FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Juha Kälkäinen

FUZZ TESTING CONTAINERIZED DIGITAL FORENSICS AND INCIDENT RESPONSE TOOLS

Master's Thesis
Degree Programme in Computer Science and Engineering
May 2022

Kälkäinen J. (2022) Fuzz Testing Containerized Digital Forensics and Incident Response Tools. University of Oulu, Degree Programme in Computer Science and Engineering, 46 p.

ABSTRACT

Open source digital forensics and incident response tools are increasingly important in detecting, responding to, and documenting hostile actions against organisations and systems. Programming errors in these tools can at minimum slow down incident response and at maximum be a major security issue potentially leading to arbitrary code execution. Many of these tools are developed by a single individual or a small team of developers and have not been comprehensively tested.

The goal of this thesis was to find a way to fuzz test a large amount of containerized open source digital forensics and incident response tools. A framework was designed and implemented that allows fuzz testing any containerized command line based application. The framework was tested against 43 popular containerized open source digital forensics and incident response tools. As a result, out of 43 of the tested tools 24 had potential issues. Most critical issues were disclosed to the respective tools authors via their preferred communication method. The results show that currently many open source digital forensics and incident response tools have robustness issues and reaffirm that fuzzing is an efficient software testing method.

Keywords: fuzzing, incident response, software testing

Kälkäinen J. (2022) Kontitettujen digitaaliforensiikka ja tietoturvahäiriön hallintatyökalujen fuzz-testaus. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 46 s.

TIIVISTELMÄ

Avoimen lähdekoodin digitaaliforensiikka ja tietoturvahäiriön hallintaan käytetyt työkalut ovat entistä tärkeämpiä tunnistamiseen, hallintaan ja dokumentoimiseen haitallisia toimintoja vastaan organisaatioissa ja järjestelmissä. Ohjelmointivirheet näissä työkaluissa voivat pienimmillään hidastaa tieturvahäiriön hallintaa ja enimmillään olla suuri tietoturvauhka, joka potentiaalisesti voi johtaa mielivaltaisen koodin suoritukseen. Monet näistä työkaluista ovat kehittäneet joko yksittäiset henkilöt tai pienet ohjelmistokehittäjätiimit eikä niitä välttämättä ole kattavasti testattu.

Tämän diplomi-insinöörityön tavoitteena oli löytää tapa fuzz-testata suuri määrä kontitettuja avoimen lähdekoodin digitaaliforensiikka ja tietoturvahäiriön hallintaan käytettyjä työkaluja. Tähän tarkoitettu ohjelmisto suunniteltiin ja toteutettiin, joka mahdollistaa minkä tahansa kontitetun komentolinjaan perustuvan ohjelmiston fuzz-testaamisen. Ohjelmistoa testattiin 43 suosittua kontitettua avoimen lähdekoodin digitaaliforensiikka ja tietoturvahäiriön hallintaan käytettävää työkalua vastaan. Lopputuloksena testatuista työkalusta 24 löytyi ongelmia. Kriittisimmät löydetyistä ongelmista ilmoitettiin työkalujen vastaaville kehittäjille heidän suosimallaan kommunikaatiometodilla. Lopputulokset osoittavat, että tällä hetkellä monet avoimen lähdekoodin digitaaliforensiikka ja tietoturvahäiriöiden hallintaan käytettävät työkalut kärsivät ohjelmointivirheistä ja vahvistaa fuzz-testauksen olevan edelleen tehokas ohjelmistotestaus metodi.

Avainsanat: fuzzaus, tietoturvahäiriön hallinta, ohjelmistotestaus

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
LIST OF ABBREVIATIONS AND SYMBOLS
1. INTRODUCTION 8
2. TECHNOLOGY REVIEW 10
2.1. Software Testing 10
2.1.1. Black-Box and White-Box Testing 10
2.2. Fuzz Testing 11
2.2.1. History of Fuzzing 12
2.2.2. Fuzzer Categorization 12
2.2.3. Test Case Mutation 13
2.2.4. Bugs and Attack Vectors 13
2.3. Modern Fuzzers 14
2.3.1. FuzzBench, a Fuzzer Benchmarking Platform 14
2.3.2. Evolutionary Fuzzing 15
2.3.3. Code Coverage Guided Fuzzing 15
2.3.4. In-Memory Fuzzing 16
2.3.5. Machine Learning Applications in Fuzzing 17
2.4. Container Technology 17
2.4.1. Docker Arcitecture 17
2.4.2. Docker Use Cases 18
3. RELATED WORK 19
3.1. Fuzzing Frameworks 19
3.1.1. American Fuzzy Lop 19
3.1.2. Boofuzz 19
3.1.3. GitLab Protocol Fuzzer Community Edition 20
3.1.4. Atheris 20
3.1.5. Honggfuzz 20
3.1.6. Radamsa 20
3.2. OSS-Fuzz 21
3.3. Fuzzing Digital Forensics and Incident Response Tools 22
4. IMPLEMENTATION OF THE FUZZING FRAMEWORK 24
4.1. Overview 24
4.1.1. Current State of Targets 24
4.1.2. Rationale behind Building a New Framework 24
4.2. Goals for Building the Framework 25
4.3. Building the Framework 26
4.3.1. Tool Execution 28
4.3.2. Tool Inputs 28
4.3.3. Injecting Inputs 28
4.3.4. Input Generation 29

4.3.5. Detecting Abnormal Behavior or Crashes 29
4.3.6. Storing Results 30
4.3.7. Process Automation 30
4.4. Adding a Target to the Framework 30
4.5. Problems and Challenges 31
5. FUZZING TESTS 33
5.1. Fuzz Test Setup 33
5.1.1. Hardware 33
5.1.2. Target 33
5.2. Results 37
5.3. Problems and Challenges 39
6. DISCUSSION 40
6.1. Further Work 40
7. CONCLUSION 42
8. REFERENCES 43

FOREWORD

First, I'd like to thank professor Juha Röning for supervising this thesis and Rauli Kaksonen for the initial research idea and guidance on the topic. I would also like to thank my friends and family for their patience listening me ramble on about my thesis for a while now and for giving me the encouragement and support to see it through.

Oulu, May 24th, 2022

Juha Kälkäinen

LIST OF ABBREVIATIONS AND SYMBOLS

AFL	American Fuzzy Lop
API	Application Programming Interface
CERT	Cyber Emergency Response Team
CI/CD	Continuous Integration and Continuous Delivery
CSC	Center of Science
CinCan	Continuous Integration for the Collaborative Analysis of Incidents
DFIR	Digital Forensics & Incident Response
EFS	Evolutionary Fuzzing System
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
OSINT	Open Source Intelligence
OUSPG	Oulu University Programming Group
PDF	Portable Document Format
POSIX	Portable Operating System Interface
SDK	Software Development Kit
SUT	Subject Under Test
TCP	Transmission Control Protocol

1. INTRODUCTION

As cyber incidents are becoming more prevalent in an increasingly digital world, the importance of proper incident response is crucial. The growth in cybercrime from ransomware to insider threats is well known and documented [1]. Digital forensics is a critical part of computer emergency incident response investigations serving a variety of roles from digital evidence collection to many legal and industry purposes [2].

Traditionally incident response has been manual work done by utilising a few basic tools. This approach requires in-depth knowledge about the underlying system, programming, and data recording methods and is in general limited to one tool at a time. With the projected shortage of highly skilled cyber security professionals in the upcoming years [3], the trend of using automated tools and suites in incident response has become more common [4]. While this offers many benefits such as iterative use and repeatable analysis, as with any crucial software special care needs to be taken that tools used are not prone to errors and are securely programmed. Running a tool when it is needed and finding it does not work as expected is a time loss and can lead to further difficulties during a busy incident response scenario. Used tools not only have to handle valid outputs from systems but invalid corrupted data as well. In real life incidents, acquisition errors can happen where the acquired sample has been corrupted during recording or transit. Malicious actors or malware can purposely produce invalid or corrupt data to slow down incident response if used tools are unable to handle robust inputs [5]. Severe programming errors not only cost time but can be a major security issue itself leading to arbitrary code execution on the system running the tool.

Currently, many widely used digital forensics and penetration testing tools are open source. A paper studying usability of forensics tools found that among forensics experts 69% of responders used open source tools at least sometimes with only 5% never having used them [6]. As such, they face the same challenges other software projects do. Continuously evaluating the current state and usability of these tools can be difficult and time-consuming. Many of these tools are developed by a single individual or a small team of developers and may not have been comprehensively tested.

There is a variety of approaches to testing software to provide information about the quality of the subject. One such testing method is robustness testing or fuzzing. Fuzzing is an automated software testing method that is done by sending anomalous data to a system in order to crash it or cause abnormal behaviour. It is widely used by security experts to find vulnerabilities and in quality assurance to ensure standards and stability of a program [7].

During 2018 to 2020 the Oulu University Secure Programming Group (OUSPG) took part in collaborative Innovation and Networks Executive Agency funded project led by the Finnish National Cyber Security Center called CinCan (Continuous Integration for the Collaborative Analysis of Incidents). The goal of this project was to establish continuous analysis as a part of national Cyber Emergency Response Team (CERT) functionality, using tools and interfaces to rapidly create, augment, correlate and share analysis and threat intelligence in collaboration between European national teams. As a part of this project experienced digital forensics investigators and other cybersecurity professionals were interviewed. Based on these interviews, most commonly used digital forensics and incident response tools were containerized [8].

During CinCan project work it was found that some of the tools, even though commonly used and quite popular, were not necessarily up to date, entirely functional, or could even have been abandoned. Updates could bring large changes to usability of the tools and affect stability. Evaluating the current state of each tool by hand requires continuous effort and is time-consuming. From this problem the following research goals for this thesis were formed:

1. Find or implement a suitable framework that allows automatic fuzzing of containerized digital forensics and incident response tools.
2. Use the framework to fuzz digital forensics tools containerized during the CinCan project and investigate the results.

The main motivation behind these goals is to study whether digital forensics tools are susceptible to robustness issues and to see if fuzzing a large amount of complex targets is feasible.

2. TECHNOLOGY REVIEW

This chapter will present briefly the different technologies that were integral to this thesis; fuzz testing and Docker. First, description of software testing in general is given to give an understanding how fuzzing fits in it. Afterwards fuzzing is introduced, its history and the technology overall in more detail. For a more thorough introduction on the topic and more advanced fuzzing, the book "Fuzzing for software security testing and quality assurance" [9] is recommended. Finally, a brief description of Docker, its underlying architecture, and use cases is given.

2.1. Software Testing

ANSI/IEEE gives a standard definition to software testing as "Testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item." [10]. Any meaningful non-trivial software can be considered to contain errors. Testing software and finding errors add value to it by raising the quality and the reliability of the program. In addition, software testing nowadays is used for code verification and validation. Time and cost limit the scope of testing and as such, the key issue of testing becomes what subset of tests yields the highest probability of detecting significant errors [11 ch1]. Software testing needs of a large company that designs specialized products for a customer vastly differ from the needs of a single open source developer building something for themselves. Testing a complex communication protocol might need a state-of-the-art testing tool from a commercial vendor while an isolated custom application might be better off with a quick Python test script [9 p14-15].

2.1.1. Black-Box and White-Box Testing

In general, testing can be divided into two categories: Black-box testing and White-box testing. White-box testing focuses on the degree to which test cases cover the source code of the program. To achieve this, the structure and execution path of the code needs to be known. Ultimate white-box test would be the execution of every path in the program, which is not a realistic goal for complex programs. In black-box testing, the subject under test (SUT) is viewed as a black-box which structure is unknown. Instead, the focus is on finding circumstances in which the program fails to behave according to its specifications [11 ch2].

In the book Fuzzing for software security testing and quality assurance [9] black-box testing is generalized to focus on three different purposes: Feature testing, performance testing, and robustness testing. Feature testing and performance testing are both classified to be positive testing, that is tests where the SUT is subjected to set of valid use cases. For feature testing, this means a pass criterion would simply be that the software conforms to the specifications. A failure would mean some functionality was missing or did not work as specified. In performance testing, the SUT is typically subjected to large amounts of valid traffic. This could for example cause issues with network performance or cause delays or session drops with local interfaces such

as the file system or API calls. Robustness testing is negative testing where the SUT is subjected to cases that should never occur in a friendly correctly functioning environment. With robustness testing fail criterion in general can be defined that a test fails if the software crashes, becomes unstable, or behaves against specifications. Robustness weakness is the most common cause for security vulnerabilities found in public [9 p85-90].

2.2. Fuzz Testing

Fuzz testing is a novel robustness testing approach where the SUT is subjected to unexpected, random, or semi-random inputs in order to discover security vulnerabilities or bugs in software applications. It is a highly automated testing technique that covers a portion of negative tests finding boundary cases that might be extremely difficult to find with traditional software testing methods. Its main focus is on functional security assessment [7]. Fuzzing can be used to test any application or system, whether it is a standalone application or running atop of web or mobile infrastructure. The functional approach to fuzzing can vary greatly based on the target, the skills of the researcher, and the data format that is being fuzzed. The steps as seen in figure 1 generally always apply, no matter what target:

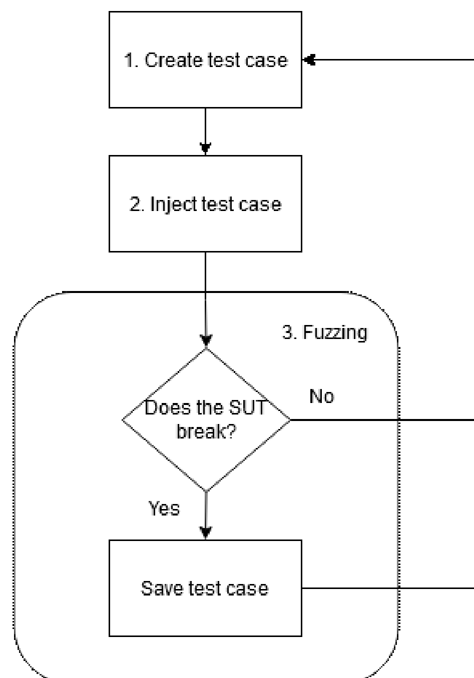

Figure 1. Fuzzing process

Depending on the goals of the researcher different emphasis can be placed on each step [12 p27-29].

2.2.1. History of Fuzzing

The idea for fuzzing originated during a stormy night in 1988 when Bart Miller, a researcher from University of Wisconsin, found that when remotely connected to Unix systems over a dial-up phone line the signal noise caused by thunder would cause common Unix utility software to crash [13]. This led to writing a paper where a fuzz generator was used to study reliability of UNIX utilities [14].

In the beginning of 1999 PROTOS-project was initiated as a joint effort by the OUSPG and Finnish Technical Research Centre. In the PROTOS-project robustness of 49 software products were tested using a black-box approach by systematically generating a large number of test cases which the products were then subjected to. As a result the project identified many serious vulnerabilities and introduced new approaches for testing software robustness which nowadays would be described as fuzz testing [15]. These testing methods were later built into a commercial software testing product by Codenomicon, later acquired by Synopsys [16].

Since then, fuzzing has grown to be an integral part of software security testing to find bugs and vulnerabilities rapidly and cost effectively. Both open source and commercial fuzzing solutions started to appear on the market during the 2000s. These fuzzers typically used some form of input file or grammar mutation. In 2014 American Fuzzy Lop (AFL) was published by Michal Zalewski which introduced compile-time binary instrumentation. This allowed the fuzzer to receive feedback during the fuzzing process which could then be used for creating more exact test cases. Feedback fuzzing further improved the testing efficacy by finding new code execution paths which resulted in discovering new bugs faster. Feedback fuzzing inspired many others coverage guided fuzzers such as LLVM libFuzzer and Google's Honggfuzz [17, 18].

Modern fuzzers have not just improved test generation but additional functionalities and features have been created such as protocol modelers, attack simulation engines, anomaly libraries and others, further improving the efficacy of fuzzing [9 p24-32].

2.2.2. Fuzzer Categorization

Understanding fuzzer categorization helps in building or choosing the right kind of fuzzer for the intended target. There are many ways to categorize fuzzers, but one method is to do so based on the following two criteria:

- According to injection vector or attack vector.
- Based on test case complexity.

Categorization based on injection vector refers to how the target can be accessed or what attack vector is possible. Some fuzzers are generic frameworks that support multiple attack vectors while others focus on single protocols such as Transmission Control Protocol (TCP) or Internet Protocol (IP). Implementations for fuzzing wireless protocols such as Bluetooth and Wi-Fi exists as well [19, 20].

Categorization based on test case complexity uses the various software or application logic layers the generated test case can target for sorting. Static fuzzers are fuzzers that focus mainly on simple request-response protocols or file formats. Block-based fuzzers

implement the basic functionality needed for simple request-response protocols and can contain additional functionality such as checksums. Evolution based fuzzers do not understand the underlying protocol but learn it by using a feedback loop receiving responses from the target system. Finally, simulation-based fuzzers implement the interface that is being tested either through a model or by simulation. They can also be a full implementation of the protocol not only fuzzing the message structures but sequences they are sent as well. Fuzzing protocol message sequences can find states where the SUT deadlocks or crashes [9 p27-30].

2.2.3. Test Case Mutation

In general test cases for fuzzing are built by mutating a valid sample input or using a library of known heuristics. Generators can be divided into two types: data generation and data mutation. Data generation can be as simple as generating an input based on a specification for how it should look. This can be for example just an integer or as complex as a PDF document.

Data mutation is based on starting with a known valid set of data and mutating it in specific or semi-random places. This could for example mean flipping bits in valid data file and seeing what happens to the SUT which has shown to be surprisingly effective [7]. More deterministic fuzzing approach has been shown to perform better in general, but a randomized approach may find some combination of inputs that deterministic approach would miss. A Hybrid approach combining both deterministic and random mutation exists and many mature fuzzers include elements of both. Completely random fuzzers are generally assumed not to find many bugs and are quite inefficient [9 p30].

2.2.4. Bugs and Attack Vectors

Important part of fuzzing is understanding what can go wrong with the target and what does it look like. Knowing what kind of bugs commonly can be found from the target is necessary for setting up proper monitoring. Numerous bugs and intricate exploits with ways to categorize and describe them exists. The goal of this section of the thesis is to give a brief overview of few significant vulnerabilities that can often be found by fuzzing.

Many types of bugs can be found in software, some more severe than others. While fuzzing works against any application no matter what language is used: C, C++, Java, Python, or others, the types of bugs that can be found can differ. Especially lower level languages such as C or C++ are susceptible to many severe memory allocation errors that are impossible or very difficult to be found in other higher level languages [9 p44-45].

Memory corruption errors have been a prevalent method for executing arbitrary code in local or remote computer systems for the past 30 years. C and C++ code traditionally handles its own memory which makes it possible to write code that executes at very high speed. At the same time if mistakes are made in memory allocation, it opens the program up to possible exploitation. This makes C and C++ applications particularly

susceptible to fuzzing; they're fast and found issues can be very severe. If memory can be corrupted, often the progress of the program can be redirected to execute a malicious payload. Mismanaged memory allocations can also lead an attacker to be able to access memory reads that should not be available, possibly leaking private keys, passwords and other private information.

Another common security issue is denial of service and other performance bugs. These can cause the system to be unresponsive to its intended use or is available at a degraded capacity. While these problems are relatively easy to detect the condition can often be transient and the system can restore itself to full capacity quickly. Denial of service bugs are still very critical issues; if an attacker finds an input that causes the system to reboot, they can render an application to be no longer available simply by repeating the attack.

File system related problems are common security issues where an application can inappropriately interact with the underlying file system. An attacker can use these types of issues to view arbitrary files on the system with the permissions of the application. Another file system related issue is injecting NULL characters into requested filenames which can cause issues creating predictable temporary file names.

Metadata injection vulnerabilities are another broad class of problems especially common with web and mobile applications. These involve injection of metadata into a process, such as SQL injections, command line injection vulnerabilities, cross-site scripting and others. SQL injections are a common form of vulnerability where an attacker causes the application to make an unauthorized request to a back-end SQL server. These are usually possible due to inappropriate input handling and can be difficult to detect [9 p167-169].

2.3. Modern Fuzzers

In the past two decades, numerous new fuzzing methods have been researched and developed further improving fuzzing efficacy [9 p22-25]. This section introduces a few prevalent techniques and fuzzing research in more detail that have been popular or otherwise significant in the field.

2.3.1. FuzzBench, a Fuzzer Benchmarking Platform

As new and improved fuzzing methods are being developed, measuring them is increasingly important. In order to compare and understand how well different fuzzers and fuzzing techniques perform, measuring them is necessary. A study done in 2018 assessed 32 different fuzzing papers and found that "no paper adheres to a sufficiently high standard of evidence to justify general claims of effectiveness" [21]. FuzzBench was created to solve this problem, allowing researchers and developers to evaluate their fuzzers making fuzzing research easier for the community to adopt. FuzzBench is a fully automated, open source, and free service for evaluating fuzzers. It can be used to asses fuzzers on a wide variety of different benchmarks, offers an API for making integrating new fuzzers easier, and has a reporting library. Benchmark reports for each

enrolled fuzzing project are published on FuzzBench web page giving an estimate of how well each performed [22].

2.3.2. Evolutionary Fuzzing

Evolutionary fuzzing is based on a white-box testing technique where evolutionary algorithms inspired by evolutionary biology are used. The basic idea is to have a group of values or members which are then modified or mutated using some genetic algorithm. After a set number of iterations, each member of a group or generation is tested for fitness. At the end of a generation, the groups that performed best in these tests are allowed to breed. After repeated iterations either a solution is found or the subjects converge and do as well as they can. If a solution cannot be found past a certain point, the landscape is described to be flat or deceptive, meaning a solution cannot be intelligently found [9 p219-221].

The idea for evolutionary fuzzing was initially proposed by J. DeMott et al. in the paper "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing". It introduces a fuzzing tool called Evolutionary Fuzzing System (EFS).

EFS uses evolving sessions to learn the target protocol. Sessions are a sequence of input and output that forms the communication with the target. EFS starts with a seed file of data that would be expected to be seen in the target protocol. This is then used to begin the initial attempts at communicating with the target. Code coverage is used to track the fitness of sessions which is in turn used as a metric to breed new sessions. The more accurate the session is, the better the fitness score it will receive. Over time, each new generation will cover more and more of the code. Fuzzing heuristics in mutation are used to keep EFS from learning the target protocol completely correct. These include typical mutations like bit-flipping, long string insertion, format string creation, etc. As EFS iteratively learns the unfamiliar protocol it is implicitly fuzzing the target. Incorrectly performing the different steps the protocol excepts may cause crashes. If this happens, the test case is stored and can be further analyzed later as EFS continues learning the protocol [23].

Some form of evolutionary fuzzing is offered in many fuzzing tools, such as AFL, VUzzer, V-Fuzz, and others [24, 25, 26].

2.3.3. Code Coverage Guided Fuzzing

Code coverage guided fuzzers use code coverage feedback information to improve input mutation. While different coverage-guided fuzzers use different strategies for fuzzing, the general idea is to use code coverage as a metric to improve mutation, allowing the fuzzer to discover and test new code branches. In addition, this allows the fuzzer to reject some inputs that a non-coverage guided fuzzer might generate as invalid [27].

In general, coverage-guided fuzzers use the following workflow:

1. Load initial test case into test case pool.

2. Select seed test cases from the pool with a specific policy.
3. Mutate selected test cases to generate a batch of new test cases.
4. Inject generated test cases to the target and execute them.
5. Monitor target behavior with instrumentation and track code coverage.
6. Filter good test cases that improved code coverage and add them to the pool.
7. Repeat from step 2.

This loop prioritizes test case mutations that improve code coverage. AFL, libFuzer, honggfuzz, and VUzzer, a couple of which will be described in more detail later in this thesis, are some state-of-art coverage-guided fuzzers [28].

2.3.4. In-Memory Fuzzing

In-memory fuzzing refers to fuzzing only a single function or a code block of the target instead of the whole application. This can be implemented in multiple ways, but in general in-memory fuzzing has the following five steps:

1. Target the function that will be fuzzed.
2. Generate a snapshot of the current memory state.
3. Generate input.
4. Execute target until set breakpoint.
5. Restore snapshot and repeat from step 3.

This can be considerably faster than traditional fuzzing where the focus is on injecting faults or anomalies in external inputs and monitoring for failures [9 p161-162]. Fuzzing large or otherwise resource intensive programs can require several seconds of processing before accepting a new input. With in-memory fuzzing, the process can be sped up by taking a snapshot of the target after initialization. With each fuzzing iteration, the memory snapshot is merely loaded again with the input changed and then executed. Many popular fuzzers nowadays offer the option for in-memory fuzzing such as AFL, LibFuzzer, and FairFuzz to name a few [29].

Snapshot fuzzing shares some similar concepts as in-memory fuzzing and overall it seems that the terminology has not yet been fully established. Much like with in-memory fuzzing, snapshot fuzzing is used to save time by using snapshots to skip potentially resource heavy and slow startup routines of a target. This can be done by obtaining a snapshot of the targets state directly before the test case is executed. This way the target can be reset to a deterministic state after each test. Snapshot fuzzing has been used to speed up fuzzing overall and has allowed fuzzers to access some targets that have previously been considered unfuzzable [30].

2.3.5. Machine Learning Applications in Fuzzing

Recent advances in fuzzing have started employing different machine learning techniques to improve the fuzzing process. Machine learning is a technique that allows computer models to perform some operations without explicitly having programmed them to do so. A paper published in 2019 by G. Savedra et al. surveyed different machine learning applications that had been employed in fuzzing so far. They present that machine learning had been most commonly used to generate better inputs for fuzzing and to boost symbolic execution performance. Machine learning had also been used for crash triage and root cause categorization.

The paper summarises that while machine learning has been shown to play an important role in the functionality of fuzzing systems, there still remain many challenges open to research. They expect that different machine learning techniques will be employed in the future to address bottlenecks in the fuzzing process [31].

2.4. Container Technology

This section uses the official Docker documentation as its main source. For a more extensive description on container technology and step-by-step tutorials it is highly recommended [32].

Modern applications are commonly assembled from existing software components and rely on external applications and services. These dependencies can change over time and major differences can break code that relies on them. This problem is commonly referred as "dependency hell" which is one of the issues container technology attempts to solve. Currently one of the most used container implementation is Docker [33]. The Linux Journal from 2014 describes Docker as follows "Docker is a set of platform as a service products that uses operating system level virtualization to deliver software in packages called containers." [34]. Docker is developed by Docker Inc. under the Apache Licence and it is an open source software project that builds atop many established older technologies such as Linux containers, OS virtualization and Git-like versioning to name a few. Docker uses a technology called containers to do this. In contrast to modern virtualization that uses hypervisors to spin up Virtual Machines, Docker containers use existing available protected portions of the operating system. Containers piggyback on running operating systems as their host environment using namespaces to isolate themselves from the rest of the system. This way resource utilisation is more efficient compared to traditional virtualization, which requires a full-fledged operating system to run, regardless of whether the environment is executing a task. Additionally, since containers are merely isolated child processes sharing the same kernel as the host machine, creating and terminating existing containers is fast. The system merely has to terminate processes running in its namespace [32].

2.4.1. Docker Arcitecture

Docker uses a client-server architecture. Running a containerized application using Docker requires four main components:

- Docker client.
- Docker daemon.
- Docker image.
- Docker container.

The Docker client is primarily responsible for handling user interaction with Docker. It accepts commands from the user and sends them to the Docker daemon using the Docker API. It can communicate with more than one daemon. The Docker daemon functions like a typical UNIX daemon. It is a background process that manages various Docker objects such as images, containers, networks and volumes. Docker daemons can be hosted client side or remotely, in which case communication is handled over network sockets. Docker images are read-only templates that contain the instructions for building a container. Typically they are based on a base image, such as Ubuntu, with added steps for installing the necessary dependencies and other components for the user application to work. The images are layered with each additional instruction adding another layer to the image. This way, if a single installed dependency changes you only need to rebuild that layer, not the entire image from scratch. Docker container is an instance of the image that Docker daemon executes. Containers run isolated from other each other or can be interconnected by one or more networks, storage or volumes. A container is defined by its image and can have additional configuration options for when it is created. Containers are stateless and any changes to it that are not stored in persistent storage are lost when the container is removed [32].

Docker Engine API offers an alternative way to interact with the Docker daemon. It is a RESTful API accessible by any HTTP client such as wget or curl. A software development kit (SDK) implementation for the API exists written in Python and Go [35].

2.4.2. Docker Use Cases

Since its creation Docker has become widely accepted among developers for multiple use cases [33]. Using Docker allows for environment standardization. Sharing an application with Docker minimizes the inconsistency between different environments. Portable deployment of applications makes sharing them among different operating systems easier and makes the development environment repeatable. Since Dockerfiles contain the instructions on how the environment is built, it works as a documentation on what components the application requires to function. Additionally, Docker is used in Continuous Integration and Continuous Delivery (CI/CD) workflows, as test environments, for scaling portable workloads, and for version tracking.

Docker containers were an integral part of the CinCan project. They were initially used in the project as part of a malware analysis CI/CD pipelines and later the CinCan command, a command line interface for using containerized tools, was built around them [8].

3. RELATED WORK

This chapter describes some of the topics that were examined more closely related to this thesis. First, fuzzing engines and frameworks that were researched and tested as potential candidates to use are presented. Then, OSS-fuzz, a significant fuzzing project by Google, that had some overlap with this thesis is described. Finally, in section 3.3, other related scientific work that were found during the discovery phase of this thesis is introduced.

3.1. Fuzzing Frameworks

Many freely available fuzzing engines and fuzzing frameworks for finding security vulnerabilities in software exists. Through research was performed to explore and understand the current state of freely available fuzzing tools. Commercial solutions were excluded from consideration for use due to the difficulty in obtaining a licence for them. This section describes some of the more popular fuzzers and fuzzing frameworks that were researched and tested as potential candidates to use for fuzzing in this thesis.

3.1.1. American Fuzzy Lop

AFL is a popular fuzzing framework that employs compile-time instrumentation and genetic algorithms to find bugs in the target. It was one of the first coverage-based fuzzers and has since been widely adopted in fuzzing C / C++ applications. Compared to other instrumented fuzzers, AFL has been designed to be simple, requiring ideally very little configuration to run. Another goal of AFL is to be able to run as many test cases as possible for maximum code coverage and numerous performance optimizations exists to further improve testing speed. AFL works best when the target is compiled with instrumentation that has helper functions which are used to track control flow and improve performance. Black box fuzzing with no instrumentation is also supported but is less effective [24].

3.1.2. Boofuzz

Boofuzz is a successor of Sulley, another popular fuzzing framework, that incorporates all common functionalities that are expected of a modern fuzzer. Its main goal is to be as extensive fuzzing framework as possible. Boofuzz is written in Python, supports most operating systems, and is free and open-source under the GNU General Public License. Special features that Boofuzz offers are built in support for fuzzing over arbitrary communication mediums, support for serial port fuzzing, support for some internet layer fuzzing, and improved recording and output of test data [36].

3.1.3. GitLab Protocol Fuzzer Community Edition

GitLab Protocol Fuzzer, previously known as Peach, is a model-based fuzzer framework. Initially Peach had two versions; Open source Peach Fuzz Community Edition, available free of charge, and a commercial closed source product Peach Fuzzer Professional Edition. The latter was also partially made open source after GitLab acquired it. GitLab Protocol Fuzzer is capable of both generation- and mutation-based fuzzing. It generates test cases from user-defined templates called Peach Pit files. Peach Pit files typically define the structure, type information, and relationships of the data which the framework will then use in the fuzzing process. The fuzzer relies heavily on these templates which at the time of writing unfortunately were not freely available and will later be offered as a commercial product [37].

3.1.4. Atheris

Atheris is a coverage-guided fuzzing engine developed by Google mainly for fuzzing Python code. It is written in Python and C++ and is freely available open source under the Apache 2.0 License. Atheris is based on libFuzzer and can take advantage of some memory error detectors such as AddressSanitizer and MemorySanitizer. Using Atheris is relatively straightforward and at minimum requires just a simple fuzzing harness to be added for the target. Code coverage information is done by bytecode instrumentation and with basic setup any uncaught exception is reported as an error [38].

3.1.5. Honggfuzz

Honggfuzz is another coverage-guided, feedback-driven evolutionary fuzzer developed by Google. It is freely available, open source, and licensed under the Apache 2.0 license. Honggfuzz is a grey-box fuzzing tool with additional support for purely black-box fuzzing if source code is not available. Error detection is done by using POSIX-signal interface and as such, requires a POSIX-based system to run. The biggest advantage Honggfuzz has over other similar fuzzers is that can utilize hardware-based counters and Intels Branch Trace Store and Processor Tracing if supported for improved code coverage tracking. Honggfuzz can automatically identify files from a given corpus that leads to additional code coverage or branches. In-memory storage is used for used files that is managed dynamically to improve performance [17].

3.1.6. Radamsa

Radamsa is a general purpose mutation fuzzer, freely available under the MIT license. The main purpose of Radamsa is to be a simple and easy-to-use robustness testing tool. It tells how well a target can withstand malformed and potentially malicious inputs.

Radamsa is first and foremost a general purpose fuzzer for any kind of data; it requires no information about the target, format or data to work. Radamsa is very

simple to use and once compiled it is a single binary that takes in any kind input and mutates it as specified by optional parameters. Due to its extreme black-box approach, Radamsa is limited in how deep into the tested system it can penetrate compared to other fuzzers implemented with instrumentation. This can be somewhat remedied by pairing Radamsa with coverage instrumentation which can improve the quality of samples during a continuous run. Additionally, due to its nature Radamsa can find cases which pure coverage-based fuzzers might miss.

Radamsa was first developed during the OUSPG's PROTOS project and has since grown into a popular fuzzer used to find numerous bugs from multitude of programs [39].

3.2. OSS-Fuzz

OSS-fuzz is a significant project focusing on improving the stability and security of widely used open source software. OSS-fuzz is led by Google and it is a continuous fuzzing service aimed at open source projects that are widely used or critical to the global IT infrastructure as a whole. It combines modern fuzzing techniques with scalable, distributed execution supporting over 500 open source software at the time of writing. OSS-fuzz uses libFuzzer, AFL++, and Honggfuzz fuzzing engines in combination with most commonly used sanitizers such as AddressSanitizer, MemorySanitizer and more. In addition OSS-fuzz offers an extensive fuzzing infrastructure ClusterFuzz. ClusterFuzz allows scalable fuzzing clusters up to 100,000 virtual machines. While this project is not specifically looking only at open source DFIR/OSINT software, few prominent tools such as Sleuthkit, Radare2, Yara, and others are supported by it [40].

In order to be accepted to OSS-fuzz, the project must be open-source and meet some requirements. The projects under OSS-fuzz are maintained by their main developers. After being accepted, new projects can be added to OSS-fuzz by following common guidelines. The project has to follow a certain file structure and contain at least the following files:

- A yaml file with metadata about the project.
- Dockerfile that defines the container environment with information on necessary dependencies.
- build.sh that build the project inside the Docker container.

Figure 2 shows an overview of the OSS-fuzz process.

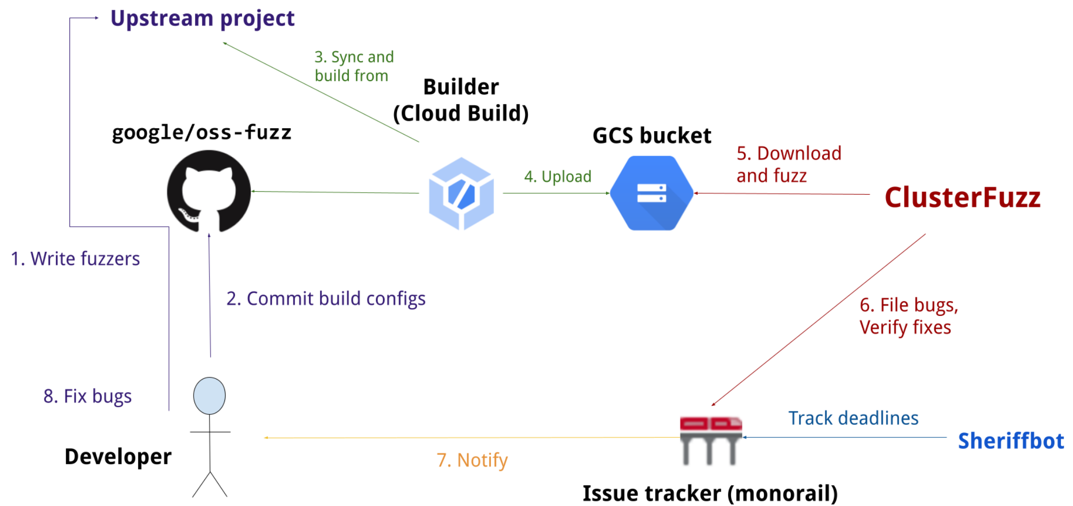

Figure 2. OSS-fuzz overview, source: OSS-fuzz GitHub repository.

Since its inception, OSS-fuzz project has found 30,000 bugs from over 500 open-source projects enrolled in it. So far, 98 of bugs found have a Common Vulnerabilities and Exposures record, meaning they are dangerous memory operations that can threaten data confidentiality or integrity or are otherwise significant to security. Google follows industry standard 90-day public disclosure policy for all found issues [41].

Some overlap exists between fuzzing targets of this thesis and projects enrolled in OSS-fuzz which can be seen in Table 1. OSS-fuzz has found numerous security related vulnerabilities such as Use-After-Free operations and buffer overflows from all reported tools.

Table 1. OSS-fuzz found issues

Project name	Found issues	Security Bugs
sleuthkit	120	46
yara	184	25
radare2	150	41
clamav	392	20

3.3. Fuzzing Digital Forensics and Incident Response Tools

During the discovery phase of this thesis, two papers focusing specifically on fuzz testing digital forensics tools were found.

A paper by A. Case et al. "Gaslight: A comprehensive fuzzing architecture for memory forensics frameworks" presents a modular fuzz-testing architecture for stress-testing both open and closed-source memory forensics frameworks. The framework takes into account critical code paths and mutates given samples accordingly to reveal implementation errors. While the framework only focuses on memory analysis digital forensics tools, it revealed a number of critical previously undiscovered bugs from two popular memory forensics tools Volatility and Rekall [5].

A paper by S. Paruchuri et al. "Gaslight revisited: Efficient and powerful fuzzing of digital forensics tools" focuses on developing a stress testing platform that assesses the robustness and reliability of digital forensics tools. The fuzzing architecture was built atop Gaslight and further improved upon it. The authors developed several new features that expanded the functionality of the original framework to find critical programming errors in any digital forensics tool that ingests filesystem data. The framework was tested against Sleuthkit, a popular open source framework used in disk image forensics, and found many critical programming errors [42].

4. IMPLEMENTATION OF THE FUZZING FRAMEWORK

The first goal of this thesis was to either find or develop a fuzzing framework that allows robustness testing of digital forensics tools that were containerized during the CinCan project in some form. The second goal was to use the fuzzing framework to subject the tools to robustness tests and report any meaningful bug or vulnerability to the author in a secure manner in order to improve the quality of the tool and to contribute to the open source software community in general. The following chapter describes the different steps that were to be taken to implement and build a fuzzing framework for this thesis.

4.1. Overview

The first step of designing the fuzzing framework was to study the fuzzing targets more closely. The goal was to understand what kind of tests the targets are currently subjected to, has any of the work been previously done and do the targets have any useful components ready (e.g. fuzzing harnesses). Since the targets are all open source and freely available, this could be done by studying their git repositories.

4.1.1. Current State of Targets

Software testing of open source digital forensics and incident response tools was found to vary quite a lot. While simply inspecting git repositories do not fully reveal what kind of software testing the code was subjected to, if any, it did give some insight as to how mature the project is. Some projects had robust unit tests covering large amount of code and had been extensively fuzzed either by the authors or by some other party e.g. OSS-fuzz, while others seemed to not have even basic unit tests available. Majority of the projects had at least some documentation available, again following the trend where more mature projects had more extensive documentation available as well.

4.1.2. Rationale behind Building a New Framework

As a target for fuzzing the chosen digital forensics and incident response tools as a whole turned out to be quite challenging as they tend to be rather complex applications consisting of hundreds of thousands of lines of code with all major programming languages represented. After some research it became apparent that while existing fuzzing engines or frameworks could be used to test any single digital forensics application, testing all of them would require significantly more resources than were available for this thesis.

In general, to use an existing solution would have required first to install the tool with all its dependencies to the fuzzing environment, choose an appropriate fuzzer, add or use existing fuzzing harness, preferably build the target with instrumentation and finally start fuzzing. The aforementioned steps would need to be taken for each

individual tool. As such, the decision was made that the best approach would be to build a simple modular framework using existing components.

For this thesis the SUT is viewed to be a black box that is subjected to robustness tests. The goal of the tests is to see how the tool behaves in practice against robust inputs, not to test its particular components or specifications. The main focus for fuzzing will be the parts of the code that a potential attacker could exploit, namely the portion that handles the actual analysis of the input. Since the intention is to test large amount of different tools built with various both high- and low-level programming languages, they are handled as if the source code was not available when designing the tests. This process could also be described to be gray-box testing which has the potential to benefit from best of both worlds; source code is studied only if problems are found to help with root cause analysis.

4.2. Goals for Building the Framework

Upon further research and planning, the following features became important:

- Simplicity. It quickly became apparent that since the fuzzing target is rather extensive and complex, adding new targets to the framework first and foremost needs to be simple. Programming language, input, or output should not limit whether the application can be fuzzed with the framework as long as it is usable via command line.
- Modularity. The framework should be built using existing industry standard components as much as possible to save time and resources.
- Performance. Fuzzing with the framework should be reasonably fast. Support for parallel processing is important to maximize efficacy.
- Detection of error conditions. The framework should at least identify and handle accordingly the following error conditions of the targets:
  - Inelegant crashes of the target.
  - Infinite loops or otherwise excessively long execution.
  - File-system resource exhaustion. Cases where the target starts to use disk space excessively should be handled as an error. The framework itself should not generate samples that exhaust all local disk space.
  - Memory resource exhaustion. The framework should detect if the target consumes an excessive amount of memory and in cases where this happens, it needs to gracefully handle it and return to normal operation.
- Usability. The framework should work via command line so that it can be quickly set up in a cloud server environment and can be scripted.

4.3. Building the Framework

The main components the framework was built atop was Python, Radamsa and Docker. Figure 3 shows an overview of the fuzzing process. In simplified terms, fuzzing was done in six steps:

1. Target is chosen, its container built or downloaded if necessary and then started.
2. Appropriate input is chosen.
3. Input is mutated using Radamsa.
4. Input is injected into the container and executed.
5. Target behavior is monitored.
6. Abnormal behavior is detected and logged.

Steps 1 and 2 are taken at the beginning of fuzzing each SUT. Steps 3-6 are repeated until a given time limit is reached or the framework meets a set limit how many cases are logged. Afterwards, the created containers are terminated and the process is repeated from step 1 against a new target.

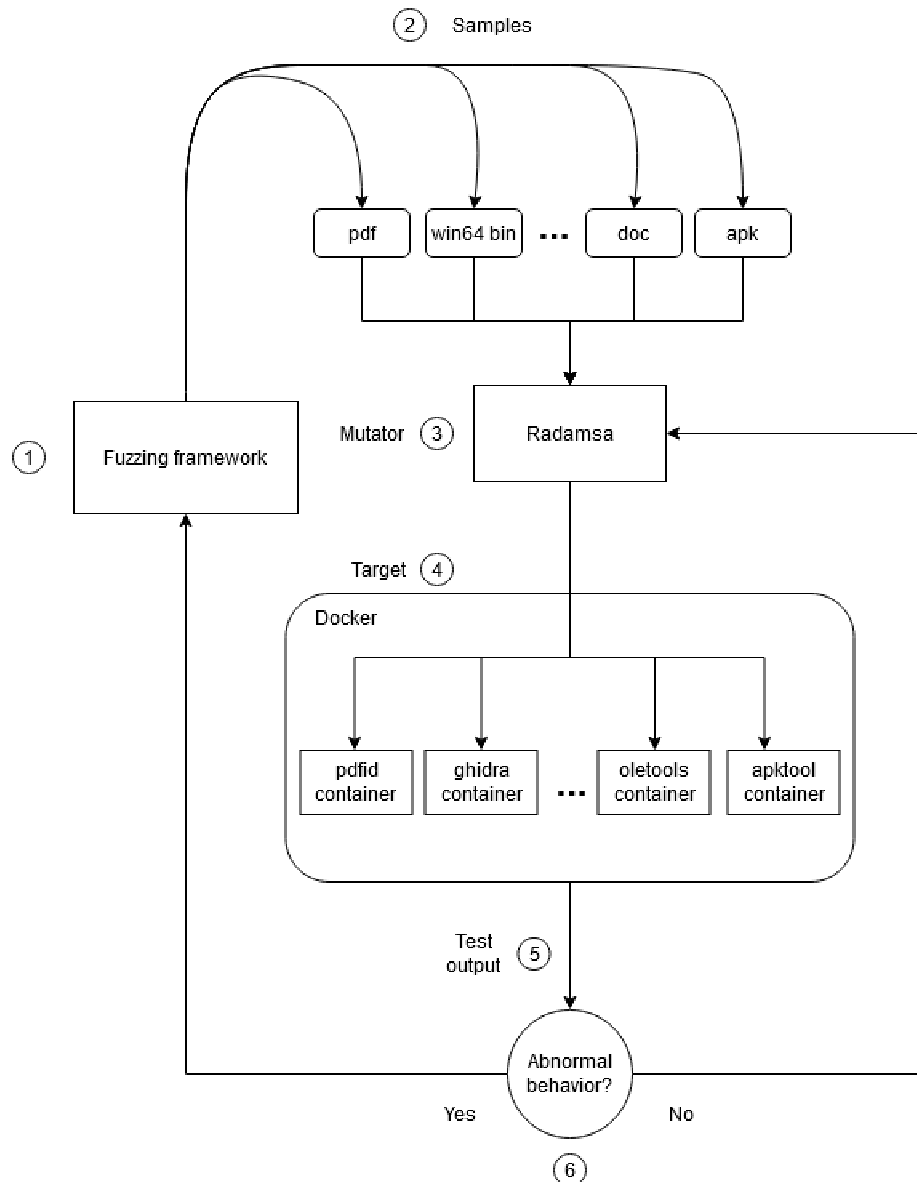

Figure 3. Overview of the framework

In practice when building the fuzzing setup the following problems had to be solved for each tool:

- How to execute the tool?
- What are the tool inputs?
- How to inject inputs?
- How are the inputs generated?
- How are bugs detected?
- How to automate the process?

The following sections describes in more detail how these problems were overcome.

4.3.1. Tool Execution

Tool execution was built on top of Docker. Each tool has its own Docker container which works as an isolated environment with the necessary dependencies installed. While this introduces an additional complexity layer to the setup and slightly slows down execution it was deemed that the benefits outweigh the cons. Installing and confirming that the tools work each time a fuzzing environment is booted up would take a considerable amount of time. By using containers one can both confirm that the tool should work as expected in the future and relatively easily run the fuzzing setup in a distributed cloud environment as well. Using containers makes both managing and scaling the environment easier as well; if the SUT hangs or crashes it merely slows down or stops its own container without necessarily affecting the rest of the setup.

Other OS-level virtualization technologies or full fledged virtualization could also have been used to achieve similar goals. However, since appropriate Dockerfiles for each SUT were already available and Docker had well documented and relatively lightweight Python SDK, it was ultimately chosen as the technology to use.

4.3.2. Tool Inputs

Input for each tool was copied from CinCan Tools Gitlab repository which had a test sample for most of the stable tools that were used in the project. These samples typically are files that the tools would be given for analysis, e.g. sample for the tool Volatility that is used for memory forensics would be a small Windows memory dump in .raw format. This amounts to 40 different samples in total available for fuzzing. Few samples that were not available or not suitable for fuzzing in the CinCan Tools repository were either self generated or found online.

4.3.3. Injecting Inputs

In general, it was viewed that there would be three different feasible methods for injecting inputs into the target:

- STDIN&STDOUT hook.
- using Docker SDK put_archive method.
- using Docker volume mount.

The initial idea was to simply hook STDIN and STDOUT of the container that is executing the tool and pass the mutated input through the STDIN hook. The assumption was that this would improve performance and would be the simplest approach. While this mostly worked with tools that were tested, issues arise when running the code parallel and occasionally with storing some inputs. Upon further testing, this method was deemed both more difficult to implement and not considerably faster since in the end most of the time processing the input was done by the tool itself,

not input injection. Adding new tools introduced further challenges that would have to be solved in order for this method to work properly.

The second approach was to store the mutated input into a file and then use the Docker Python SDK method put_archive to copy the file into the container file system which then could be used by the SUT. This method in theory should have been slightly slower as STDIN hooks use the low-level API and is less complex as a whole, but upon testing the impact to performance again seemed negligible.

Third approach was to use Docker Python SDK method create_volume to mount a folder from the host machine to the container which could then be used to both store and pass the inputs to the SUT. Both methods create_volume and put_archive were similar in performance and had pros and cons. Mounting a folder allowed for test cases remain stored in cases where the container stopped responding or was terminated due to a fatal error in the Docker engine. Copying each individual file into the container using put_archive would require an additional read/write step as compared to volume mount. Using volume mounts were ultimately chosen as the preferred method as it was the simplest to implement and worked best with parallel processing.

4.3.4. Input Generation

Input generation was done by mutating valid samples using Radamsa. The main motivation for choosing Radamsa as the file mutator over other methods was its usability and low software overhead with good performance. Radamsa can accept any kind of data for mutation and is simple to use [39].

4.3.5. Detecting Abnormal Behavior or Crashes

Crashes were detected by relying on the exit codes of the tests. Docker follows the chroot standard [43] with exit codes as shown in Table 2.

Table 2. Table of chroot exit codes and their meanings.

Exit Code Number	**Meaning**
1	Catchall for general errors
2	Misuse of shell builtins
126	Command invoked cannot execute
128	"command not found"
128+n	Invalid argument to exit
130	Script terminated by Control-C
255	Exit status out of range

Abnormal behaviour was determined based target output and on the different characteristics these actions produce. Each fuzz case had limited execution time based on empirical observations how long the tool typically takes to analyze an input. If this set execution time was exceeded the test is terminated by the framework and fuzz case with possible output is stored. Each individual container were manually limited

to have 4 gigabytes of memory and if this memory is exceeded by the test case the test is again terminated by the framework with the relevant information stored.

4.3.6. Storing Results

By default, the framework stores any test case that exit code is not 0 or 1 and marks them as critical. Only some of the test cases where the executed command returns 1 are stored, the reasons for this are further elaborated on the discussion section of this chapter. These cases are marked as non-critical to differentiate them from other results.

First time when a test case is stored, a folder with the targets name is created under fuzz_results folder. The created folder contains the input file that caused the error and a log file containing the targets output and the exit code. A mention of this is also added to a log file that keeps track of the frameworks overall progress.

4.3.7. Process Automation

The fuzzing framework instrumentation and automation was built using Python 3. The framework contains a functional testing system that handles the logic flow of the fuzzing process as described in Figure 3. The built instrumentation is responsible for overall control of the the target. It sets up a new fuzzing environment for each target and starts and restarts them as needed. Python programming language was chosen as it had many existing modules available reducing the amount of code that needed to be written. Docker SDK has native support for Python 3 and Pyradamsa, a Python Library implementing mutation engine Radamsa, was available for Python as well [44]. Parallelization was added using Python native multiprocessing library [45].

4.4. Adding a Target to the Framework

To add a new target to the framework certain conditions need to be met.

- The target either needs to be a command line based application or a script needs to be added that allows it to be accessed via command line.
- The target needs to be non-interactive or portions of the application that require user interaction needs to be automated.
- The target has to be containerized with all the necessary dependencies installed. The image can be built locally or needs to be publicly accessible through Dockerhub or other similar container hosting platforms.
- The framework requires an input sample for mutation. All the necessary samples needed for this thesis come with the framework.

After these conditions are met, the target can be added to the framework. The framework requires three parameters for fuzzing: the target image name, command that executes the target and a relevant sample. Optionally, parameter timeout can be

set which indicates how long a single test case can take to execute before it is deemed an error. Example targets can be seen in Figure 4.

```
"iocextract":["timeout %s iocextract --input"%timeout, "pdf_ioc"],
"jadx":["timeout -t %s ./bin/jadx"%timeout, "jar"],
"jd-cli":["timeout %s java -jar /usr/local/bin/jd-cli.jar -od decompiled"%timeout, "jar"],
"jsunpack-n":["timeout %s jsunpackn.py -c /jsunpack-n/options.config -V"%timeout, "pdf_javascript"],
"manalyze":["timeout %s manalyze"%timeout, "exe"],
"oledump":["timeout %s /usr/local/bin/python /oledump/oledump.py -S -s 3"%timeout, "docm"],
```

Figure 4. Example commands

After adding a new target to the framework it is recommended to confirm its functionality in debug mode which prints STDOUT&STDERR streams and exit code of each fuzz iteration for 10 seconds. Figure 5 shows an example fuzz test case log from fuzzing tool oletools.

```
ExecResult(exit_code=8, output=b'ERROR    Unhandled exception in main: \'NoneType\'
 object has no attribute \'name\'
Traceback (most recent call last):
    File "/home/appuser/.local/lib/python3.9/site-packages/oletools/olevba.py",
    line 4668, in main
    curr_return_code = process_file(filename, data, container, options)
    File "/home/appuser/.local/lib/python3.9/site-packages/oletools/olevba.py",
    line 4477, in process_file
    vba_parser = VBA_Parser_CLI(filename, data=data, container=container,
    File "/home/appuser/.local/lib/python3.9/site-packages/oletools/olevba.py",
    line 4029, in __init__
    super(VBA_Parser_CLI, self).__init__(*args, **kwargs)
    File "/home/appuser/.local/lib/python3.9/site-packages/oletools/olevba.py",
    line 2770, in __init__
    self.open_ppt()\n  File "/home/appuser/.local/lib/python3.9/site-packages/oletools/olevba.py",
    line 3107, in open_ppt
    ppt = ppt_parser.PptParser(self.ole_file, fast_fail=True)
    File "/home/appuser/.local/lib/python3.9/site-packages/oletools/ppt_parser.py",
    line 1198, in __init__
    root_streams = self.ole.listdir()
    File "/home/appuser/.local/lib/python3.9/site-packages/olefile/olefile.py",
    line 1853, in listdir
    self._list(files, [], self.root, streams, storages)
    File "/home/appuser/.local/lib/python3.9/site-packages/olefile/olefile.py",
    line 1825, in _list
    prefix = prefix + [node.name]\nAttributeError: \'NoneType\' object has no attribute
    \'name\'\nolevba 0.60 on Python 3.9.6 - http://decalage.info/python/oletools\n')
```

Figure 5. Tool oletools stacktrace with exit code.

4.5. Problems and Challenges

Initially the framework was set to store any test case which exit code was not 0, meaning the command did not execute successfully. Upon further testing, it quickly became apparent that this would result in excessive amounts of data that would be of little interest. However, flat out refusing any such test cases would also not be ideal as this might lead to missing some relevant information. By default, the framework was set to store some of the test cases where the exit code was 1, and all of the cases where they were not either 0 or 1. Ultimately relying on just exit codes is not ideal for error detection as some programs might suppress serious flaws intentionally or accidentally by having a catch-all error handling in the programs logic flow. Without proper instrumentation memory related issues are more difficult to detect and estimating how much of the code is covered becomes challenging.

Differences in access rights between Dockerfiles caused some unexpected issues when fuzzing specific tools. By default, all CinCan containers have a dedicated user

appuser with limited access rights that executes any given command. Depending on what Linux flavour the host operating system is running surprisingly affected what file permissions a fuzz case had that the containerized tool would access. Undocumented differences in functionality were found between Manjaro and Ubuntu Linux systems. This could be partially remedied by running each tool as root user with unlimited access rights, but this in turn would break some installations. In the end directly specifying file rights to each generated case had to be done which in theory might slightly negatively affect performance by adding additional steps to the process.

Some issues with Docker stability arise when executing longer test runs with the framework. Occasionally the framework would terminate execution due to Docker engine responding with Internal Server Error specifying temporary failure in name resolution. The root cause for this issue remained unclear and recommendations online for similar issues suggested fixing it with merely restarting Docker engine. Additionally, few cases where the Docker engine would hang execution unrelated to the target being fuzzed were observed. These issues could partially be remedied by automatically catching these errors during execution, restarting Docker engine and resuming fuzzing but some uncommon issues still remain unresolved.

5. FUZZING TESTS

This chapter introduces the fuzz tests that were done with the framework. First, the test setup is described along with the hardware that was used. Afterwards, results of the tests are presented and analyzed and finally some of the challenges that arise during testing are discussed.

5.1. Fuzz Test Setup

The main purpose for fuzzing in this thesis was to test how well selected open source digital forensics and incident response tools can handle robust inputs. Running these tests also gave an indication how well the built framework works over longer period of time. Each tool was fuzzed for 3 hours which resulted in the tests taking roughly 6 days in total. This specific duration was chosen to limit testing period of this thesis under one week as it was expected that the tests need to be repeated as unexpected issues arise. Ideally, each SUT would be fuzzed at least for days or weeks, but with the set time constraints of this thesis this could not feasibly be achieved.

Most of the tools had multiple operating modes with numerous parameters changing how it functions and what libraries it uses. Again, due to limited resources only what was observed to be the main function of each tool was tested.

Table 3 shows all the different file-formats that were used for mutation.

Table 3. List of used file-format extensions for mutation.

.apk	.bin	.csv	.dd
.elf	.doc	.eml	.exe
.html	.jar	.log	.pcap
.pdf	.png	.raw	.txt

5.1.1. Hardware

The fuzzing efforts were performed on a cloud server offered by Center of Science (CSC). CSC is a Finnish state owned company offering cloud computing and other ICT services for higher education, research institutes, culture, public administration and enterprises [46]. Hardware resources available for fuzzing were 16 virtual CPUs, 32 gigabytes of ram and 1 terabyte of disk space. Performance of the framework is related directly to the speed of the processor and the number of cores available on the test system.

5.1.2. Target

The fuzzing targets were chosen based on digital forensics and incident response tools that were containerized during the CinCan project. The tools were gathered from questionnaires and interviews that were conducted during the project. One of the goals of the project was to recognise and document open source digital forensics and

incident response tools which are widely used and trusted among incident response professionals. This led to the containerization of over 70 tools, out of which 46 were categorised stable. Tools were considered stable if they had a working container, pass some basic unit tests and and had a brief description on how to use them and on what [8].

Stable tools were then chosen as the fuzzing targets for this thesis since they have been most recently confirmed to be in working condition and as such would require less work to install and prepare for fuzzing. Out of the 46 stable tools, few were dismissed from fuzzing based on how the tool works. As an example tool virustotal, a containerized client that uses the Virustotal API to query the database, was dismissed as a target since the way this tool would be fuzzed, would send an unreasonable amount of requests to the Virustotal service. In total, this resulted in 43 tools that were the target for fuzzing. Table 4 contains a list of all the targets that were fuzzed with a short descriptions on what the tool does, language it was written in and files it is commonly used for.

Table 4. List of fuzzing targets and their description.

Target description			
Tool Name	Language	Description	Input file
clamav	C	Antivirus engine	Any file or directory
osslsigncode	C	Authenticode sign for binaries	.exe, .dll, others
radare2	C	Reverse engineering and binary analysis	Variety of different binary architectures
sleuthkit fsstat	C	Fetch file system information	.raw, .ewf, .vmdk, .vhd
sleuthkit fls	C	Read the files and directories of the file system	.raw, .ewf, .vmdk, .vhd
sleuthkit rip	C	Identify recently accessed documents and devices	.raw, .ewf, .vmdk, .vhd
ssdeep	C	Calculate CTPH of file	Any file
tshark	C	PCAP parser	.pcap, other network traffic files
yara	C	Match patterns in files	Any file
manalyze	C++	static analyzer for PE executables	PE files
snowman-decompile	C++	C/C++ decompiler	.c, .ccp, others
7zip	C++	compression software	.7z, .zip, gzip, others
ilspy	C#	Dotnet assembly decompiler, PDB generator	.net
apktool	Java	Decompile .apk files	.apk
cfr	Java	Java Decompiler	.jar
dex2jar	Java	decompile dex files	.apk
ghidra-decompiler	Java	Reverse Engineering Framework	Any native instruction binary
jadx	Java	Decompile dex & apk files into Java source	.dex, .apk
jd-cli	Java	Decompile dex & jar	.dex, .jar
regripper	Perl	Extract data from Windows registry	.xz
xsv	Rust	Manipulate CSV files	.csv, .tsv

Target description			
Tool Name	Language	Description	Input file
access-log-visualization	Python	Visualize webserver access logs	.log
binwalk	Python	Analyze and extract files	Any file
eml_parser	Python	Eml parser	.eml
flawfinder	Python	source code analyzer	C/C++ source files
floss	Python	Obfuscated string solver	.exe
ioc_strings	Python	Extract ioc strings from file	Any file/directory
iocextract	Python	Extract iocs from files	Any file
jsunpack-n	Python	Emulate browser to detect exploits	javascript, .pdf, .pcap, others
oledump	Python	OLE file analyzer	.doc, .xls, ppt, others
oletools	Python	Analyze Microsoft OLE2 files	.dot, dot, .xls, others
output-standardizer	Python	Generate markdown report from input	.html
pdf-parser	Python	Deconstruct PDF to elements	.pdf
pdfid	Python	Scan PDFs for keywords	.pdf
pdfxray-lite	Python	Analyze PDF files	.pdf
peepdf	Python	Analyze PDF files	.pdf
peframe	Python	PE file static analysis tool	.exe
pyocr	Python	Optical character recognition	.pdf, .png, jpg
ssdc	Python	Cluster files into a tar file	Any file
ViperMonkey	Python	VBA Emulation engine	.doc, xml, pptm, others
pastelyzer	Lisp	Find security artifacts from text	.txt
zsteg	Ruby	Detect stegano-hidden data in images	.png .bmp

5.2. Results

Tests were successfully completed with the developed fuzzing framework with no major issues. No critical security related programming errors were found from the targets, which was understandable due to the relatively short duration of fuzzing and limited scope. With most of the tools having been written in higher level languages such as Python or Java, the expectation was that relevant issues found would be memory leaks, infinite loops, errors leading to the tool being unable to analyze a sample, and other similar issues.

Issues were found from 24 tools out of 43 targets. A list of these tools can be seen in the Table 5 below. Some tests uncovered cases where the tools did not have proper error handling and crashed with malformed inputs. Some tools, such as tshark and sleuthkit, had proper error handling and appropriately detected malformed or otherwise unknown inputs, but exited with non-zero exit code causing the framework to report them as false positives. Real life incident cases where a tool would be given a completely malformed sample is not that likely and as such was not deemed relevant enough to report to the authors. However, better error handling could improve tool stability and can be important for example in an automated analysis pipeline. Completely malformed files could still also be intentionally left behind as digital evidence to break down forensics tools.

Table 5. Found issues.

Tool	**Language**	**General error descriptions**
osslsigncode	C	Issues handling malformed pe files.
ilspy	C#	Issues handling malformed pe files.
apktool	Java	Issues handling malformed apk files.
ghidra-decompiler	Java	Multiple issues analyzing malformed files. Issues with HeadlessAnalyzer analyzing valid files.
jadx	Java	Multiple issues handling malformed jar files.
jd-cli	Java	Multiple issues handling malformed jar files.
access-log-visualization	Python	Excessively long execution.
eml_parser	Python	Issues handling malformed eml files.
oledump	Python	Extension zipfile unable to extract malformed but valid doc files.
oletools	Python	Issues analyzing malformed doc files.
output-standardizer	Python	Issues handling malformed files.
pdf-parser	Python	Issues handling malformed pdf files.
peframe	Python	Few issues handling malformed pdf files.
pyocr	Python	Used extension Wand unable to open malformed but valid images.
ViperMonkey	Python	Issues handling malformed doc files.
zsteg	Ruby	Multiple issues handling malformed png files.

After manually browsing through the data, out of the 24 tools 8 had issues that were deemed important and were then reported to the tools respective authors either via email or by creating an issue on their git repositories. Table 6 shows a list of reported

issues and a short description on what type of an error was in question. Couple issues not listed here with how the tools had been containerized was found and reported as well.

Table 6. Reported issues.

Tool	Language	Bug Description
manalyze	C++	1. Infinite Loop.
jsunpack-n	Python	1. Parsing Error.
peepdf	Python	1. Parsing Error.
pdf-parser	Python	1. Parsing Error.
pdfxray	Python	1. Memory Leak. 2. Infinite loop.
ViperMonkey	Python	1. Parsing Error.
zsteg	Ruby	1. Runtime Error. 2. Memory Leak.

As can be seen, various programming errors were uncovered:

- Peepdf had couple cases where a slight change to the structure of a pdf file would render the tool unable to analyze the input while still be readable with common pdf readers.
- ViperMonkey had a case where a slightly malformed, but readable doc file would cause the tool to crash and not produce any output. Additionally, this error led the official ViperMonkey Docker image to crash with a segmentation fault. Overall the tool appeared to have difficulty handling malformed inputs.
- Few issues were found from the tool zsteg where an openable png could not be read leading to a runtime error. An issue how the tool handles large or invalid IHDR chunks in png format structure was found causing it to consume all available memory and excessively long execution. Overall the tool had difficulties handling malformed inputs.
- A case was found from the tool Manalyze where an input would lead it to what appeared to be infinitely loop during analysis and produce a repeating log until disk space runs out. However, overall the tool handled malformed inputs well.
- Tool pdf-parser had a few cases where a malformed, but readable pdf could not be analyzed. Overall the tool had difficulty handling malformed inputs.
- Multiple cases from the tool pdfxray were found where a malformed, but readable pdf would cause the tool to consume all available memory from the system and infinitely loop during an analysis. It appears that the tool is not currently under active development and overall has difficulty handling malformed pdf files.
- Tool jsunpack-n had a case where it could not read a valid pdf file. Overall the tool handled robust inputs well.

Malicious actor could use these issues to obfuscate a file from being analyzed, slow down incident response or break automation while still be executable on a victim's system. Since at the time of writing these issues were reported rather recently, a more in depth description of them will not be given just to be on the safe side and give the authors of the tools enough time evaluate the severity of the issue and fix them if necessary.

5.3. Problems and Challenges

Closer analysis of results showed that some test cases did not penetrate very deep into the targets. Some tests broke the tools in its initial steps that are not directly related to analysis. For example, a tool called Oledump starts off analysis by attempting to extract the given file and break it down to its components using a native Python extension zipfile. Majority of found issues were related to the extension zipfile not being able to extract the file in some cases if it is malformed and not in the tool itself. Without improved instrumentation, code coverage is more difficult to estimate.

During longer test runs of the framework, couple cases occurred where the system running the tests stopped responding and became unreachable even with SSH. This required a hard boot of the instance through CSC server management interface to resolve. This issue was suspected to be on the server side, unrelated to the tests as they coincided with CSC maintenance which should not have affected server stability.

6. DISCUSSION

As the importance of IT infrastructure grows so does the need for cyber security professionals. DIFR&OSINT tools are increasingly important as practitioners in the field are responsible for detecting and responding to growing number of hostile actions against any systems. During the discovery phase of this thesis it was found that not many scientific studies had looked at the current state of digital forensics tools and specifically fuzzing had not been widely done. Outside of the scientific community as the importance of open source software has grown, so has the interest and resources for keeping them up-to-date and in working order. Major efforts targeting security of open source software have been taken by large companies as the issue will become more prevalent as their use spreads. A good example of this is Google's OSS-fuzz project.

Discovery phase and the results of this thesis show that some digital forensics and incident response tools do suffer from robustness issues. These range from trivial programming errors to severe security issues. The experiments in this thesis were limited in resources and just 3 hours of testing per target is in general considered a short time to fuzz any program. Additionally hardware resources for the tests were also relatively limited. Still, issues were found in several programs and while these tests do not give a definite answer about the current state of the tools, it would appear that more mature, up-to-date tools handle robust inputs better.

Whether robustness issues are an actual challenge in real life incident response can vary case by case. Obfuscating a malicious PDF file so that it cannot be easily read with common analysis tools is merely an annoyance and can be bypassed by reading the file manually. At the same time, heavily obfuscated PE files can require a skilled reverse engineer to deobfuscate. Especially with digital forensics tools written in low-level languages, major security issues could at worst lead to arbitrary code execution on the analysis system. Previous scientific work and fuzzing efforts by Google's OSS-fuzz show that digital forensics and incident response tools do suffer from these types of severe security issues.

Some program authors have described obfuscation to be an endless cat-and-mouse game with almost limitless possibilities how to evade detection. An argument can be made that open source developers time could be better spent developing overall stability and usability of the program, instead of attempting to catch every edge-case how a file could be obfuscated from analysis.

The developed fuzzing framework successfully did what it was designed to do. The main goal was to perform at least entry level fuzz testing against all relevant targets and be quick and easy to use. Even though the framework does not employ many of the state of the art methods generally present in modern fuzzing frameworks, the results show that some fuzzing is still better than no fuzzing at all. If time and resources allow for it, adding fuzzing to each projects testing arsenal could be used to improve how well the tools handle robust inputs and general functionality overall.

6.1. Further Work

The framework in its current state is closer to a proof of concept than a modern full-fledged framework. It can be used to very quickly fuzz test any containerized command

line based tool, however numerous steps could still be taken to further improve the framework and the test results.

- First and foremost better instrumentation would lead to improved error detection. For example, building each target with AddressSanitizer would enable the framework to better detect memory related issues among other things. Improved instrumentation would allow for tracing which parts of the code are actually reached giving an indication of code coverage.
- The framework currently can only use one input for mutation. Common recommendation for fuzzing is to use as many valid samples for mutation as is available which has been shown to improve results.
- Framework performance could likely be improved by minimizing steps that need to be taken for each fuzz case. More through analysis of its performance might find issues that were missed.
- Currently what was observed to be each tools main function was added as the target. Testing more functions with different parameters would likely find new issues that the current tests miss and improve code coverage.
- Repeating the whole fuzzing process with more computing resources and a longer test time for each tool would likely give a better indication of their current state.

7. CONCLUSION

The increased prevalence of cybercrime and attacks has led to a significant increase in the relevance of incident response. Digital forensics and incident response software are commonly used day-to-day tools for many practitioners in the field for detecting, responding, and documenting hostile actions against organisations and systems.

The goal of this thesis was to robustness test open source digital forensics and incident response tools containerized during the CinCan project and study the results. A brief introduction to current popular open source fuzzing frameworks and their capabilities that were researched as a part of this thesis is given. The gathered information was then used decide whether to build a new framework or use an existing one. Ultimately a new fuzzing framework was built capable of fuzzing any containerized command line tool.

The framework was built using Python 3 with the help of Docker Python SDK and Radamsa, a popular general purpose fuzzer. The framework allows for quick and easy fuzzing of any containerized command line based application and comes with samples for mutation used by many popular digital forensics and incident response tools. The developed fuzzing framework was tested against 43 containerized open source digital forensics and incident response tools to see how well they handle robust inputs. The framework successfully remained running for the duration of the tests with no major issues. Out of 43 tested tools 24 had potential issues. Most critical issues were disclosed to the respective tools authors via their preferred communication method. Additionally, couple issues with how the tools had been containerized were uncovered by fuzzing as a result.

As a main outcome of this thesis, a quick and easy to use fuzzing framework was designed and implemented. The fuzzing results show that currently many open source digital forensics and incident response tools suffer from robustness issues.

8. REFERENCES

[1] Cybercrime and cybersecurity statistics. URL: https://www.comparitech.com/vpn/cybersecurity-cyber-crime-statistics-facts-trends/, accessed: 2022-05-03.

[2] Johansen G. (2017) Digital Forensics and Incident Response. Packt Publishing Ltd, 8-10 p.

[3] Ventures C. (2017) Cybersecurity jobs report. Herjavec Group .

[4] Sommer P. (2010) Forensic science standards in fast-changing environments. Science Justice 50, pp. 12–17. URL: https://www.sciencedirect.com/science/article/pii/S1355030609001786, special Issue: 5th Triennial Conference of the European Academy of Forensic Science.

[5] Case A., Das A.K., Park S.J., Ramanujam J.R. & Richard III G.G. (2017) Gaslight: A comprehensive fuzzing architecture for memory forensics frameworks. Digital Investigation 22, pp. S86–S93.

[6] Hibshi H., Vidas T. & Cranor L. (2011) Usability of forensics tools: a user study. In: 2011 Sixth International Conference on IT Security Incident Management and IT Forensics, IEEE, pp. 81–91.

[7] Oehlert P. (2005) Violating assumptions with fuzzing. IEEE Security & Privacy 3, pp. 58–62.

[8] The cincan project. URL: https://cincan.io/, accessed: 2022-05-03.

[9] Takanen A., Demott J.D., Miller C. & Kettunen A. (2018) Fuzzing for software security testing and quality assurance. Artech House.

[10] (1994) Ieee guide for software verification and validation plans. IEEE Std 1059-1993 , pp. 1–87.

[11] Myers G.J., Sandler C. & Badgett T. (2011) The art of software testing. John Wiley & Sons.

[12] Sutton M., Greene A. & Amini P. (2007) Fuzzing: brute force vulnerability discovery. Pearson Education.

[13] Miller B., The fuzz generator. https://fuzzinginfo.files.wordpress.com/2012/05/cs736-projects-f1988.pdf. Accessed: 2021-08-20.

[14] Miller B.P., Fredriksen L. & So B. (1990) An empirical study of the reliability of unix utilities. Communications of the ACM 33, pp. 32–44.

[15] Kaksonen R. (2001) A functional method for assessing protocol. implementation security: Licentiate thesis .

[16] Defensics fuzz tester. URL: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html, accessed: 2022-01-05.

[17] Honggfuzz. URL: https://github.com/google/honggfuzz, accessed: 2022-22-02.

[18] libfuzzer. URL: https://llvm.org/docs/LibFuzzer.html, accessed: 2022-01-27.

[19] Wlan under fuzzing with defensics. URL: https://www.synopsys.com/blogs/software-security/wlan-fuzzing-defensics/, accessed: 2022-02-02.

[20] Karhumaa M. (2021) Bluetooth low energy link layer injection .

[21] Klees G., Ruef A., Cooper B., Wei S. & Hicks M. (2018) Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, Association for Computing Machinery, New York, NY, USA, p. 2123–2138. URL: https://doi.org/10.1145/3243734.3243804.

[22] Metzman J., Szekeres L., Maurice Romain Simon L., Trevelin Sprabery R. & Arya A. (2021) FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, p. 1393–1403. URL: https://doi.org/10.1145/3468264.3473932.

[23] DeMott J., Enbody R. & Punch W.F. (2007) Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. BlackHat and Defcon .

[24] american fuzzy lop. URL: https://github.com/google/AFL, accessed: 2022-22-02.

[25] Li Y., Ji S., Lyu C., Chen Y., Chen J., Gu Q., Wu C. & Beyah R. (2022) V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. IEEE Transactions on Cybernetics 52, pp. 3745–3756.

[26] Rawat S., Jain V., Kumar A., Cojocar L., Giuffrida C. & Bos H. (2017) Vuzzer: Application-aware evolutionary fuzzing. In: NDSS, vol. 17, vol. 17, pp. 1–14.

[27] Lou B. & Song J. (2020) A study on using code coverage information extracted from binary to guide fuzzing. International Journal of Computer Science and Security (IJCSS) 14, pp. 200–210.

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

[29] Manès V.J., Han H., Han C., Cha S.K., Egele M., Schwartz E.J. & Woo M. (2019) The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering 47, pp. 2312–2331.

[30] Schumilo S., Aschermann C., Jemmett A., Abbasi A. & Holz T. (2022) Nyx-net: network fuzzing with incremental snapshots. In: Proceedings of the Seventeenth European Conference on Computer Systems, pp. 166–180.

[31] Saavedra G.J., Rodhouse K.N., Dunlavy D.M. & Kegelmeyer P.W. (2019) A review of machine learning applications in fuzzing. arXiv preprint arXiv:1906.11133 .

[32] Docker documentation. URL: `https://docs.docker.com`, accessed: 2021-06-27.

[33] Overflow S., Stack Overflow developer survey 2021. URL: `https://insights.stackoverflow.com/survey/2021/`.

[34] Merkel D. (2014) Docker: lightweight linux containers for consistent development and deployment. Linux journal 2014, p. 2.

[35] Develop with docker engine sdks. URL: `https://docs.docker.com/engine/api/sdk/`, accessed: 2022-02-05.

[36] boofuzz: Network protocol fuzzing for humans. URL: `https://github.com/jtpereyda/boofuzz`, accessed: 2022-22-02.

[37] Gitlab protocol fuzzer community edition. URL: `https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce`, accessed: 2022-22-02.

[38] Atheris: A coverage-guided, native python fuzzer. URL: `https://github.com/google/atheris`, accessed: 2022-22-02.

[39] Radamsa. URL: `https://gitlab.com/akihe/radamsa`, accessed: 2022-22-02.

[40] Oss-fuzz: Continuous fuzzing for open source software. URL: `https://github.com/google/oss-fuzz`, accessed: 2022-22-02.

[41] Ding Z.Y. & Le Goues C. (2021) An empirical study of oss-fuzz bugs. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 131–142.

[42] Paruchuri S., Case A. & Richard III G.G. (2020) Gaslight revisited: Efficient and powerful fuzzing of digital forensics tools. Computers & Security 97, p. 101986.

[43] Exit codes with special meanings. URL: `https://tldp.org/LDP/abs/html/exitcodes.html`, accessed: 2022-05-03.

[44] pyradamsa. URL: `https://pypi.org/project/pyradamsa/`, accessed: 2022-05-18.

[45] multiprocessing. URL: https://docs.python.org/3/library/multiprocessing.html, accessed: 2022-04-13.

[46] Ict solutions for brilliant minds. URL: https://www.csc.fi/tietoa-meista, accessed: 2022-05-03.