming languages other that C or C++, and 2 were focused on testing kernels. This left 2 possible gray-box fuzzers for further analysis. The exclusions were made based on rough assumptions based on available documentation. Table 6.1 lists the considered fuzzers and the requirements that they fulfilled.

The 2 gray-box fuzzers that we were left with were LibFuzzer [24] and Redqueen [33]. The documentation of Redqueen was not enough to confirm whether it fulfilled the environment and initialization requirements. Aschermann et al. [33] also described the tool as a prototype of a novel input-to-state correspondence fuzzing method rather than a mature testing tool. For these reasons, this tool was also excluded. This left LibFuzzer as the only suitable gray-box option that was listed by Manès et al. On the other hand, there were multiple listed black-box fuzzers that seemed to fulfil the requirements. Because there were too many options for in-depth analysis on each one, a single black-box fuzzer was selected. Peach [30] was chosen as the one fuzzer because it was brought up in conversations within M-Files Inc. in regard to investigating fuzzing options.

The third method of finding fuzzing options was by using a search engine. The top result for fuzzers with Windows support was WinAFL [1] by Google's Project Zero. Although it has not received recognition by Manès et al., it is forked from from AFL [23] which they credit as the forerunner of fuzzers using gray-box fuzz configuration scheduling. Based on source code review, WinAFL seems to follow AFL's approach very closely. It was also written by a well-known professional group of security analysts. For these reasons, the tool could be considered credible. It may have been possible to also find other Windows branches of the fuzzers listed by Manès et al.

6.2.1 OWASP ZAP

OWASP Zed Attack Proxy [38] is a man-in-the-middle proxy for penetration testing web applications. It is used by M-Files Inc. for automatic security testing of the M-Files Web client and M-Files Server's REST API. ZAP also has a tool for fuzzing web services. A custom plugin could possibly be implemented that would generate valid gRPC request messages.

OWASP ZAP has been found to raise large amounts of false positive findings when testing API services. The tool seems to assume that the output of the services is in HTML format. For example, every API method whose output somehow reflects the input is reported as a cross site scripting vulnerability by ZAP. The user experience of the tool has also been found lacking in previous testing. Finally, the tool does not appear to support HTTP/2

Table 6.1. *Open sourced white- and gray-box fuzzers listed by Manès et al. [28] evaluated to the requirements.*

Fuzzer	White / gray-box	Found online	Target environment	Input loop / in-memory	Structure aware	Other rejecting reason
Angora	White*	✓	Unix	N/A	N/A	
Chopper	White	✓	Unix	N/A	N/A	
Cyberdyne	White*		N/A	✓	N/A	
Dewery et al.	White		N/A	N/A	✓	
Driller	White*	✓	Unix	N/A	N/A	
KLEE	White	✓	Unix	N/A	N/A	
Narada	White		N/A	N/A	N/A	
QSYM	White*	✓	Unix	N/A	N/A	
SymFuzz	White*		N/A	N/A	N/A	
T-Fuzz	White*	✓	Unix	✓	N/A	
VUzzer	White*	✓	Unix	N/A	N/A	
AFL	Gray	✓	Unix	✓	N/A	
AFLFast	Gray	✓	Unix	✓	N/A	
AFLGo	Gray	✓	Unix	✓	N/A	
AtomFuzzer	Gray	✓	Java	N/A	N/A	
CalFuzzer	Gray	✓	Java	N/A	N/A	
DeadlockFuzzer	Gray	✓	Java	N/A	N/A	
FairFuzz	Gray	✓	Unix	✓	N/A	
go-fuzz	Gray	✓	Go	N/A	✓	
honggfuzz	Gray	✓	Unix	N/A	N/A	
kAFL	Gray	✓	N/A	N/A	N/A	Kernel testing
LibFuzzer	Gray	✓	✓	✓	✓	
MagicFuzzer	Gray		Unix	N/A	N/A	
Nautilus	Gray	✓	Unix	N/A	✓	
RaceFuzzer	Gray	✓	Java	N/A	N/A	
RedQueen	Gray	✓	✓	N/A	N/A	
Syzkaller	Gray	✓	N/A	N/A	✓	Kernel testing

* Fuzzer uses black- or grey-box features in addition to white-box [28].

traffic at the time of writing.

6.2.2 Burp Suite

Burp Suite [39] by PortSwigger is another penetration testing tool that is used by M-Files Inc. Similarly to OWASP ZAP, Burp functions as a HTTP proxy and is able to manipulate the traffic that is routed through it. It also has its own fuzzing capabilities, which are based on injecting various strings in request parameters. The tool analyses the response strings for error messages that could indicate security vulnerabilities. The tool has been found to perform better than OWASP ZAP in terms of true and false positive findings in M-Files Server testing.

This tool also provides extensibility features which could possibly be used to implement a gRPC fuzzer. There are also some existing Burp extensions for decoding Protocol Buffers messages [40]. However, M-Files testers were not able to get these extensions to function correctly for gRPC requests. This may have been caused by the fact that gRPC protocol prefixes the request messages with the message size. Burp also does not have HTTP/2 support at the time of writing, although support seems to be on the roadmap for 2020. Implementing a fuzzer using Burp was determined to be too difficult.

6.2.3 Peach fuzzer

Peach fuzzer by Peach Tech [30] was one of the fuzzers that were considered for testing M-Files Server. it has both an open-sourced community edition and a commercial licensed version. The commercial version appears to have mostly the same functionality but offers additional model definitions and a graphical user interface. Peach is a black-box fuzzer, as it does not do static or dynamic analysis to monitor target internals. It uses both generation- and mutation-based techniques for test case selection.

Peach requires model definitions for both generation- and mutation-based fuzzing. Both input data and program states can be modelled using XML definitions. A data model for a gRPC interface would require modelling of both Protocol Buffers encoding, the data types, and the actual API message definitions. The Protocol Buffer encodings, such as varints and ZigZag, may be easy to model using .NET modules as custom transformers. The different Protocol Buffers field types could probably also be modelled without much difficulty. The API message definitions on the other hand would require a massive effort to model. The messages in the M-Files Server gRPC definition can be very large with many fields and deep hierarchy. There are hundreds of messages and enumerated types, which would take ages to model by hand. However, it should be possible to automate the model generation using either custom scripts or Peach analyser modules.

Although Peach does not monitor target internals, there are multiple different types of monitoring available as modules. These modules can be used to monitor unexpected process exits, opened windows, memory limits, or network responses. These monitors

are also used for controlling the target, such as starting and stopping it. The monitor modules are loaded by a separate agent process, which can run on a different machine than the actual fuzzer process. This enables remote fuzzing of targets.

The M-Files Server states may also be challenging to model. On one hand, a state definition could be very helpful for fuzzing. As an example, the M-Files Server offers functionality to check-out data objects for editing. Any further calls to check-out files will fail until the object is checked-in. Therefore, if the fuzzer is testing the check-out API method and the request succeeds, the code coverage of subsequent fuzzed request have no chance of invoking code coverage past the predicate that checks for checked-out state. A state-aware fuzzer could potentially restore the state by checking in the object on purpose. However, previous attempts to use model-based testing for M-Files Server have been unsuccessful. State modelling for Peach may face similar challenges.

Publishers are the Peach modules that deliver test cases to the target. By default, peach contains publisher modules for file IO, command line applications, TCP server and client interfaces, and HTTP. At the time of writing, no gRPC or HTTP/2 publishers seem to be available in either the community or commercial versions. Such a publisher module should be relatively simple to implement. A custom publisher could be written by implementing some interface in a .NET assembly. The interface does not appear to be well documented online, but it could probably be inferred from analysing the code. The publisher could either use the gRPC library or a high-level HTTP/2 library to send the gRPC messages. This input delivery method would rely on gRPC network input loop and would not require the target to be reset between iterations.

6.2.4 American fuzzy lop

AFL [23] is a popular fuzzer framework. The fuzzer uses a genetic algorithm to mutate input test cases. The algorithm is guided by code coverage that is measured during fuzzing. Studies have been made for improving the effectiveness of the algorithm using different heuristics [25]. AFL is intended for Unix systems, but a separate branch WinAFL [1] has been written for Windows environments.

AFL is designed for testing command line applications that read input from a file. AFL saves the mutated data to the input file before executing the program. This means that a server application under test would have to be modified to run as a command line program. While modifying M-Files Server this way would be feasible, WinAFL also supports implementing custom input delivery module for example for network fuzzing [1]. A gRPC delivery module could be implemented with reasonable effort by using an HTTP/2 client library.

AFL uses a modified versions of GCC or Clang compilers to build the target program with instrumentation [23]. The instrumentation is required for measuring code coverage. These modified compilers are likely the reason why the core AFL branch cannot be used on Windows environments. WinAFL uses dynamic instrumentation instead. It offers a

few different options for instrumentation, one of which is based on DynamoRIO [41].

Launching M-Files Server using the DynamoRIO instrumentation option was observed to be very slow. However, this option enables a persistent mode, where a target function is named and DynamoRIO resets the target process state to the beginning of that function after the function returns [1]. This would enable continuous fuzzing without having to launch M-Files Server between iterations. By logging the target function's parameter values it was observed that this feature was unable to correctly reset the parameter values if they were changed. There is also an In App Persistence mode which provided better results but requires the target program to loop the target function.

AFL uses syntax-unaware mutation. It is optimized for fuzzing data formats such as images, multimedia, compressed data, regular expression syntax, or shell scripts [1]. Because of the binary format of Protocol Buffers messages that are used by gRPC, selecting test cases without structure definition was assumed to be inefficient. Although the genetic algorithms used may slightly alleviate the issues of structure unawareness, the previously mentioned test results showed that the portion of invalid mutant is too great for practical application.

WinAFL can be optionally executed with a user-provided library containing functions for mutating and feeding data to the target [1]. A Protocol Buffers mutator function could be easily implemented for WinAFL using *libprotobuf-mutator* library [42]. The library accepts a generalized protobuf message and mutates its primitive fields individually. This ensures that the mutant message is always valid. In the comments of the library [42] users are encouraged to implement mutator functions for primitive field types using another mutation library instead of using libprotobuf-mutator by itself. AFL and WinAFL have some design choices that make re-using their mutations difficult. For example, all the mutations are implemented in a single function using jumping statements. A protobuf mutator for WinAFL would therefore have to either rely on the default mutation functions or the AFL mutation operations would have to be manually copied.

Another attractive aspect of the WinAFL extensibility feature is that it could be used to send gRPC messages to the target over network. The target server would not have to be modified to accept input from a file. The gRPC client code could be implemented statically using code files generated from the `.proto` files or dynamically using a dynamic protobuf message factory and a high-level HTTP/2 library. The latter option would make the fuzzer totally reusable for any gRPC server target.

6.2.5 LibFuzzer

LibFuzzer is a coverage-guided mutational fuzzer that is based on AFL [23]. The LibFuzzer website [24] lists many of the same features as AFL It uses genetic algorithms to improve its fuzzing over time. It is also able to fuzz the target program continuously without restarting the program similarly to WinAFL. It also works in Windows environments as long as the target can be compiled using Clang compiler.

LibFuzzer has some advantages over AFL. First, LibFuzzer can be executed with other Clang sanitizers such as AddressSanitizer, UndefinedBehaviorSanitizer, and MemorySanitizer. These sanitizers can for example detect memory errors that do not necessarily result in a crash but could possibly be exploited. These sanitizers would slow down fuzzing due to the overhead of the instrumentation but would significantly improve the quality of fuzzing results.

LibFuzzer also allows users to implement custom mutators. This could be used to implement a structure aware mutator to improve the efficiency of fuzzing. The *libprotobuf-mutator* library was designed to integrate with LibFuzzer [42]. The intended purpose of the integration seems to be implementing structure aware fuzzers for targets that do not necessarily consume Protocol Buffers messages [24]. However, the library integration would naturally also work for protobuf-based gRPC.

The problem of LibFuzzer is that it requires the target to be built with the clang compiler. M-Files server is a very large code base with some ageing code and configurations. It has been designed to be compiled with the Microsoft Build Engine and the Visual Studio compilers and the building process is not very flexible. Although Visual Studio can be installed with the clang compiler, M-Files Server could not be compiled with it without changes. Changing the code to enable clang compilations was investigated, but the conclusion was that it would require much effort.

6.3 Selected framework

WinAFL was determined to the best alternative for fuzzing M-Files Server. The hope was that using gray-box methods would result in better testing coverage than using the black-box Peach fuzzer. Additionally, there was no need to define any data models, because WinAFL could be easily integrated with the libprotobuf-mutator library. LibFuzzer may have been preferred for its sanitizers and cloud fuzzing integrations, but the requirement for clang compiler was problematic. On the other hand, WinAFL could be used to test binaries from any pre-built installers rather than building a specialized fuzzing version of the server. Compiling M-Files Server can take several hours, so this feature represented a significant time save in developing the fuzzer. The time spent compiling would not have been such an issue when the fuzzer is in production and continuously testing new versions. However, back when implementing the fuzzer using a server-side harness was investigated, the incremental build would occasionally not work, which caused long delays in fuzzer development.

With the fuzzing framework selected, there were still some steps to take before M-Files Server could be fuzzed. First was to integrate WinAFL with the libprotobuf-mutator for efficient mutation of Protocol Buffers messages. Secondly, test case delivery had to be designed. Analysis of the WinAFL documentation and source code revealed that it was possible to fuzz M-Files Server over network without any changes to the server code. It was decided that this test case delivery would be implemented using generated gRPC

code and the gRPC library. There were also some issues in the base WinAFL code, that had to be resolved for fuzzing of M-Files Server.

6.4 Test case selection

For efficient structure-aware mutation, WinAFL was integrated with the libprotobuf-mutator library. The integration was trivial using the WinAFL custom mutator module feature. Libprotobuf-mutator is able to mutate Protocol Buffers messages by applying the mutations on the messages' fields before the messages are encoded. When mutating a message, the library first selects one of the following mutations:

- Add new field with default value.
- Mutate field value.
- Delete a field.
- Replace a field's value with a copy of another field's value.
- Clone the value of a field into a new field.

If a field value is mutated, the mutation functions depend on the field's type. The default library implementation comes with very simple mutation functions: random bit flips are applied to integer and floating-point values, boolean values are inverted, and random value is selected for enumerated values. String values can be mutated by adding or removing random bits and applying bit flips. UTF-8 encoding is maintained for mutated UTF-8 strings. If mutation operation is selected for a nested message, the mutation functions are applied to it recursively.

These mutation functions lack some of the commonly used fuzzing mutation operations. For example, boundary values for numerical values and format strings for text values are not used. This type of mutation functions would probably not result in very high error-based coverage. In fact, using the default library mutations is discouraged in a comment of one of the header files [42]. The user of the library is supposed to implement the mutation functions for the different primitive field types using some other mutation library, such as LibFuzzer. The mutation functions used by AFL include walking bit and byte flips, arithmetic operations using blocks of data, and boundary values.

Unfortunately, all the mutation operations of AFL and WinAFL are implemented in a single mutations, which makes reusing them difficult. The operations had to be rewritten as reusable functions[1] to integrate them with libprotobuf-mutator. The AFL author's blog post [43] that shortly described the mutation logic was invaluable in replicating the behaviour. The mutations can be divided into two groups. First, deterministic mutations are applied sequentially over the whole input buffer. The mutant is delivered to the target after every mutation operation to measure whether it invokes new code coverage. The deterministic operations are followed by random and stacked mutations, which is referred to as the havoc stage. The time spent in this stage is determined by the energy value. Because

[1] `https://github.com/NimatLa/AFLMutationFunctions`

Table 6.2. *Mutable field types in libprotobuf-mutator and the AFL operations that could be applied.*

Field type	bit flip	Boundary integers	Arithmetic operations	Set random byte	Block deletion	Block insertion	Set random block	Dictionary
`int32_t`	✓	✓	✓	✓	✓		✓	
`int64_t`	✓	✓	✓	✓	✓		✓	
`uint32_t`	✓	✓	✓	✓	✓		✓	
`uint64_t`	✓	✓	✓	✓	✓		✓	
`float`	✓		✓	✓	✓		✓	
`double`	✓		✓	✓	✓		✓	
`bool`								
`enum`								
`string`	✓	✓	✓	✓	✓	✓	✓	✓

libprotobuf-mutator randomly selects the message-level mutations and the fields for field value mutations the replicated mutation logic could not be made deterministic. Therefore, the deterministic mutation operations were not integrated.

The AFL random havoc stage seems to mostly consist of the same mutation operations as the deterministic stage. The difference is that havoc chooses randomly the mutation operation and the area of the input buffer where the operation is applied. These havoc operations could be used to mutate the field values with libprotobuf-mutator. However, some mutation operations are not very applicable for some field types. The mutation and field type pairings are depicted in table 6.2.

The AFL mutation operations for interesting values only considers integers. These values include common buffer sizes and integer boundary values. These values might not be very likely to cause issues with floating-point fields. The implemented mutation algorithm could therefore either ignore the interesting values operation or use some custom interesting values for floating-point types. However, the version implemented as part of this thesis mutates floating-point values reinterpreted as integers. The block insertion operation is only applicable to string because that is the only type with a variable size. The other types are considered constant sized despite their values. For example, a 32-bit integer's length is 32 even if its value is 0. The Dictionary operation was not implemented for this version. It would be a valuable improvement in later versions for testing M-Files, because it could be used to inject SQL syntax into the data. This could help finding SQL vulnerabilities in the system, although detecting them would require some method to monitor SQL exception.

As seen in table 6.2, boolean and enumerated type values are special cases for which no mutation operations are applicable. There are only two possible encoded values for a boolean field, as was observed using a test application using the Protocol Buffers library. If the field's value is true, the value is encoded as a single byte varint with value 1. If the field value is false, the field is excluded from the encoded message. Therefore, the only possible mutation operation is negation. On the other hand, enumerated type fields can have any 32-bit values on C++ implementations [9]. The mutation functions that are applicable to `int32_t` type would therefore technically also be applicable for `enum` fields. However, the valid values should probably be used more than non-existing values. It is likely that `enum` values are often used in switch-case statements where all non-existing values would be handled as the default case. It would still be valuable to use non-existing values in fuzzing in case that those default clauses are faulty or missing. The problem is that the libprotobuf-mutator library only allows the `enum` mutator to select one of the existing values. Producing non-existing `enum` values would require changes either to the library or the `enum` field descriptors.

The user-provided mutator module feature of WinAFL does not provide an energy value to the custom mutator function. The energy value is an important factor for the genetic algorithm because it allows some queued test cases to be prioritized over others. The energy determines how many times the queued test case in mutated and tested before moving on to the next test case. The code was changed to also look for a custom mutation function that accepts the energy value as a parameter[2]. Running the custom mutator with the energy value resulted in the fuzzer finding more unique code paths per second than without energy.

Aside from the mutation functions, the original seed test case selection also affects coverage of mutational fuzzers. To achieve maximum coverage in M-Files Server API methods, the original seed cases must result in successful execution of the method. The number of inputs that result in failure of the API call is typically much greater than the number of inputs that can be used to call the method successfully. For example, a login method that takes username and password as parameters would only be successful if the username existed in the database and the password was correct for that user. Using mutation, it would be difficult to mutate successful login input data from seed that does not already contain the correct credentials. On the other hand, finding inputs that do not pass validation is trivial in comparison. By using correct seeds, minimally mutated test cases would achieve more deeply nested coverage.

For the seed test cases to have a chance of successfully executing API methods, the environment had to be set up to contain the data expected by the seed cases. As an example, the expected login accounts had to be configured on the server. This could have been done by hand, but that would have taken an enormous effort especially if the seed cases had to be changed later or if more API methods were taken into fuzz-testing. Setting up the environment using the same scripts that were used for generating

[2] `https://github.com/googleprojectzero/winafl/pull/293`

the seeds was determined to be the better option. That would also guarantee that the environment always contained the exact data that the seed cases expected.

The generation of seed test cases would have to be implemented for every fuzz-tested method. The API definitions alone did not contain enough information for generating both correct seeds and an environment where those seeds would succeed. This meant that the seed generation required manual work for implementing the scripts. With hundreds of API methods in the interface, this would take much time. To ensure that all methods eventually get fuzzing coverage, the generation scripts would have to notify users of methods without coverage. Using the existing API integration tests to generate both the environment and the seed cases was also considered. The major challenge of this approach would be to identify whether an API call is used to set up the environment or to actually test the API. For this thesis, only the login method seed generation was implemented.

There was one issue in WinAFL, which prevented mutation of Protocol Buffers messages. AFL and WinAFL trim their saved test cases to reduce their size on disk and to improve performance [1]. The inputs are trimmed by iteratively removing portions of the input buffer and measuring the coverage. If coverage remains unchanged, the trimming is applied permanently. This trimming resulted in invalidation of Protocol Buffers messages in almost all cases. This was fixed by adding an option for a custom trimming function to the user-provided module[3]. If WinAFL detects the custom function, the regular trimming is skipped, and the custom function is called instead. Custom trimming could have been implemented using a libprotobuf-mutator type mutator object that only reduces the input message's size. This would have required changes to the libprotobuf-mutator library, and it was therefore left unimplemented.

6.5 Delivering inputs

There were two options for delivering inputs to M-Files Server. The first was to implement a gRPC client on the fuzzer's side. The second was to implement a feature in M-Files server that reads input protobuf messages from a file and calls the corresponding gRPC method implementation. This is the recommended method of testing server targets with AFL, because it lacks a user-provided test case handling features. Both methods were attempted during development of the fuzzer.

Instead of modifying the SUT, writing a gRPC test case delivery function for WinAFL was preferred. Implementing a fuzzing harness on the server-side would have required testing to bypass some important functions in the API method call stacks because of the opaqueness of some used objects. A server side harness would also have required every tested server version to be specifically built for fuzzing with some additional preprocessor macros to avoid the harness from getting into production code. Implementing a custom fuzzer delivery module enables fuzz testing of pre-built binaries that are also used for other testing. This would significantly decrease the time resources required by fuzzing.

[3] https://github.com/googleprojectzero/winafl/pull/294

Another decision related to delivering test cases was whether to use gRPC code files that are generated from `.proto` files using protoc compiler or to write a generic gRPC client using a high-level HTTP/2 library. Although the `grpc_cli` tool can dynamically call gRPC methods given the method name and description, that feature is not available in the standard gRPC library. Therefore, the gRPC library could not be used to implement an interface-generic client. Using an HTTP/2 library would make the delivery module totally reusable for any gRPC server target. It could also be used to test any method in a single interface as long as the method routing information and valid request messages were provided to the module.

This thesis was scoped to only focus on the M-Files Server login method. Therefore, the selected implementation choice was a user-provided delivery module that uses generated gRPC code to send messages to the one endpoint. Implementing generalized gRPC server input delivery was left as a possible topic for further development. While a generalized dynamic method would have been more suitable for open-source publishing, the static method may fit M-Files requirements better. some M-Files Server API methods change the server state in ways that could affect coverage-guided fuzzing. A state restoring function could be implemented for the delivery functions of those API methods. This would not be possible to implement as a generalized module.

Both seed case generation and input delivery would require manual effort for each gRPC method. Although fuzzing support for methods could be implemented iteratively over a long time period, that would leave a large portion of the interface without fuzz-testing for a long time. A malicious actor could possibly find a vulnerability in the interface during this time. Therefore, it would be recommended to implement generic seed generation and delivery methods for API calls without specific fuzzing implementations. Although, generic methods would likely not result in deep coverage for these methods, the coverage of the whole API could be much wider.

6.6 Monitoring testing

WinAFL uses instrumentation to measure code coverage and to monitor for faulty behaviour. DynamoRIO [41] is used for dynamic instrumentation of the target binaries. The instrumented code uses a named pipe to communicate with the WinAFL process. The instrumented code uses the pipe to signal about detected exceptions that would result in crashes. It is also used to stop the target execution until WinAFL is ready to input the next fuzzed test case. The coverage map is held in a shared memory bit vector, which is visible to both the instrumented process and WinAFL. This information is used in the fuzzer process to select which inputs are saved for further fuzzing.

DynamoRIO [41] is a platform that allows runtime analysis and modification of a program. The platform loads basic blocks into a cache, where it is modified before it is executed natively. This approach should improve performance compared to emulation [41]. WinAFL also has two other instrumentation options [1], but those do not appear to support user-

provided mutation and delivery functions according to source code analysis.

The WinAFL DynamoRIO client collects either basic block or edge coverage for each fuzzing iteration. The client maps a shared memory bitmap of 2^{16} bytes initialized with zeros that is shared with the fuzzer process. For every basic block taken to execution the DynamoRIO client calculates an offset `offset = pc - module_start` where `pc` is the current value of program counter register and `module_start` is the starting memory address of the executed binary module. In basic block mode, the client calculates the hit counts of the blocks by incrementing a byte in the shared memory bit vector: `coverage_vector[offset]++`. In edge mode the offset that is used to increment the cache counters is determined as `offset = (pc - module_start) ^ (prev_offset >> 1)` where `prev_offset` is the similarly calculated offset of the previous basic block. This way the occurrence count of specific branches can be calculated. The coverage vector is susceptible to path collisions, meaning that two different paths can appear to be the same in the vector. Although this issue has been previously solved [44], the WinAFL code is not going to be fixed as part of this thesis.

Because both block and edge mode calculate the number of occurred basic blocks and branches respectively, they both consider different loop iteration counts as inequivalent paths. This could result in infinite possible paths similarly to path coverage criterion. More specifically, there could be 256 distinct path from a single loop because the single octet counter would eventually overflow. As a solution, WinAFL classifies each counter based on the ranges that their values fall to. The different classification ranges are $0, 1, 2, 3, [4, 7], [8, 15], [16, 31], [32, 127], [128, 255]$.

By default, WinAFL's DynamoRIO mode does not restart the target between every fuzzing iteration. Instead, the instrumentation code attempts to save the program's internal state when the target function is hit and to restore that state once the target function exits. This fulfils the requirement of working around M-Files Server's long initialization time. However, when this feature was tested using a server-side fuzzing harness, it was not able to correctly restore some function parameters that were passed by pointers. There are also two options to run WinAFL with the assumption that the target loops the target function itself. The in-app option considers the fuzzing iteration to be over when the target function was called the second time. This worked better than the default option with the server-side harness but couldn't be used with the fuzzer-side gRPC client because a suitable target function was not found. The other option simply skipped the target state reset using the `no_loop` parameter. This was used in the WinAFL TCP fuzzing example and it worked with the fuzzer-side gRPC delivery.

M-Files Server occasionally does some scheduled background operations. These could interfere with coverage measurements of WinAFL. Using the `thread_coverage` options would alleviate this. In this mode, only the threads that have hit the target function modify the shared memory coverage vector. However, once a thread hits the target function, coverage measurement of that thread is not stopped until the target process exits. This is a problem with gRPC that uses thread pools to serve API requests. After a thread returns

from the target API method to the thread pool loop, it can serve other requests that what fuzzing targets. This would be the case, for example, if the input delivery tries to reset the server state using other API methods. The thread pool threads also occasionally do some operations even if there are not API requests in the queue. The WinAFL instrumentation code had to be changed so that it stops measuring coverage after the thread exits the target function in the `no_loop` mode[4].

When testing the fuzzer during its development, there were a few occasions when the instrumentation code reported a crash. WinAFL was not able to detect these crashes and instead stopped fuzzing because it received an unexpected message from the named pipe. The issue was that the crash message was read in a wrong phase of fuzzing. The target process had already returned from the target function when the crash occurred, which is not something WinAFL knew to expect. This is because WinAFL instrumentation code resets the target state in normal operation mode. In M-Files Server, some background thread may cause a crash even if the API call has already returned. WinAFL had to be modified to check for crash messages between two fuzzing iterations.

In addition to the instrumentation, some other target monitors were also considered. One attempted monitor was Application Verifier [45], which is one of the debugging tools that are shipped with the Windows Software Development Kit. It is able to dynamically monitor a running application and apply stresses and tests to it. Unfortunately, the instrumented M-Files Server could not be executed with Application Verifier enabled with any configuration. Trying to run the instrumented server always resulted in a crash before the server was able to fully initialize. The saved crash dumps and log files were not enough to deduce where the crash occurred, although they did suggest that the crash was caused by a heap corruption. After a while, it was determined that Application Verifier simply could not be used to monitor an instrumented M-Files Server instance.

Another considered monitor was the Global Flags debugging tool [46]. GFlags is used to enable advanced debugging features on running applications. The features of GFlags include page heap verification that adds inaccessible pages after every heap allocation. If the monitored program tries to access this page, the program is stopped. The hope was that the resulting crash dumps would help to debug where memory faults occur in the M-Files Server code even if the crashes were not easily reproducible. Fuzzing with GFlags options enabled had a significant effect on the performance of fuzzing. The execution speed reduced from roughly 40 test cases per second to around 2 when GFlags was used with the options that were recommended by an M-Files security mentor.

6.7 Measuring coverage

After a fuzzing run is finished, the fuzzer will have generated collection of test cases. Each test case should execute a distinct code path. On the other hand, unsaved mutations that were selected and executed during fuzzing should not exercise any additional coverage

[4] `https://github.com/googleprojectzero/winafl/pull/292`

in the criterion used by the fuzzer. Therefore, by executing the SUT using the test set, the resulting coverage should be equal to that of the fuzzing period. Any code coverage tool can be used to generate a report of the fuzzing coverage. The generated test set can also be used to calculate a mutation score for the fuzzer, if those measurements are required.

The scope of this thesis was limited to a single API method. The possible coverage for a single method in M-Files Server is a fraction of the entire code base. This meant that in order to calculate an accurate indication of testing coverage, the structural coverage should only consider the nodes that are reachable from the target method. Accomplishing this would require for the coverage tool to analyse the code for possible and impossible paths. However, a tool with such a functionality was not found. The coverage percentage of the results would therefore not indicate any useful information by itself and would always have to be compared to other results for any useful analysis.

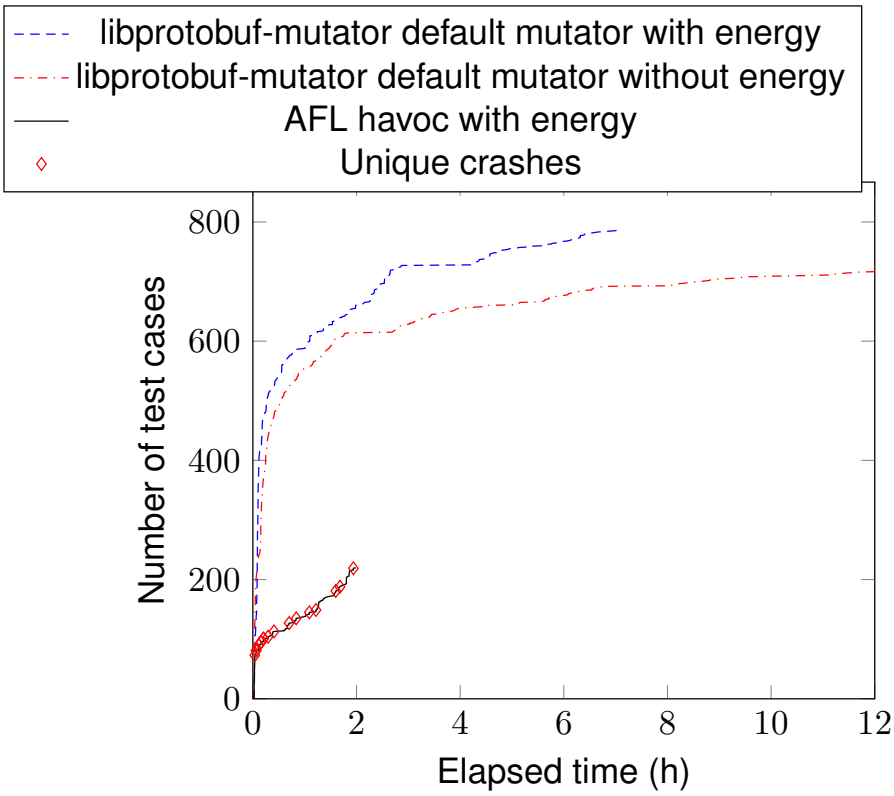
7 FUZZING RESULTS

One of the objectives of this thesis was to investigate the fuzzer's ability to find bugs in the system. Another objective was to study how long an M-Files API method would have to be fuzzed to achieve adequate coverage. Effectively, this would involve defining the adequacy for fuzzing M-Files Server. Both objectives required actually running the selected fuzzing tool and analysing the results.

There were several issues with the fuzzer itself that caused difficulties in testing M-Files Server. First, the dynamic instrumentation of WinAFL was unstable in some environments. The instrumented server process would consistently crash on systems with an antivirus enabled when the server process was launched. Even on environments with antivirus disabled the instrumented process would start crashing after an operating system update. The fuzzer had to be moved to a Windows Server machine with updates and antivirus disabled to solve these issues. Even so, the fuzzer had issues restarting M-Files Server during fuzzing. The instrumented server would occasionally crash, hang, or just not launch at all. The crashes and hangs were eventually accepted as instrumentation transparency issues because they could not be reproduced without the instrumentations. The problem was mitigated by increasing the default restart rate from 1 thousand to 2 billion iterations which effectively disabled restarting. The fuzzer would still restart the server when it detected crashes or hangs and the issues would still occur if there were repeated findings.

After getting the fuzzer to work, it was set to test the M-Files login method for 12 hours or until the fuzzer aborted in order to generate a test set of distinct test cases. The fuzzer saves each mutated test case that invokes new code paths as a new file. The fuzzing coverage progress over time can be analysed by looking at when the test case files were created according to the timestamps. The results of running the fuzzer on different configurations can be seen in figure 7.1. Running the test cases without mutations should also theoretically achieve the same coverage as the fuzzing session that generated the set. This would enable making coverage and mutation score measurements using any available tools as necessary. The WinAFL instrumentation code was unable to create human readable reports of the covered lines, which is why a separate code coverage measurement tool had to be used. The coverage tool was set to monitor M-Files Server while the test set gRPC messages were sent to it using a client console application.

The coverage measuring tool used measures statement coverage whereas WinAFL measures a form of path coverage. A greater number of test cases would not guaran-



Mutator	Run time (s)	Total executions	Rate (/s)	Total paths	Unique hangs	Unique crashes	Covered lines
----	30 177	5.68 M	188.22	789	0	0	16 570
-·-·	57 001	6.10 M	107.02	737	4	0	16 412
——	7390	17.3 k	2.34	221	0	17	16 154

Figure 7.1. *Results of testing M-Files Server using WinAFL.*

tee higher statement coverage because path coverage considers different loop iteration counts unique paths. However, in the measurements taken the larger number of test cases happens to equate to higher statement coverage. The results suggest that there was significant value to implementing passing the energy value to the custom mutator. Unfortunately, the results of the reimplemented AFL operations are not very comparable because it kept repeatedly having to restart the server because of found crashes. The constant restarting caused the fuzzer to abort early because of the previously mentioned issue. The fuzzer also aborted early using the default libprotobuf-mutator mutation operations an unknown reason. The validity of the results is also threatened by the indeterminism of M-Files Server. The fuzzer reported some of the generated test cases to have differing coverage on consecutive unmutated iterations. The rate of running test cases may also have been affected by varying CPU load on the machine.

Test sets that achieve higher code coverage generally have better fault-finding capability than sets with lower coverage [12]. A fuzzers ability to find issues could therefore be compared to other testing methods using the code coverage of the generated test set. However, the coverage percentage itself without a comparison point is not sufficient in

investigating the fuzzer's ability to find issues. If the target application contained a number of known issues, the fault-detection ability of the fuzzer could be measured by seeing how many of those issues could be detected by the fuzzer. Because there were no saved crash-level bug reports for M-Files that were clearly related to the gRPC login method, a set of intentionally created bugs would have to be seeded into the target. However, M-Files Inc. was more interested in verifying that the fuzzer was able to correctly detect and report issues than calculating a comparable measurement of fault-detecting capability. Therefore, it was decided that bug seeding would only be done in the absence of actual findings.

7.1 Found faults

The final objective of this thesis was to investigate the implemented fuzzer's ability to find and correctly report faults. There were two potential methods of achieving this. First, the number of actual bugs found by the fuzzer in M-Files Server could be measured. Actual security flaws in a software can be sparse especially when testing is scoped to a very limited part of the attack surface. If no real bugs could be found, using seeded bugs could have been an effective alternative. On the other hand, using both actual and seeded bugs would have allowed making some estimations on the number of undetected issues. However, M-Files Inc. was more interested in verifying that the fuzzer works as specified than making such estimates. In the interest of time, it was decided that bug seeding would not be done because actual findings already verified the functionality of the fuzzer.

As seen from the fuzzing results in figure 7.1, the fuzzer was able to find actual crash-level issues in M-Files Server. All the unique crashes were confirmed to have been caused by the same issue. They were reported separately because the code paths leading up to the occurrence differed. The issue was a read access violation caused by mishandling of null-terminated strings. The issue did not cause a crash of the server because the exception was handled. It can still be considered a security issue because it may allow an attacker to access values in the server's memory. Finding this issue proves that the fuzzer is able to find some security critical bugs even if they do not normally result in a crash. Reproducing the issue required the server to be executed in a debugger. When the issue was investigated using a debug build of the server, the input also revealed an assertion failure.

The assertion failure finding gave impetus to repeating each unique test case on a debug build of the server. This resulted in a few additional assertion failure findings. The fuzzer was not able to detect any issues with these inputs because they did not result in exceptions that could be interpreted as dangerous. In fact, the issues may not be security-related but they still clearly indicate incorrect behaviour. The findings were therefore reported as bugs for developers.

7.2 Defining a stopping criterion

Reaching full adequacy using RT is impossible for a software such as M-Files Server that has a practically infinite input search space. Therefore, a stopping criterion must be set for the fuzzer. One option for such criteria would be an adequacy degree based on the percentage of the code that is covered. However, defining a required percentage of coverage is not possible because the fuzzer does not measure the percentage at runtime. The number of distinct test cases could use as a replacement for the coverage percent, but it also would not be usable for a stopping criterion. The test set generated by the fuzzer is reusable meaning that the set can be used as the seed test cases on later fuzzing runs. After the initial run, the generated set would contain the number of test cases specified in the criterion. Every subsequent run that reuses the set would therefore match the criterion before even starting. The number of new test cases also should not be used because the number likely decreases every time the set is reused as the seed cases cover more of the target code.

Another possible criterion would be to define a limited testing resource such as time or fuzzing iterations that the fuzzing run is allocated. However, such an allocation would have to be made for every method in the target API, because the possible code paths for each method varies. Setting a single criterion for all methods would likely either waste testing resources testing smaller methods or not reach adequate coverage on larger methods. Defining a criterion for each tested method would require too much work especially as the complexity of the methods may change in different M-Files versions.

From the fuzzing results in figure 7.1 we can see that the rate of finding new code paths reduces over time. If the fuzzer was left running for an extended time, it would eventually reach coverage saturation where no new test cases are found. However, the fuzzer could not reach saturation even in 12 hours of running. Because there are over 500 API methods to test, spending over 12 hours on a single method clearly cannot be afforded. Instead, the stopping criteria should stop testing at some elbow point when the benefits of continuing to fuzz start to significantly reduce.

Tramontana et al. [15] have suggested two possible stopping criteria: All Equivalent (AEQ) and All Included in One (AIO). Both criteria require measuring the coverage over time of multiple test which they then compare to the union of the sets. Especially AIO seems to be a promising criterion to stop testing in the elbow of the achieved coverage [15]. However, it is not clear if the criterion could be set while running the test. Reusing the generated set would likely cause issues with these criteria too because the fuzzer would start with the union of the previous sets. Setting the criteria before running the fuzzer would have the same workload issue as the allocated resource criteria.

The only suitable criteria that could be found was based on the interval between finding new code paths. We can use this method to determine any arbitrary point in the elbow as the stopping point. WinAFL natively has an option which when enabled stops the fuzzing when it has cycled the queued cases without new coverage. This option was not able to

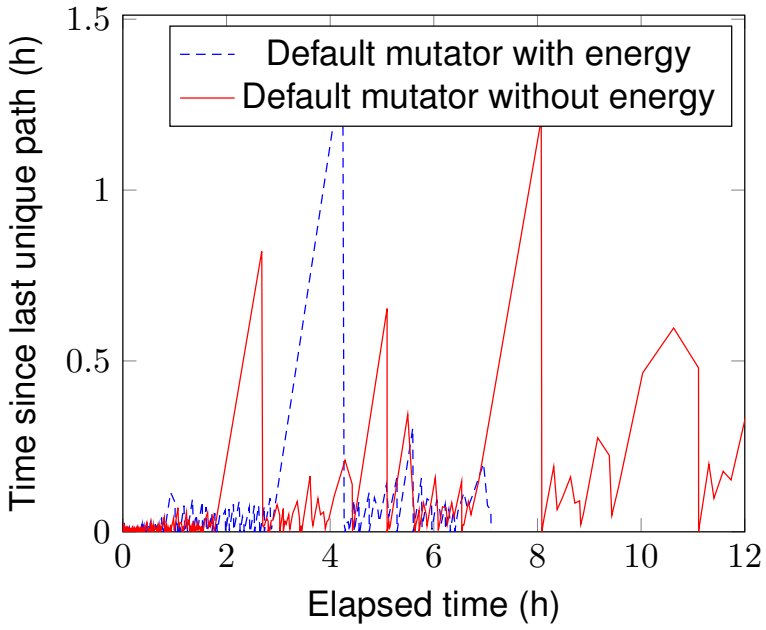


Figure 7.2. *Intervals between found unique test cases.*

stop testing M-Files Server login method within 12 hours. Therefore, a custom maximum interval without finding has to be defined and implemented in WinAFL. To define such a value, the intervals between coverage findings in the results of figure 7.1 were inspected more closely. These intervals were plotted in figure 7.2.

With both comparable runs, the fuzzing experienced a clear reduction in efficacy before running for 4 hours. However, the intervals were reduced again after the initial spike in both cases. This could indicate an insufficient initial seed test set. The seed set currently includes only one login request even though the login method allows multiple login types. It is possible that the fuzzer found some of these types after a long period of mutations. The newly found type then would have enabled many new code path findings in the new code area. If the seed set had better coverage, the interval experienced in the spikes could be used as the stopping criterion. The suggested value in that case would be 0.5 h, because that matches each of the observed spikes. Additionally, the criterion should specify that the mutation queue does not contain unmutated test cases, as is the case with the WinAFL stopping rule option.

8 CONCLUSIONS

The M-Files system was determined to be an attractive target for attackers. The purpose of this thesis was to investigate different options for fuzz testing the M-Files System and their efficacy in finding vulnerabilities. The work was scoped to only include the Internet facing gRPC interface of M-Files Server. Additionally, the work focused only on the login RPC method to save time.

The first objective of this work was to find a suitable fuzzing tool for testing M-Files Server. The interfaces, environment, and other properties of a target software dictate how it can be fuzz tested, which is why many fuzzers are developed for specific targets and are not openly available [22]. To find a suitable tool for M-Files Server, some requirements and criteria were defined based on the server's properties. Various fuzzing tools were then collected from different sources and compared to the criteria using superficial analysis of their documentations. A few tools were selected from the qualifying ones for more detailed analysis. The most suitable tool was determined to be WinAFL, which was extended to be able to test the M-Files gRPC interface.

The second objective of this thesis was to investigate if fuzz testing would be able to find issues in M-Files Server. To do so, WinAFL was used to fuzz test the M-Files API login method. There were some non-deterministic crash and hang findings, but these were suspected to be caused by the instrumentation. There was also a crash-level finding that could be deterministically reproduced. This issue was also determined to be a security issue, because it may enable attackers to access the server's memory. The issue therefore verifies the fuzzer's ability to find security issues in M-Files Server.

The final objective of this thesis was to determine a suitable stopping rule for fuzz testing. A rule based on length of intervals between found distinct code paths was selected after considering various criteria. The maximum interval 0.5 h was suggested based on the intervals observed in the fuzzing results. In the observed results, this value would have stopped fuzzing when the rate of finding new paths significantly reduced. The criterion would therefore allow the fuzzer to cover a large number of paths before reaching saturation while also leaving fuzzing time resources to test multiple API methods. It may have been preferred to define the stopping criterion in terms of total executions between found distinct code paths so that the criterion would not be affected by variance in fuzzing speed. The results measurement method would have had to be changed to also record the total executions over time to reliably define such a criterion.

The stopping criterion interval value was suggested with the caveat that the mutation seed test set would be improved to cover more login types. The intervals should perhaps have been remeasured with the improved seed set to specify a more reliable stopping rule. Also, the method used for determining the maximum interval may not be able to objectively find the optimal value, although it was the best method that was found at the time. The suggested value was purposefully set relatively lax based on the observed results so that there would be time resources available for testing multiple API methods. Judging by the fuzzing results, the suggested value could stop the fuzzing of at least some methods before 4 hours. There is not enough data to conclude that all methods could be fuzzed in this time frame because only one method was tested at this point.

Even if fuzzing of a single method could be concluded in 4 hours, fuzzing over 500 methods consecutively could take months. If fuzzing could be done in parallel, for example 10 methods at a time, the total fuzzing time may be reduced to an order of days or weeks, which would be much more practical. However, it is not possible to run multiple instances of the same M-Files Server version on the same machine because they cannot share the resources. At the same time, fuzzing multiple methods with a single instrumented server process is not possible because coverage measurements of one method would interfere with measurements of the other methods. Fuzzing could possibly run in parallel using isolated containers. Parallel fuzzing could also be achieved by allocating more machines to fuzzing. If the computing resources cannot be afforded and the fuzzing must be finished fast, one option to shorten the required time would be to select a random smoke set of methods to fuzz for each tested M-Files version.

The investigation of the fuzzer's fault-finding ability was concluded more as a binary verification than an actual analysis. A more valuable analysis could have been done by using seeded bugs together with the actual findings. This would have enabled calculating an estimate of the undiscovered actual findings left in the M-Files login method code. However, it would have likely been impossible to limit the seeded bugs to only the possible code paths from the login method using an automated mutation score tool. Manual bug seeding would have been very laborious, and the bugs created using this method may not resemble actual bugs.

The fuzzing tools that were selected for closer analysis were selected from a long list of possible tools using very superficial requirement matching. The requirements were matched based on skimming the documentations of the tools. This may have led to disqualifying some tools that were actually viable options. Some tools were disqualified because they could not be found in open code repositories, but a more thorough search may have found them. Only one black-box fuzzer was arbitrarily selected from the options for closer analysis. The rest of the tools were not validated at all. It is possible that the skipped fuzzing tools would have contained options that are more suited, have less issues, or require less integration than WinAFL.

When selecting the tool, grey-box fuzzers were prioritized over black-box fuzzers because they were assumed to have better issue-finding capabilities. However, M-Files Server

server may not be very suitable as a target for grey-box fuzzers. The libFuzzer documentation [24] describes the optimal target to be narrow, fast, and deterministic. The target should also not modify the global state. Especially non-determinism would apparently make fuzzing inefficient according to the documentation. M-Files Server API methods match none of these requirements. A black-box fuzzer may therefore have been more suitable after all. To later verify this, WinAFL could be used to fuzz the server without using the genetic algorithm. These results could then be compared to the coverage-guided results. The comparison and analysis will not be done as part of this thesis because of time restraints.

Regardless, the most important objective of this thesis was achieved. A fuzzing tool capable of testing M-Files Server was found and integrated with the server. Furthermore, the tool was proven to work and it has already produced tangible results. However, the integrated fuzzer is still not ready to be used in a production environment. Doing so still requires the fuzzer to be extended to test more gRPC API methods. It is important for the security of the system that all methods are eventually tested for vulnerabilities. The easiest method of expanding the coverage may be to implement a method-generic gRPC request sender function using an HTTP/2 library. To expand the fuzzing coverage, it will probably be necessary to solve the issue of how to restore changes to the system's state in case of successful requests. Some of the API methods can change the server's state on success so that consecutive request requests cannot achieve the same coverage. This would interfere with the coverage-guided genetic algorithm.

Secondly, implementing the AFL dictionary mutation operations would be valuable for testing SQL injection vulnerabilities. The operation uses a user-provided list of terms that are randomly included in the fuzzed data [23]. This would enable randomly mutating SQL syntax into the data. Detecting SQL vulnerabilities would also require some type of special monitor to be implemented. For example, the monitor could look for SQL errors in the class stacks returned in M-Files gRPC errors.

Although this thesis focused on testing M-Files Server, the other components in the M-Files system could also benefit from fuzzing. It should be relatively simple to change the implemented WinAFL fuzzer to be able to impersonate as M-Files Server and to fuzz the connected clients. However, the client applications are not very high priority of components to be fuzzed because if an attacker is able to impersonate as the server, they can take over the clients even without vulnerabilities using the extensibility features. The other components such as indexers and artificial intelligence cloud services could be higher priority targets for fuzz-testing.

REFERENCES

[1] Fratic, I. *WinAFL*. 2016. URL: https://github.com/googleprojectzero/winafl.

[2] Miller, B., Fredriksen, L. and So, B. An empirical study of the reliability of UNIX utilities. eng. 33.12 (1990), 32–44. ISSN: 0001-0782.

[3] Takanen, A., Demott, J., Miller, C. and Kettunen, A. *Fuzzing for Software Security Testing and Quality Assurance, Second Edition*. eng. Second edition. Artech House information security and privacy series. Norwood: Artech House, 2018. ISBN: 1608078507.

[4] Chen, T., Leung, H. and Mak, I. Adaptive Random Testing. eng. *Advances in Computer Science - ASIAN 2004*. Vol. 3321. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, 320–329. ISBN: 354024087X.

[5] *About gRPC*. URL: https://grpc.io/about/.

[6] Belshe, M. and Peon, R. *RFC 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2)*. IETF, 2015. URL: https://tools.ietf.org/html/rfc7540.

[7] Peon, R. and Herve, R. *RFC 7541 - HPACK: Header Compression for HTTP/2*. IETF, 2015. URL: https://tools.ietf.org/html/rfc7541.

[8] Ryan, L. *gRPC over HTTP2*. Google, 2015. URL: https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md.

[9] *Language Guide (proto3)*. Google. URL: https://developers.google.com/protocol-buffers/docs/proto3.

[10] Zhu, H., Hall, P. and May, J. Software unit test coverage and adequacy. eng. *ACM Computing Surveys (CSUR)* 29.4 (1997), 366–427. ISSN: 0360-0300.

[11] Chen, T. Y., Kuo, F.-C., Liu, H. and Wong, W. E. Code Coverage of Adaptive Random Testing. eng. *IEEE Transactions on Reliability* 62.1 (2013), 226–237. ISSN: 0018-9529.

[12] Hutchins, M., Foster, H., Goradia, T. and Ostrand, T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. eng. ICSE '94. IEEE Computer Society Press, 1994, 191–200. ISBN: 081865855X.

[13] Wong, W., Horgan, J., London, S. and Mathur, A. Effect of test set size and block coverage on the fault detection effectiveness. eng. IEEE Comput. Soc. Press, 1994, 230–238. ISBN: 081866665X.

[14] Wong, W. E., Horgan, J. R., London, S. and Mathur, A. P. Effect of test set minimization on fault detection effectiveness. eng. *Software: Practice and Experience* 28.4 (1998), 347–369. ISSN: 0038-0644.

[15] Tramontana, P., Amalfitano, D., Amatucci, N., Memon, A. and Fasolino, A. Developing and Evaluating Objective Termination Criteria for Random Testing. eng. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28.3 (2019), 1–52. ISSN: 1049-331X.

[16] Hetzel, W. C. *The complete guide to software testing, second edition*. eng. New York.

[17] Gold, R. Control flow graphs and code coverage. eng. *International Journal of Applied Mathematics and Computer Science* 20.4 (2010), 739–749. ISSN: 1641-876X.

[18] Myers, G. J., Sandler, C., Badgett, T. and Thomas, T. M. *The Art of Software Testing*. eng. Hoboken: John Wiley & Sons, Incorporated, 2004. ISBN: 0471469122.

[19] DeMillo, R., Guindi, D., McCracken, W., Offutt, A. and King, K. An extended overview of the Mothra software testing environment. eng. *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*. IEEE Comput. Soc. Press, 1988, 142–151. ISBN: 9780818608681.

[20] Wu, H., Nie, C., Petke, J., Jia, Y. and Harman, M. An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing. eng. *IEEE Transactions on Software Engineering* 46.3 (2020), 302–320. ISSN: 0098-5589.

[21] Arcuri, A. and Briand, L. Adaptive random testing: an illusion of effectiveness? eng. ISSTA '11. ACM, 2011, 265–275. ISBN: 1450305628.

[22] Rathaus, N. and Evron, G. *Open source fuzzing tools*. Syngress Pub, 2007.

[23] Zawlewski, M. *American fuzzy lop*. URL: `https://lcamtuf.coredump.cx/afl/`.

[24] Serebryany, K. *libFuzzer – a library for coverage-guided fuzz testing*. URL: `https://llvm.org/docs/LibFuzzer.html`.

[25] Karamcheti, S., Mann, G. and Rosenberg, D. Adaptive Grey-Box Fuzz-Testing with Thompson Sampling. eng. AISec '18. ACM, 2018, 37–47. ISBN: 9781450360043.

[26] Böhme, M., Pham, V.-T., Nguyen, M.-D. and Roychoudhury, A. Directed Greybox Fuzzing. CCS '17. ACM, 2017, 2329–2344.

[27] Li, M. J., Zhang, Q., Tang, C. J., Wu, B., Wen, S. M. and Zhang, B. Directed Fuzzing Based on Dynamic Taint Analysis for Binary Software. eng. *Applied mechanics and materials* 571-572 (2014), 539–545. ISSN: 1660-9336.

[28] Manes, V. J. M., Han, H., Han, C., s. k. cha sang kil, Egele, M., Schwartz, E. J. and Woo, M. The Art, Science, and Engineering of Fuzzing: A Survey. eng. *IEEE Transactions on Software Engineering* (2019), 1–1. ISSN: 0098-5589.

[29] Godefroid, P., Levin, M. and Molnar, D. SAGE: whitebox fuzzing for security testing. eng. *Communications of the ACM* 55.3 (2012), 40–44. ISSN: 00010782.

[30] Eddington, M. *Peach fuzzing platform*. URL: `https://community.peachfuzzer.com/WhatIsPeach.html`.

[31] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S. The Oracle Problem in Software Testing: A Survey. eng. *IEEE Transactions on Software Engineering* 41.5 (2015), 507–525. ISSN: 0098-5589.

[32] Felderer, M., Zech, P., Breu, R., Büchler, M. and Pretschner, A. Model-based security testing: a taxonomy and systematic classification. eng. *Software testing, verification & reliability* 26.2 (2016), 119–148. ISSN: 0960-0833.

[33] Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R. and Holz, T. REDQUEEN: Fuzzing with Input-to-State Correspondence. *Symposium on Network and Distributed System Security (NDSS)*. 2019.

[34] *OSS-Fuzz*. 2017. URL: https://github.com/google/oss-fuzz.

[35] *Microsoft Security Risk Detection will be discontinued June 25, 2020*. URL: https://www.microsoft.com/en-us/security-risk-detection/.

[36] *OneFuzz, A self-hosted Fuzzing-As-A-Service platform*. 2020. URL: https://github.com/microsoft/onefuzz.

[37] Cygwin. eng. 249 (2019), 84–84. ISSN: 1470-4234.

[38] *OWASP ZAP Getting started*. URL: https://www.zaproxy.org/docs/desktop/start/.

[39] *Burp Suite documentation*. URL: https://www.zaproxy.org/docs/desktop/start/.

[40] Wielgoszewski, M. *burp-protobuf-decoder*. 2013. URL: https://github.com/mwielgoszewski/burp-protobuf-decoder.

[41] Bruening, D. L. Efficient, transparent, and comprehensive runtime code manipulation. PhD thesis. Massachusetts Institute of Technology, 2004.

[42] Buka, V. *libprotobuf-mutator*. 2016. URL: https://github.com/google/libprotobuf-mutator.

[43] Zawlewski, M. *Binary fuzzing strategies: what works, what doesn't*. 2014. URL: https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html.

[44] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z. and Chen, Z. CollAFL: Path Sensitive Fuzzing. eng. IEEE, 2018, 679–696. ISBN: 1538643537.

[45] *Application Verifier*. 2017. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/application-verifier.

[46] *GFlags*. 2018. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags.