UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

# MASTER THESIS

## OpenCRS: Attack Surface Approximation, Vulnerabilities Discovery, and Automatic Exploiting of Binaries

George-Andrei Iosif

**Thesis Advisors:**

Adrian-Răzvan Deaconescu
Constantin-Eduard Stăniloiu

**BUCHAREST**
2023

# ACKNOWLEDGEMENTS

I would like to thank everyone who contributed to the creation of this thesis.

Adrian-Răzvan Deaconescu and Constantin-Eduard Stăniloiu, the thesis coordinators, were available for regular support meetings, which helped me and other colleagues working on OpenCRS define our theses and the open source project.

This thesis' peer reviews were provided by Andreia-Irina Ocănoaia and Amir Naseredini. Their suggestions were extremely helpful and most of them have been incorporated into the current version of the text.

Finally, much like during the bachelor's studies, Iulian was available to offer helpful guidance based on his personal experience.

## ABSTRACT

The objective of this thesis is to introduce four distinct modules of OpenCRS. The latter is a novel open source cyber reasoning system (CRS), which is an automated collection of software components designed to identify, exploit, and patch vulnerabilities in an automated fashion. Despite the passage of seven years since the completion of Cyber Grand Challenge, a competition organized by DARPA that introduced this family of automated security systems, the open source CRS landscape has exhibited a lack of change. The finalist Shellphish continues to be regarded as the prevailing state-of-the-art solution in this domain. The thesis introduces modules that incorporate the technological advancements achieved between 2016 and the present time. These modules include a dataset containing public test suites of vulnerable programs, a module designed to approximate the attack surface of an executable, a vulnerability detection module that utilizes fuzzing techniques, and a final module focused on automated exploitation. The approach taken in each module will be methodical, involving the definition of employed technologies, an in-depth explanation of the software architecture and its functioning, and the presentation of tests and evaluations.

## SINOPSIS

Obiectivul acestei teze este de a introduce patru module ale OpenCRS, un nou sistem de raționament cibernetic (CRS) cu sursă deschisă. Un CRS este o colecție de componente software proiectate pentru a identifica, exploata și mitiga vulnerabilități, totul într-o manieră automatizată. În ciuda trecerii a șapte ani de la Cyber Grand Challenge, o competiție organizată de DARPA ce a introdus această familie de sisteme de securitate, contextul CRS-urilor cu sursă deschisă a rămas neschimbat, cu finalistul Shellphish considerat încă cel mai evoluat. Această teză introduce module ce încorporează avansul tehnologic realizat din 2016 până în prezent. Aceste module includ o colecție de programe vulnerabile, un modul pentru aproximarea suprafeței de atac a executabilelor, un modul de detecție a vulnerabilităților folosind fuzzing și un modul de exploatare automată. Teza va prezenta metodic fiecare modul prin definirea tehnologiilor implicate, prezentarea arhitecturii software și a funcționalității, cât și validarea rezultatelor prin teste și evaluări.

## TABLE OF CONTENT

# TABLES

## FIGURES

# LISTINGS

# GLOSSARY

**attack surface** set of points on the boundary of an executable where an attacker can try to attack.

**emulator** software that permits executables written for one instruction set architecture to be run on another architecture.

**executable** set of computer instructions that has been created (usually by a compiler) so that it can be run by a CPU without additional processing.

**exploitation** act of an attacker for maliciously using an executable in a way in which it was not programmed.

**fuzzing** automated testing technique that involves providing invalid, unexpected, or random data as inputs to an executable.

**input stream** way of communication between an executable and its environment, through which it reads data to process.

**instrumentation** insertion new code at any point in an existing executable to observe or modify its behavior.

**signature** distinguishing pattern associated with an attack.

**symbolic execution** technique for executing an executable by finding all execution paths and generate corresponding input for each path.

**virtual machine** software that allows a single host to run one or more guest operating systems.

**vulnerability** weakness in an executable that could be exploited or triggered by an attacker.

**weakness** condition in an executable that, under certain circumstances, could contribute to the introduction of vulnerabilities.

# ACRONYMS

**API** Application Programmable Interface  
**ASLR** Address Space Layout Randomization  
**CLI** Command Line Interface  
**CRS** Cyber Reasoning Systems  
**CSV** Comma Separated Values  
**CTF** Capture the Flag  
**CWE** Common Weakness Enumeration  
**DARPA** Defense Advanced Research Projects Agency  
**DBI** Dynamic Binary Instrumentation  
**DOP** Data-Oriented Programming  
**ELF** Executable and Linkable Format  
**PIE** Position Independent Execution  
**ROP** Return-Oriented Programming

# 1 INTRODUCTION

Although computers had already emerged in their analog form during the 19th century, their significant proliferation began in the mid-20th century. 1941 marked the creation of Z3, the initial operational digital computer developed by Konrad Zuse. The distinctive feature of this machinery lies in its programmability through the utilization of punch cards. The source code, which comprises instructions that can be comprehended and executed by the computer, has the potential to be stored in an analog form.

After eighty years, these computing machines have become ubiquitous in our surroundings. Despite the rise in their usefulness, the fundamental structure remains unchanged: a collection of software programs dictate the behavior of physical circuits or hardware. As a result of the rise in utility costs, there was a corresponding increase in the number of users. Initially, a part of the users exhibit benign behavior, utilizing technology for its intended purpose. Conversely, there exist malicious actors who aim to acquire either material or immaterial (such as reputation or recognition) advantages, through the exploitation of technology, including its software sections.

The susceptibility of software to attacks can be attributed to the fact that it is developed by individuals who are inherently prone to errors. The liability for the error should not rest solely on the individual who authored the code, but rather, measures for validation could be implemented to safeguard the written code. The techniques employed comprise either a manual or automated procedure. One instance of the former is a security code review, wherein a proficient engineer assesses the security posture of a codebase. Regarding the latter, it is possible to incorporate security linters, scanners, and fuzzers into integrated development environments (IDEs), continuous integration/continuous deployment (CI/CD) pipelines, or staging environments.

The Cyber Grand Challenge was convened in 2016 by the Defense Advanced Research Projects Agency (DARPA), an agency dedicated to research and development under the purview of the United States Department of Defense. The event involved a competition among automated systems, which engaged in a demonstration of their accuracy and effectiveness by engaging in warfare with one another. Termed cyber reasoning system (CRS), their primary objective is to identify weaknesses in the binaries that are executed by both them and their opponents simultaneously. Participants in the Capture The Flag (CTF) competition had the opportunity to accumulate points through the automated generation of exploits that could be executed against their opponents, as well as by patching their binaries to safeguard against potential attacks from adversaries.

Currently, the three leading contenders are deemed to be state-of-the-art. Mayhem [22], the winner, was turned into a commercial solution. Although Shellphish [27] was released as an open-source project, the open-source environment has not undergone any significant changes. This matter serves as the initial point for two theses, specifically the present one and another [13] authored by Claudiu Ghenea. The authors present OpenCRS, an open-source system for cyber reasoning that is capable of identifying, exploiting, and remedying vulnerabilities in C-based executables designed for `i386` and in Executable and Linkable Format (ELF).

## 1.1 Objectives

The primary aim of this manuscript is to explain and assess four components of the cyber reasoning system that were developed by the author:

1. The dataset module, responsible for managing test suites of vulnerable programs;
2. The attack surface approximation module, which identifies the executable's interaction with the environment;
3. The vulnerability detection module, which generates input that triggers a vulnerability in the executable; and
4. The automatic exploit generation module, which produces a functional exploit for each identified vulnerability.

In addition, the document will briefly outline how OpenCRS works, an aspect that will be handled in greater depth by the other thesis that we mentioned.

## 1.2 Structure

The thesis is divided into eight chapters.

- Chapter 1: This one described the present context for software security, cyber reasoning systems, and the objective of this publication.
- Chapter 2: The subsequent chapter will expound upon OpenCRS, providing a comprehensive overview of its architecture and the limitations that have been established for this particular system.
- Chapters 3 to 6: The next four chapters will expound on each module that was referenced in the enumerated list aforementioned. Each module provides a detailed description of the theoretical aspects of binary analysis subfields that are implemented, including the state-of-the-art in each area. Additionally, the architecture and implementation of each module, as well as the technologies employed, are thoroughly discussed.

- Chapter 7: Chapter seven of the module focuses on testing and evaluation, providing precise data that demonstrates the accuracy of the results.
- Chapter 8: The final chapter provides a summary of the thesis content and draws conclusions based on the findings presented. In addition, it offers recommendations for future enhancements and modifications to the OpenCRS platform and its associated modules.

## 2 OPENCRS

This study, together with Claudiu Ghenea's [13], presents an open-source cyber reasoning system. OpenCRS aims to implement a comprehensive security assessment process for executables by utilizing the latest advancements in binary analysis from both academic and industrial sectors. The purpose of OpenCRS is to execute a comprehensive security evaluation of executable files, encompassing the identification of input methods and the development of patches to address any detected vulnerabilities.

Given the significant diversity in architectures, programming languages, and binary behaviors, the objective was to create a proof of concept that addresses the execution of programs exhibiting the subsequent attributes:

- The system operates on a 32-bit Intel architecture, specifically utilizing the `i386` instruction set architecture.
- The software is compatible with the Linux operating system and utilizes the ELF format.
- These are generated from the source code written in the C programming language.
- The input streams for arguments, standard input, and files are utilized.

The diagram below depicts the comprehensive architecture of OpenCRS and the inter-module communication taking place within the system.

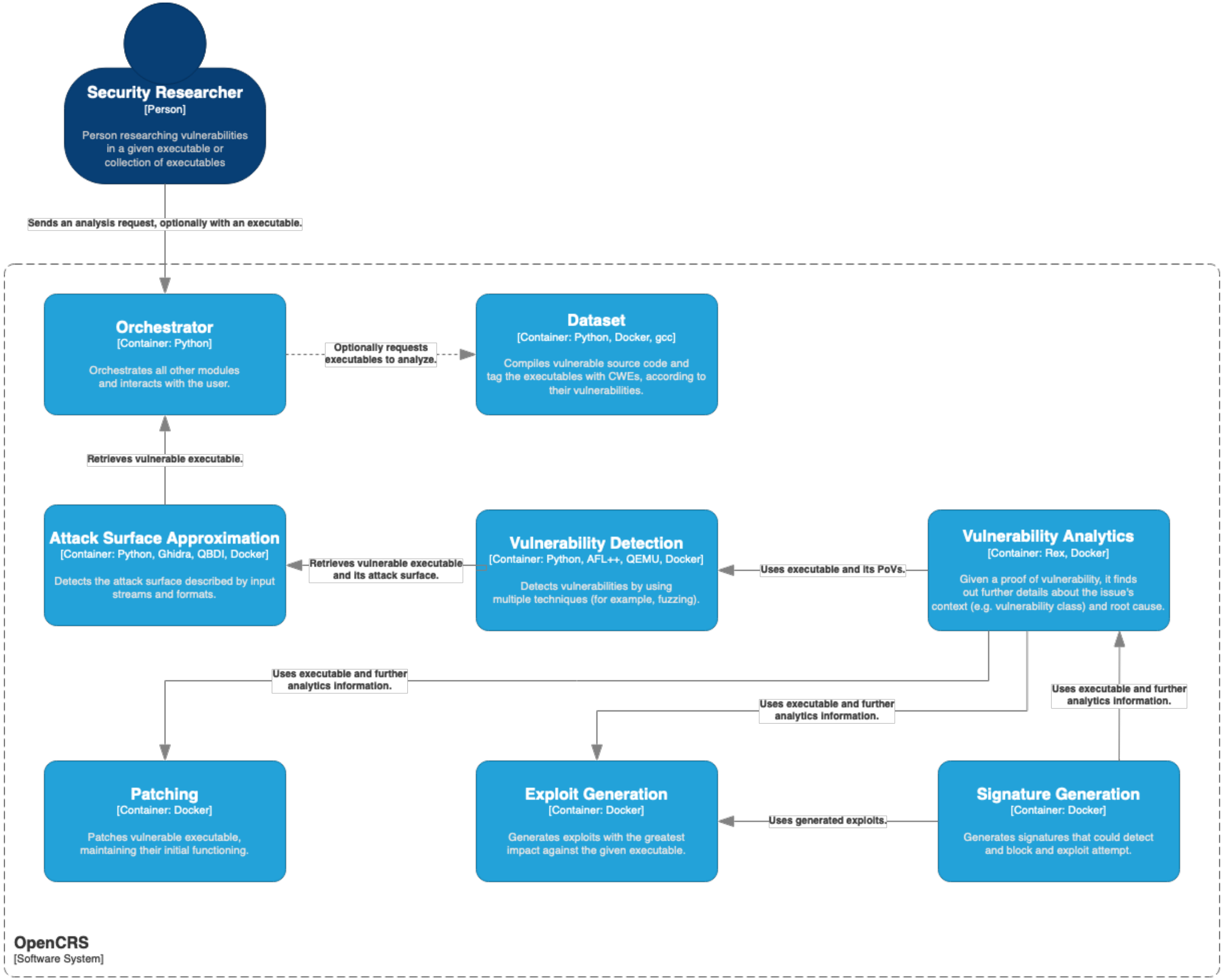

Security Researcher
[Person]
Person researching vulnerabilities in a given executable or collection of executables
Sends an analysis request, optionally with an executable.
Orchestrator
[Container: Python]
Orchestrates all other modules and interacts with the user.
Optionally requests executables to analyze.
Dataset
[Container: Python, Docker, gcc]
Compiles vulnerable source code and tag the executables with CWEs, according to their vulnerabilities.
Retrieves vulnerable executable.
Attack Surface Approximation
[Container: Python, Ghidra, QBDI, Docker]
Detects the attack surface described by input streams and formats.
Retrieves vulnerable executable and its attack surface.
Vulnerability Detection
[Container: Python, AFL++, QEMU, Docker]
Detects vulnerabilities by using multiple techniques (for example, fuzzing).
Uses executable and its PoVs.
Vulnerability Analytics
[Container: Rex, Docker]
Given a proof of vulnerability, it finds out further details about the issue's context (e.g. vulnerability class) and root cause.
Uses executable and further analytics information.
Uses executable and further analytics information.
Uses executable and further analytics information.
Patching
[Container: Docker]
Patches vulnerable executable, maintaining their initial functioning.
Exploit Generation
[Container: Docker]
Generates exploits with the greatest impact against the given executable.
Uses generated exploits.
Signature Generation
[Container: Docker]
Generates signatures that could detect and block and exploit attempt.
OpenCRS
[Software System]

Figure 1: OpenCRS's Architecture

Given that an analyst demands an executable analysis, the orchestration module manages the modules and their internal procedures as follows:

1. Dataset module: By incorporating various public test suites into the C source code, the module can compile it and generate a sequence of executable files. The primary benefit of this methodology is that OpenCRS can implement a validation mechanism that takes into account the vulnerabilities present in the source code, as indicated by the CWE labels, as well as those that have been identified, exploited, and remediated by the system. The dataset module is not mandatory since the analyzed executables utilized by OpenCRS may have already been constructed from alternative origins. Both open-source software, such as HiColor[1], and closed-source software, such as Dropbox[2], offer their users the option to download prebuilt binaries that cater to a variety of architectures.
2. Attack surface approximation module: With an executable (and no other knowledge about it) as input, this module will determine how it might be attacked: either through input streams or a predefined format for them (for example, the arguments that the program expects in `argv`).
3. Vulnerability discovery module: Subsequent to the identification of the attack surface by the preceding module, this module will employ vulnerability discovery methodologies such as fuzzing and symbolic execution to ascertain the existence of vulnerabilities, specifically inputs that result in erroneous program behavior.
4. Vulnerability analytics module: Upon detecting a proof of vulnerability, this module conducts an analysis to provide additional information regarding the specific vulnerability that has been identified. This may include identifying the category of vulnerability, such as a buffer overflow.
5. Automatic exploit generation module: The preceding module's proof of vulnerability is utilized to develop an exploit that triggers it and has the greatest possible impact.
6. Signature generation module: In contrast to the offensive nature of its predecessor, the module responsible for generating signatures serves a protective function. By utilizing identical input as the automatic exploit generation module, a signature is generated with the purpose of identifying and preventing exploitation endeavors. The module is deemed valuable in scenarios involving critical or outdated systems, wherein the installation of an alternative program is not feasible due to the unavailability of the program or compatibility issues.
7. Healing module: The objective of this module is akin to that of the signature generation module, which is to safeguard the binary from exploitation techniques. In contrast to

[1] https://github.com/dbohdan/hicolor

[2] https://dropbox.com

the preceding module, the current one has made alterations to the binary in order to eliminate the vulnerable code or implement appropriate sanitization measures, while preserving the original functionality.

Subsequent chapters will provide a comprehensive account of the internal operations of various modules. It is noteworthy that the outstanding modules have been addressed in separate theses: the vulnerability analytics and healing modules in [13], and the signature generation module in [25].

# 3 THE DATASET MODULE

The initial component of the cyber reasoning system is the dataset module. The aim of this component is to create and manage collections of vulnerable software programs utilizing public test suites.

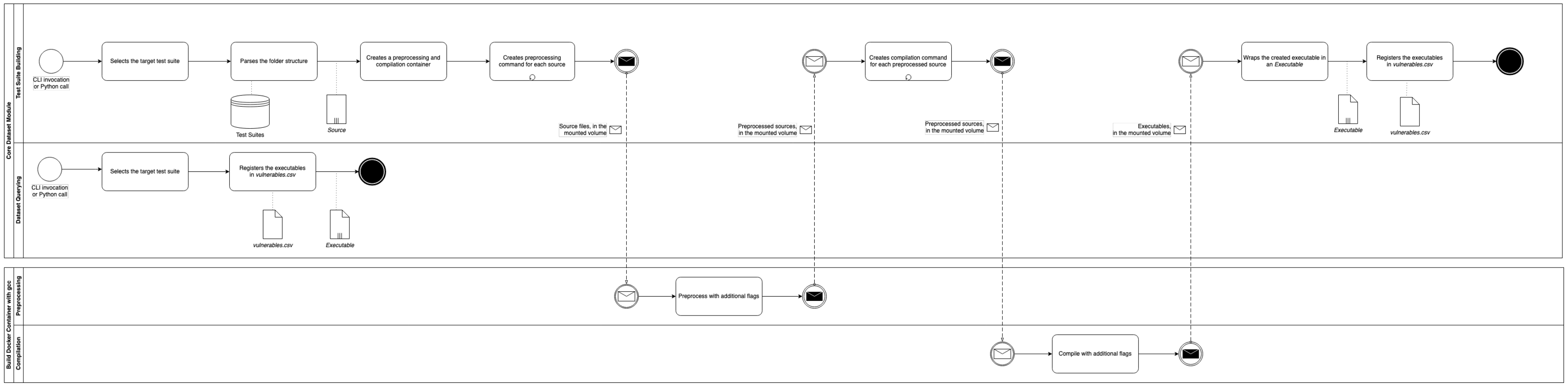
An additional prerequisite for the resulting binary files is that they must be labeled with appropriate information pertaining to their vulnerabilities. This will aid in calculating the overall system's accuracy by comparing the vulnerabilities found (correctly or wrongly) by OpenCRS with the intended ones, as designated in the initial datasets.

## 3.1 Implementation

### 3.1.1 Embedded Test Suites

We began by scanning the public domain for well-known test suites that had insecure C applications. We filtered out datasets that contained non-compilable code because OpenCRS analyzes full executables using static and dynamic analysis techniques. Articles in Academia [12], for example, use just the code modifications introduced by a commit, a property that is incompatible with the executable-level, black-box analysis done by our CRS. On the other side, we chose to use programs that had been precisely tagged with vulnerability tags, such as by constructing synthetic, vulnerable-by-design programs. This contradicts the strategy of utilizing a static code analyzer (such as SonarCloud [26]) to find and label potential vulnerabilities in code.

C Test Suite for Source Code Analyzer v2 - Vulnerable [7], developed in partnership between the National Institute of Standards and Technology and Alexander Hoole of the University of Victoria in Canada, is one of the test suites integrated into the dataset modules. It is made up of 54 C source codes plus a manifest file. The latter adheres to the Extensible Markup Language (XML) standard and includes a `<testcase>` entry for each program in the test suite, as well as the required child tags listed below. Because each program has a vulnerability, the latter is stated in each tag on the XPath `/testcase/file[]/flaw` and described in `line` (i.e. the line where the security issue occurs) and `name` (which is, in fact, the Common Weakness Enumeration item).



Core Dataset Module
Test Suite Building
CLI invocation or Python call
Selects the target test suite
Parses the folder structure
Test Suites
Source
Creates a preprocessing and compilation container
Creates preprocessing command for each source
Source files, in the mounted volume
Preprocessed sources, in the mounted volume
Creates compilation command for each preprocessed source
Preprocessed sources, in the mounted volume
Executables, in the mounted volume
Wraps the created executable in an *Executable*
*Executable*
Registers the executables in *vulnerables.csv*
*vulnerables.csv*
Dataset Querying
CLI invocation or Python call
Selects the target test suite
Registers the executables in *vulnerables.csv*
*vulnerables.csv*
*Executable*
Build Docker Container with gcc
Preprocessing
Preprocess with additional flags
Compilation
Compile with additional flags

Figure 2: Architecture of the Dataset Module

Despite the fact that the build command was attached in /testcase/@instruction, we chose to relocate the build logic into a separate Makefile. This allows the executable to be created without first parsing the XML file. We also published this revision to a separate repository on our GitHub organization [23], in addition to the test suite. This allowed us to add it to the dataset repository as a submodule.

The second test suite is Juliet C/C++ 1.3 [18], which is already available on GitHub [19]. The same American institution built it, but in partnership with the National Security Agency's Center for Assured Software. It consists of 64123 susceptible test cases and an XML manifest file with minor variations from the previous one (for example, not having the build command in /testcase/@instruction).

The necessity for custom binaries became more apparent during the testing of other components. We ended up building an additional custom test suite in the dataset module called toy_test_suite with a simple folder structure: a source containing vulnerable code and a cwe.txt file with the associated Common Weakness Enumeration (CWE) ID. Small executables such as stack buffer overflows, NULL pointer dereferences, and tainted format strings were included here.

### 3.1.2 Test Suites Building. Results Retrieval

These three datasets were linked as submodules in the dataset's [11] raw_testsuites folder. In the dataset.parsers module, a distinct class (called parser) is created for each of them. The class should implement the following abstract methods because it should inherit the BaseParser class:

- _get_all_sources: Parses raw_testsuites/<testsuite_id>, which is the folder structure of the dataset, and returns a list of Source objects. The latter contains information such as the full path of the source file and the embedded CWEs.
- preprocess: Having the Source objects already created, the method deals with preprocessing the sources with gcc, eventually by considering custom preprocessing flags. The resulting files are placed in the sources folder.
- _generate_gcc_command: Based on the preprocessed sources in sources and additional compilation flags, uses gcc to compile the executables in executables. Further details are placed in a Comma Separated Values (CSV) file, vulnerables.csv, which has columns indicating the executable ID, the test suite it comes from, its CWEs, and a boolean indicating if it is built or not.

All gcc-related operations (both preprocessing and building) are carried out in a Docker container running Ubuntu. This allows for the replication of the building process as well as separation from the host operating system. We like to communicate with the container using volumes (for file sharing) and Docker Application Programmable Interface (API) calls. This was preferable to using a gRPC service because the module's code resides on the same host as the build container, and communication delays are reduced by avoiding transferring large files (e.g. the sources and executables) via (a virtualized) network.

A parsers' manager is enabled when the build functionality is started from the command-line interface or by invoking the specified method from the Python module. It chooses the necessary parsers and invokes the preprocessing and construction techniques. The process's output, specifically `Executable` objects, can be queried using other procedures, in which `vulnerable.csv` is parsed once again.

# 4 THE ATTACK SURFACE APPROXIMATION MODULE

Aside from the internal procedures that are automated, the software should have ways of interacting with the outside world, either the environment or the users. The following are the most commonly utilized communication channels:

- `stdin` (standard input);
- Arguments;
- Files;
- Network packets;
- System and library calls;
- Graphical user interface (GUI); and
- Environment variables.

When data is sent with malicious intent, the input streams are referred to as attack vectors. They constitute the attack surface: the entire interface of the executables with the outside world, which can be exploited by a hostile actor.

Before looking for vulnerabilities in OpenCRS, it should be aware of the attack vectors. This observation serves as the foundation for the following module [3] in OpenCRS' pipeline, the attack surface approximation. With a binary as input (eventually fed from the dataset module), it discovers attack vectors that can be used by the CRS's subsequent modules.

As previously stated, the variety of input sources is extensive. In our proof of concept, we simply used three of the most common: `stdin`, files, and arguments.

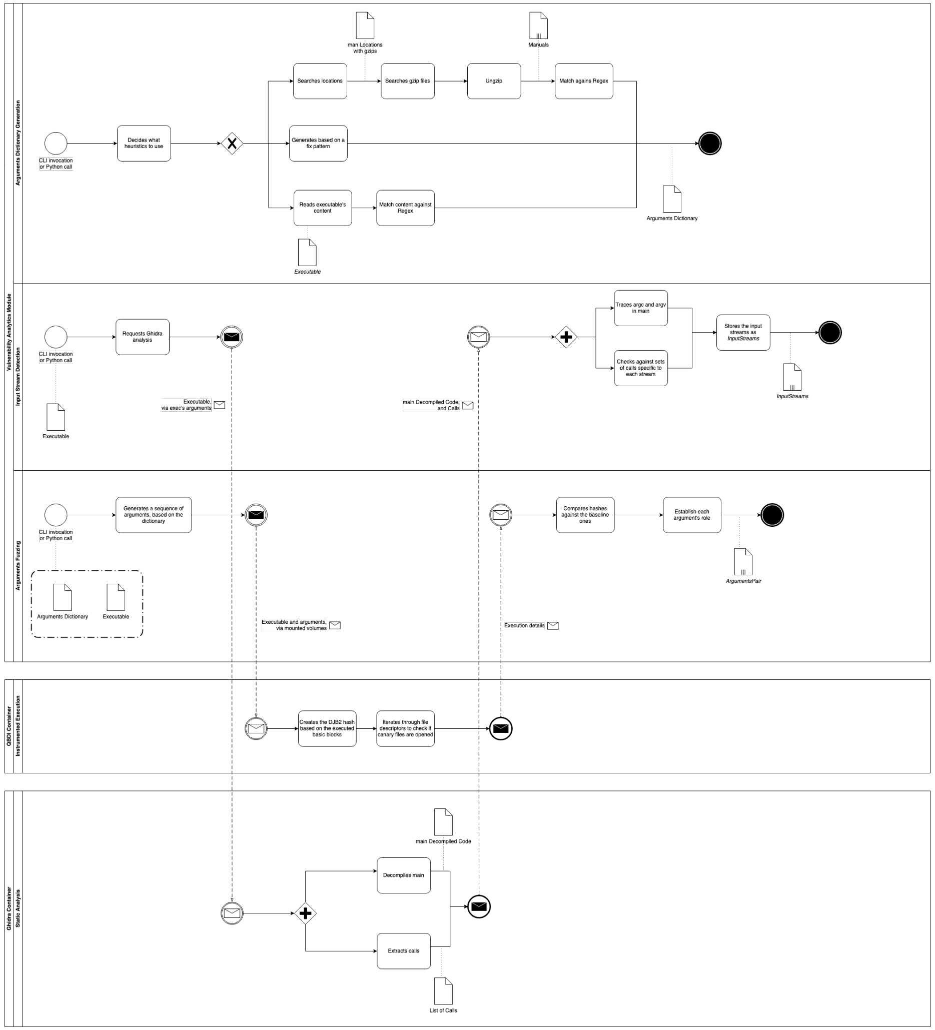

Vulnerability Analytics Module
Arguments Dictionary Generation
CLI invocation or Python call
Decides what heuristics to use
Searches locations
man Locations with gzips
Searches gzip files
Ungzip
Manuals
Match agains Regex
Generates based on a fix pattern
Reads executable's content
Executable
Match content against Regex
Arguments Dictionary
Input Stream Detection
CLI invocation or Python call
Executable
Requests Ghidra analysis
Executable, via exec's arguments
main Decompiled Code, and Calls
Traces argc and argv in main
Checks against sets of calls specific to each stream
Stores the input streams as InputStreams
InputStreams
Arguments Fuzzing
CLI invocation or Python call
Arguments Dictionary
Executable
Generates a sequence of arguments, based on the dictionary
Executable and arguments, via mounted volumes
Execution details
Compares hashes against the baseline ones
Establish each argument's role
ArgumentsPair
QBDI Container
Instrumented Execution
Creates the DJB2 hash based on the executed basic blocks
Iterates through file descriptors to check if canary files are opened
Ghidra Container
Static Analysis
Decompiles main
main Decompiled Code
Extracts calls
List of Calls

Figure 3: Architecture of the Attack Surface Approximation Module

## 4.1 Theoretical Considerations

The paper [10] begins by introducing the concept of front line functions, or functions adjacent to a source of input. [21] expands on this concept by introducing the concept of direct entry point: a method that includes a call to one of the specified input methods (for example, `read` from `unistd.h`). Despite the fact that the papers give theoretical notions that are analogous to the input streams discussed in this thesis, they do not provide any instrument or approach for identifying these input entrances in processes. It should also be noted that the papers approach this topic by studying the source code.

This CRS module, on the other hand, implies creating valid arguments that can be used in later modules (for example, in vulnerability identification via fuzzing). Papers such as [15][14][20] propose argument identification, but only with extensive understanding of the investigated software. To parse, they require a grammar, a `getopt` option parsing call, or a man page.

The only implementations identified were in AFL[1], afl++[2], and ManFuzzer[3]. The first two have the `argv_fuzzing'`mode, which inserts the generated random bytes into the main function's `argv`. The third piece of software generates fuzzing sequences by scanning the man page and the output of programs run with the `-h`, `-H`, and `--help` options. The downside of this technique is that it does not take into account the typical format of an input (e.g. `-letter_or_word>` or `--letter_or_word>`). For the latter, there is an inherent risk that the software lacks a help page or has hidden arguments (e.g. experimental ones).

## 4.2 Implementation

The purpose of the module in detecting the input streams was divided into numerous tasks, implying distinct technologies.

### 4.2.1 Indicators Discovery

The first objective is to statically examine the binary for evidence (dubbed indications) that a particular input stream is being used. It should be emphasized that the presence of an indicator is sufficient but not required. Despite the fact that the program does not call `getenv` internally, we cannot guarantee that the environment variables are not used altogether because the program may employ an unusual method of parsing its stack to acquire that information.

This analysis aids in the activation of features in the pipeline's subsequent modules in OpenCRS. For example, if the executable does not use its command line interface (CLI) arguments, a fuzzer altering (non-existent) arguments is pointless and will yield no significant findings.

[1] https://github.com/google/AFL/tree/master/experimental/argv_fuzzing

[2] https://github.com/AFLplusplus/AFLplusplus/tree/stable/utils/argv_fuzzing

[3] https://github.com/GroundPound/ManFuzzer

We accomplish this by utilizing the API of a well-known reverse engineering tool, Ghidra, and confirming the required criteria for an executable to utilise a specific input stream.

- For arguments: The executable's 'main' function is decompiled, and the resulting code is processed as an abstract syntax tree. The tree is traversed in order to find interactions between the source code of 'main' and the program's arguments, `argc` and `argv`.
- For standard input and files: All function calls are collected into a set and repeatedly compared with multiple sets of function names, one for each input stream (e.g. `("read", "fgetc", "fread")` for `stdin`). The executable uses an input stream if there is an intersection between the two sets.

### 4.2.2 Generating Arguments Dictionaries

Before determining whether the binary uses any parameters, a list of possible arguments should be produced. We used three generation heuristics:

- `generation`: Generates parameters while adhering to the Regex format `-[a-zA-Z0-9]`.
- `binary_pattern_matching`: Looks for bytes sequences in binary material that match the Regex `\s-{1,2}[a-zA-Z0-9][a-zA-Z0-9_-]*` (matching, for example, `-f` and `--file`).
- `man_parsing`: First, it scans the `man` configuration files to determine which folders contain manuals. Finds all gzip files in this location, unzips them, and matches their content against the same Regex pattern as above. The returned arguments list can optionally be reduced to a specified amount of elements based on their occurrence.

### 4.2.3 Arguments Fuzzing

The dynamic approach is better for detecting that an argument (or combination of arguments) is used by the executable, as the static approach may result in erroneous results due to superficial inspection of the control flow graph. By evaluating the program dynamically, on the other hand, the input causes the program execution to take a different path, reaching distinct basic blocks (predictable sequences of instructions, represented as nodes in a control flow graph) and, subsequently, invoking different system API methods.

The purpose of the subtask is to extract coverage information that can then be utilized to distinguish between two executions of the same executable but with different parameters. This information can be extracted using a variety of dynamic binary analysis approaches, including:

- Debugging: A debugger can identify the execution flow by establishing hardware or software breakpoints on each basic block or system call. However, because it is designed for people (programmers, vulnerability researchers, etc.), it is not well-suited for automation in which no manual interaction is necessary. The performance hit was caused by either user-kernel space switches (when invoking the `ptrace` API from the debugger process) or a type of dynamic binary instrumentation, which involved rewriting the code with debugging-related calls or interrupts.
- Static instrumentation and execution: This method entails translating the program into a high-level abstraction language, inserting instrumentation code, and compiling it. The newly produced executable is then run and can report execution coverage information. This strategy, as an alternative to compiler instrumentation, may result in unanticipated behavior due to a lack of standardized technology in this field of research.
- Dynamic binary instrumentation (DBI): A DBI instrument attempts to resolve debugger difficulties. Using Quarkslab's engine, QBDI (which is utilized in our implementation), it is possible to inject instrumentation code inside the binary at runtime. The optimization is that the analysis tool and the analyzed program are both performed under the same process, which reduces friction.

Following several experiments with the first two methodologies, we found that the QBDI could be integrated by taking the following steps:

- Convert the executable to a library by removing the Position Independent Execution (PIE) flag (which is present in executables) and marking the `main` function as exported with the LIEF Python library.
- `dlopen` is used to load the executable into program memory.
- Instrument the code with the QBDI C API.

It failed because of the QBDI's execution transfer: it deems any calls to dynamically linked libraries to be non-reentrant and moves the execution from instrumented to native (with no instrumentation at all). It returns the instrumentation after the function return.

This limitation prompted us to retry with a different API of QBDI, `QBDIPreload`. It is a utility library in which the DBI engine overwrites and calls all of the callbacks exported by the QBDI API during runtime. The instrumentation library is loaded into process memory via the Linux loader's `LD_PRELOAD` command. This second method involved a number of steps:

- Design a callback method to handle basic block calls. To eliminate the variations generated by Address Space Layout Randomization, it employs an address normalization technique. This relativization of the basic block's start address is saved in a list.

```
start_address -= segments[parent_segment].start;
abstract_address = (parent_segment << 24) + start_address;
```

Listing 4.1: Address Normalization Technique

- Write a callback function to handle execution transfers. It checks to see if `close` is called on a canary file passed to the application as an argument.
- Make a non-cryptographic DJB2 hash over the first 1000 addresses in the control flow graph and dump it into a file upon exit.

The fuzzing process begins by creating baseline hashes using the stated coverage extraction mechanism by executing the program with no arguments and a series of random, 10-character long arguments. Following that, a series of commands are executed into an Ubuntu-based Docker container, with a full QBDI setup and communication via mounted Docker volumes:

- Only a filename as an argument;
- Each argument in the dictionary, one at a time;
- – as an argument; and
- Each argument in the dictionary, one at a time, plus a canary string.

Each execution generates a DJB2 hash, which is compared to the baseline provided. If the hash was not observed until that point, the algorithm can conclude that the argument resulted in a different execution of the binary (i.e. a different sequence of nodes in the control flow graph) and is therefore registered as a legitimate argument. Aside from that, each argument has a role that specifies its effects:

- Flag (e.g. `--force`): The argument alone generates a unique hash.
- File enabler (e.g. `--config production.yaml`): The program is invoked with the current input followed by the name of an OpenCRS canary file with the `.opencrs` extension. On each execution transfer, the process's file descriptors are examined to see if the canary file is open.
- `stdin` enabled (e.g. `--stdin`): If an execution fails due to a timeout, the program is repeated with input from `stdin`. If the hashes produced by these two executions differ, the flag is regarded to be enabling `stdin`-reading capabilities and is tagged appropriately.
- String enabler (e.g. `--action deploy`): The program's execution with the arguments alone is compared against one of the arguments supplemented with a string. If the first differs from the baselines and the latter differs from the first, the argument will be assigned this role.

# 5 THE VULNERABILITY DETECTION MODULE

The vulnerability detection module's [29] objective is to uncover vulnerabilities in executables using the information retrieved by the attack surface approximation module (the input streams, arguments, and their responsibilities). This approach not only concludes the probability of a vulnerability occurring, but it also generates an actual proof of vulnerability (i.e. an input that triggers the vulnerability) object, `ProofOfVulnerability`. It will then be provided outside the module, namely to the vulnerability analytics module, to obtain the root cause and the affected executable internals.

Even though this chapter only discusses fuzzing to find vulnerabilities, [9] discusses merging symbolic execution with Retdec[1] and KLEE[2] in this OpenCRS module.

## 5.1 Theoretical Considerations

Path explosion, a problem particular to approaches such as symbolic execution, can arise during the investigation of the executable and its execution. Because symbolic values are propagated down each path of the graph, the complexity of detecting vulnerabilities in such a vast quantity of data is impractical given memory and compute capacity constraints.

We then use fuzzing to get around this. The sending of random inputs on each input stream causes the execution to go in different directions, with different input variables. The likelihood of triggering a vulnerability could then be raised.

Another major constraint for fuzzing is the CRS's black-box (or binary-only) nature: the vulnerability discovery module does not have access to the executable's source code, but only to the binary itself. This narrowed our search because fuzzers usually use source code instrumentation, which involves inserting small amounts of code during the compilation stage so that the fuzzer can determine which path of the control flow graph was reached by the execution.

To begin, we investigated American Fuzzy Lop (AFL) [2] as a cutting-edge blackbox fuzzer that we could integrate into the cyber reasoning system, but Google's upstream has had no development activity since June 2021, and the repository itself is archived. A possible alternative is afl++ [1], a fork of it that is constantly developed by the open-source community and offers features such as:

[1] https://github.com/avast/retdec

[2] https://github.com/klee/klee

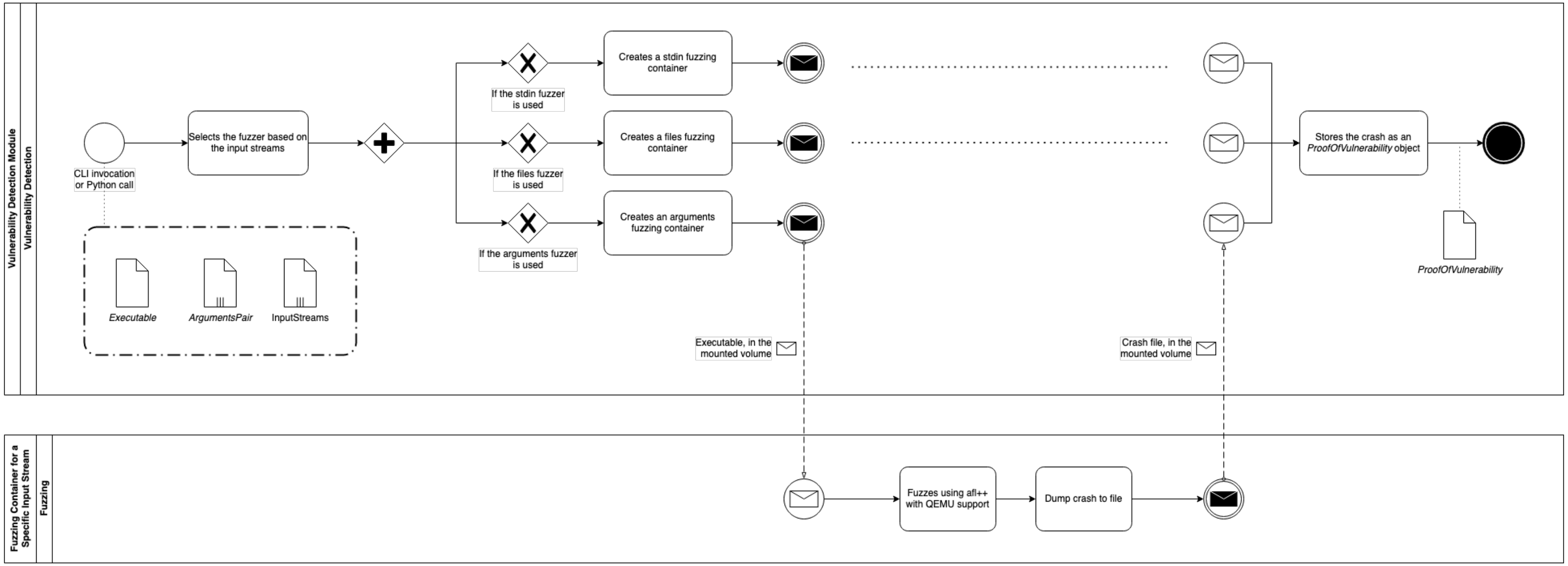

Vulnerability Detection Module
Vulnerability Detection
CLI invocation or Python call
Executable
ArgumentsPair
InputStreams
Selects the fuzzer based on the input streams
If the stdin fuzzer is used
If the files fuzzer is used
If the arguments fuzzer is used
Creates a stdin fuzzing container
Creates a files fuzzing container
Creates an arguments fuzzing container
Executable, in the mounted volume
Crash file, in the mounted volume
Stores the crash as an ProofOfVulnerability object
ProofOfVulnerability
Fuzzing Container for a Specific Input Stream
Fuzzing
Fuzzes using afl++ with QEMU support
Dump crash to file

Figure 4: Architecture of the Vulnerability Detection Module

- QEMU emulation for deducing relevant information (for example, coverage) on runtime, without the need for source code instrumentation prior to compilation;

- Persistent mode for specifying a section of the code segment that should be executed in a loop by the fuzzer, without the need to create a process each time new inputs are provided to the executable; and

- Increased speed, resulting in more mutations in the fuzzing engine.

## 5.2 Implementation

Our method began by evaluating a subset of all possible input streams (files, standard input, and arguments) that an executable could have. When calling the vulnerability detection functionality, this information is explicitly provided, and it can be derived using the attack surface approximation module.

As previously stated, afl++ was chosen as a tool for blackbox fuzzing. We designed a custom Docker image to serve as its environment, with the aims of isolation and easy regeneration in mind. QEMU support for binary-only fuzzing was enabled by running the `build_qemu_support.sh` script.

It is instantiated by utilizing the Docker API for Python, with the necessary volumes (for target executable, sample inputs, and analysis) automatically attached. Furthermore, the same Python code runs `afl-fuzz` inside the newly constructed container to fuzz the supplied target executable with the sample inputs (if any). These are mutated and given to standard input or placed in files (afl++ supports two input streams by default), data that is consumed by the binary.

When afl++ detects a new crash for `stdin` and files, it creates a new file in the analysis directory (`/tmp/fuzzer/<session>/output/default/crashes/`), which is continuously watched by a Python watchdog. The crash file is read in order to construct a new instance of the `ProofOfVulnerability` class.

For argument fuzzing, we utilized the same Docker image. The primary difference is that the fuzzed binary is no longer the target binary, but rather a customized adaptor written in C. The wrapper reads from `stdin` the random input created by afl++. The input is then tokenized by utilizing space as a delimiter (`0x20` ASCII) and detecting formatting placeholders (`%s`) that it replaces with the input generated by afl++. The generated `char *` sequence is passed as an `argv` argument to an `execve` call.

The adapter's image is the process's initial image as formed by the fuzzer. Using the given approach, the system call substitutes the process image with the actual target binary image, with a random string of characters as parameters. Mismanagement of an argument can result in a crash, which afl++ can identify and report to our watchdog.

# 6 THE AUTOMATIC EXPLOIT GENERATION MODULE

The last module [4] given in this thesis aims towards automatic exploit generation. It is directly linked to two modules to obtain the data it requires: the vulnerability detection module, which provides proofs of vulnerabilities (specifically, `ProofOfVulnerability` objects), and the vulnerability analytics module, which provides further details about the exploitation circumstances and exploitability.

## 6.1 Theoretical Aspects

We investigated several articles on the subject to determine the current academic development in the field of automatic exploit generation. All of them can be classified as follows:

- Papers describing an automatic exploit generation (AEG) (sub)system [5] [8] [16] [30]: The majority of the studies examined handle the problem by combining it with vulnerability detection. In this manner, the systems discover software flaws by combining several methodologies and developing functional exploits.
- Papers summarizing the current AEG context [6] [17] [28]: This category does not show specific AEG system implementations, but rather examines methodologies and approaches proposed in functional cyber reasoning systems (for example, those from DARPA's Cyber Grand Challenge) and academia as separate exploitation research.

The following lists provide a summary of the most common flaws, mitigations, (support) exploitation approaches, and consequences mentioned or used in the studied papers:

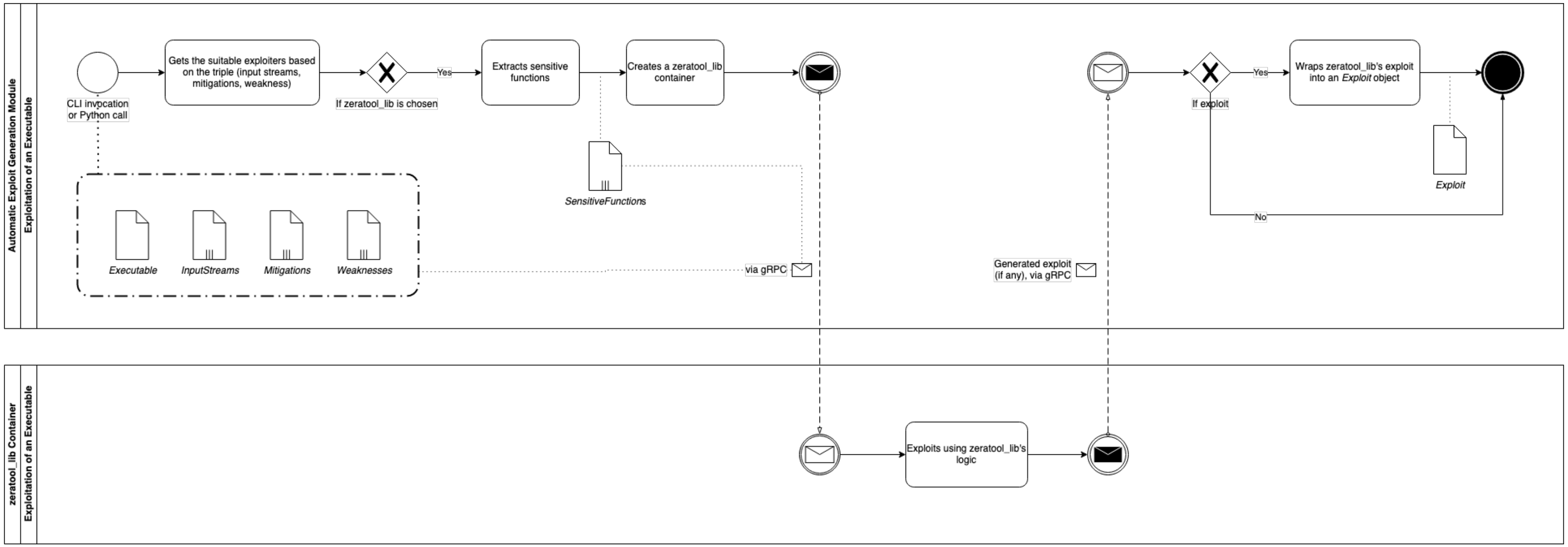

Automatic Exploit Generation Module
Exploitation of an Executable
CLI invocation
or Python call
Gets the suitable exploiters based on the triple (input streams, mitigations, weakness)
If zeratool_lib is chosen
Yes
Extracts sensitive functions
Creates a zeratool_lib container
SensitiveFunctions
Executable
InputStreams
Mitigations
Weaknesses
via gRPC
Generated exploit (if any), via gRPC
If exploit
Yes
No
Wraps zeratool_lib's exploit into an Exploit object
Exploit
zeratool_lib Container
Exploitation of an Executable
Exploits using zeratool_lib's logic

Figure 5: Architecture of the Automatic Exploit Generation Module

- Weaknesses
  - Out-of-bound memory writes: Consists of writing data whose length exceeds the buffer length. There are two types of allocation: on stack, for variables defined in a function, or heap, when the buffer is dynamically allocated. If the stack is targeted, then the return address and old base pointer can be overwritten.
  - Type overflow: The correct result of an operation could not be stored in a certain type of variable.
  - Tainted format strings: The first parameter of functions working with a format string (such as `printf`) is user controlled.
  - Information leaks: The program divulges information about its execution (for example, memory layout).
- Mitigations
  - Address Space Layout Randomization (ASLR): Segments such as the stack, dynamic libraries, and heap are placed randomly in the process memory space.
  - Position Independent Execution: The code segment is placed, as in ASLR, randomly in the memory space. The program should be compiled with Position Independent Code enabled.
  - NX bit: If enabled, the bit prevents a page to be executed and writable at the same time.
  - Stack canaries: The canaries are values placed on the stack after entering a frame. It is checked on return and, if the check fails, then the execution is aborted.
  - Control flow integrity: Permits only certain code, part of the valid control flow graph, to be executed. New code, injected by attackers in process memory, could not be executed anymore.
- Exploitation techniques (including support ones)
  - Shellcode: Machine code is injected in process memory and executed by chaining with other techniques for hijacking the execution flow.
  - Return-oriented programming (ROP): Unlike shellcodes, which inject new code, ROPs reuse code already existent in process memory. A chain of gadgets is created and executed by coupling the technique with execution flow hijack.
  - Data-oriented programming (DOP): The accent in DOP resides on data flow and how it can be altered by using existent code constructs (similar to ROP).
  - NOP sleds: This support technique is used with shellcodes to increase the probability that it will be executed correctly. The shellcode is prefixed with a `nop` sequence such that the execution could be redirected (for example, with return address overwrites) anywhere on the sled. This is in contrast to raw shellcode when the execution needs to start from the first shellcode byte.

- Outcomes
  - Code execution: The programs execute the code desired by the attacker.
  - Denial of service: The program functioning is blocked or halted at all.
  - Information leaks: Sensitive information is extracted.

[30] is an example of the whole data flow and operation of AEG systems. It takes a similar approach to what we picked for our cyber reasoning system. To begin, the proposed method obtains basic information regarding active mitigations such as NX bits and ASLR. After their mining module detects vulnerabilities, the protections are circumvented via returns to system libraries in the case of NX bits and `jmp esp` gadgets in the case of ASLR bypass. Then, a mathematical solver is utilized to automatically construct an exploit while taking the limitations into account. Binaries from the Cyber Grand Challenge were used to validate the entire process.

For OpenCRS, we looked into open-source projects that already have exploitation capabilities. Given our emphasis on 32-bit, C-based ELF executables, we came up with two candidates:

1. Ropstar [1]: Given an executable that accepts file input and eventually has PIE, ASLR, NX, and canaries enabled, Ropstar attempts to launch a shell or call a sensitive (or "win"") function by exploiting buffer overflows and improper format string usage. It may utilize techniques such as `.got` and `.plt` overwriting, as well as return-oriented programming (with return to `libc` or `.plt`, or direct `setuid` and `system` calls) depending on the exploitation scenario. We choose to integrate the following sophisticated solution due to the limited input streams that are provided by default. It should be noted that we uncovered a command injection vulnerability during the code examination (to better understand the tool's internals), which we filed via a public issue [2] on GitHub. At the same time, a patch with a PR [3] was proposed, which had not yet been integrated by the repository maintainer at the time of writing this thesis.
2. Zeratool [31]: The second open-source option we investigated (and integrated) is Zeratool, which is primarily used for CTF challenges. It is capable of exploiting stack-based out-of-bounds writes and format string weaknesses in binaries, although only the first has been included due to the prominence of the weakness in C-based executables. The exploits avoid eventual mitigations like as NX, PIE, and ASLR, and have the potential to:
   - Code execution

[1] https://github.com/xct/ropstar

[2] https://github.com/xct/ropstar/issues/12

[3] https://github.com/xct/ropstar/pull/11

  - Sensitive functions: Functions with names containing strings such as "*secret*", "*shell*", and "*system*", can be called as functionality relevant to an attacker can be unlocked;
  - Shellcodes;
  - Shell functions; and
- Leaking data from memory, which can respect a Regex pattern that is set at the beginning of the exploitation.

## 6.2 Implementation

OpenCRS modeled exploiters, which are independent exploitation logic capable of generating exploits under specified parameters dictated by input streams, mitigations, and results. These are validated by the method `_is_exploitation_supported`, which should be implemented in every exploiter that derives from the `BaseExploiter` class.

In this manner, the automatic exploitation module can query all of its exploiters to determine which can be used for a certain executable using an exploiters manager. The attack surface approximation module already provides the input streams and mitigations, and the results are specified in the CRS configuration.

As previously stated, Zeratool was integrated into OpenCRS as an exploiter by first forking the main repository [32] into our organization's new repository, `zeratool_lib`. The original codebase was modified as follows:

- The CLI was replaced with a unique function that CRS modules can call. As parameters of this function, all necessary parameters (input stream, method of exploitation, sensitive function, etc.) are accessible.
- Because of the local exploitation element of OpenCRS, all remote exploitation logic was removed.
- The main function returns the produced exploit.
- `libpwnable`, a library primarily used to aid with exploitation, is no longer an input stream. The only ones that are now supported are `stdin` and arguments.

If the query manager chooses Zeratool as a suitable exploiter for a certain case, the sensitive functions are extracted. These are supplied together with the binaries and its details through gRPC to a freshly built Docker container, where a Python 3 service will unpack them and invoke `zeratool_lib`. If the exploitation procedure is successful, the exploit is read and wrapped into a `Exploit` object.

# 7 TESTING AND ASSESSMENT

The forthcoming chapter will delineate the process of functional testing and evaluation of the OpenCRS modules that were previously explicated.

For testing and evaluation, a virtual machine was employed, which was configured to run Ubuntu 22.04. The guidelines for configuring each module were followed, thus guaranteeing the fulfillment of all requirements, including Python libraries, Docker, and built images.

Even though every module possesses the ability to operate as a Python library, we have chosen to employ the command line interface of each module owing to its superior suitability for manual testing. The integration of the modules into the OpenCRS will involve the utilization of a Python library within an orchestration module.

## 7.1 Dataset Generation

For the dataset module, we have constructed all three integrated test suites, which are identifiable in the OpenCRS context by a unique ID: `nist_juliet`, `nist_c_test_suite`, and `toy_test_suite`. The present task involved the utilization of the `built` command in combination with the `get` command, which serves to retrieve the index of the built executables.

```
$ opencrs—dataset build ——testsuite NIST_C_TEST_SUITE
Successfully built 50 executables.
$ opencrs—dataset get
The available executables are:

______________________________________________________________________________________________________
| ID                    | CWEs                                          | Full Path                               |
|-----------------------|-----------------------------------------------|-----------------------------------------|
| nist_c_test_suite_113 | Use of Externally—Controlled Format String    | executables/nist_c_test_suite_113.elf   |
| nist_c_test_suite_133 | Use of Externally—Controlled Format String    | executables/nist_c_test_suite_133.elf   |
[...]
```

Listing 7.1: Example of Building and Retrieving a Dataset

The outcome of this procedure yielded a set of 54591 vulnerable ELF executables, which were generated from C source code and designed for the 32-bit i386 architecture. The open-source community was granted access to the dataset through the establishment of a distinct repository, namely `opencrs_dataset` [24].

The observed distribution of executables across the test suites, namely 54531 for `nist_juliet`, 50 for `nist_c_test_suite`, and 10 for `toy_test_suite`, suggests the presence of compilation errors during the executables' generation process. Upon analyzing the build logs, it has been determined that the primary factors contributing to the issue are the absence of essential headers such as `mysql/mysql.h` and `security/pam_appl.h`, as well as inconsistencies between the Windows and Linux operating systems, such as the utilization of data types such

| Weakness | Count |
|---|---|
| Stack-based Buffer Overflow | 13836 |
| Heap-based Buffer Overflow | 11088 |
| Integer Overflow or Wraparound | 3960 |
| Mismatched Memory Management Routines | 3564 |
| Integer Underflow | 2952 |
| Free of Memory not on the Heap | 2680 |
| Use of Externally-Controlled Format String | 2410 |
| Buffer Underflow | 2048 |
| Buffer Under-read | 2048 |
| OS Command Injection | 1921 |

Table 1: Distribution of Vulnerable Executables, by CWE

as `wchar_t`.

The executable details were recorded in a CSV file denominated as `index.csv`, featuring the subsequent labels:

- `name`: Unique identifier of a program, the format `executables/<name>.elf` being utilized to determine the path of the executable file;
- `cwes`: One or more CWEs that are present in the executable; and
- `parent_dataset`: Parent dataset's identifier.

Finally, it is noteworthy that the employed test suites do not conform to a consistent distribution of executables across each vulnerability type, as identified by the CWE ID.

## 7.2 Arguments Dictionaries Generation

Subsequently, the module subjected to testing and evaluation was the attack surface discovery module.

Through utilization of the `generate` command and the three implemented heuristics, it is possible to generate dictionaries of arguments that may be employed for the purpose of argument fuzzing:

- `generation`: 62 entries with arguments like `-a`, `-D`, and `-9`;
- `man_parsing`: 6701 entries with arguments like `--ascii`, `--cert-status`, and `--message-id`; and

- `binary_pattern_matching` for `/usr/bin/uname`: 37 entries with arguments like `--processor`, `--operating-system`, and `-linux-x86-64`. As evident from the list, the precision is not unitary since not all the strings returned are valid arguments. This phenomenon occurs due to the adherence of certain character sequences within the binary files to the regular expression pattern for arguments. However, this does not present an impediment, as the aforementioned arguments will undergo two additional filtering stages, specifically the argument fuzzing and the vulnerability detection module.

```
$ opencrs-surface generate --heuristic binary_pattern_matching --elf /usr/bin/uname --output uname.dict
Successfully generated dictionary with 37 arguments
$ head -3 uname.dict
--operating-system
--processor
--version
```

Listing 7.2: Example of Generating a Dictionary with Arguments

## 7.3 Arguments Fuzzing

The `fuzz` command was subjected to testing and evaluation in the identical module, in comparison to the `uname` Linux utility, which is utilized for obtaining details regarding the present host and operating system. The previously generated dictionary was utilized for reference purposes.

```
$ opencrs-surface fuzz --elf /usr/bin/uname --dictionary uname.dict
Several arguments were detected for the given program:

 ──────────────────────────────────────────────
| Argument                     |      Role       |
├──────────────────────────────┼─────────────────┤
| -                            |      FLAG       |
| --all                        |      FLAG       |
| --all string                 | STRING_ENABLER  |
| --hardware-platform          |      FLAG       |
| --hardware-platform string   | STRING_ENABLER  |
[...]
```

Listing 7.3: Example of Fuzzing Arguments

`-`, `--all`, `--hardware-platform`, `--kernel-name`, `--kernel-release`, `--machine`, `--kernel-version`, `--nodename`, `--operating-system`, and `--processor` were identified as valid program arguments with two roles: `FLAG` (valid since they activate functionality) and `STRING_ENABLER` (invalid because they do not require the user to give any string after that). Upon examining the outcomes produced by QBDI, it can be inferred that the reason for this occurrence is due to the fact that the DJB2 hash generated for `<param>` (e.g. `-939574273` for `-a`) and for `<param> <string>` (e.g. 657648750 for `-a <string>`) are different.

Conversely, the hyphen symbol (`-`) does not constitute a legitimate argument, and the argument `--version`, which was originally incorporated in the produced dictionary, was not identified as a valid one. The observed occurrence could be attributed to the implementation of `uname`, as QBDI fails to recognize the code flow related to this option as a basic block transfer.

| Executable | Arguments Stream | Files Stream | stdin Stream |
|---|---|---|---|
| `null_pointer_deref_args.elf` | Detected (TP) | N/A | N/A |
| `null_pointer_deref_files.elf` | Detected (TP) | Detected (TP) | Detected (FP) |
| `null_pointer_deref_stdin.elf` | Detected (TP) | Detected (FP) | N/A |
| `multiple_inputs_streams.elf` | Detected (TP) | Detected (TP) | Detected (TP) |

Table 2: Accuracy in Detecting the Input Streams

The final aspect to note regarding the functionality of the attack surface approximation module pertains to the detection of aliases. The `uname` command has the capability to accept abbreviated forms for all of its preceding arguments. By way of illustration using the `--all` option and further scrutinizing the QBDI outcomes, it is observable that the hash value of `-939574273` is obtained when the flag role is tested, corresponding to the DJB2 hash of the `-a` parameter. Upon collision detection by the OpenCRS module, it will selectively incorporate solely the initial argument from the dictionary that has been detected with the corresponding hash (namely, `--all`).

## 7.4 Input Streams Detections

Four executables from the toy test suite were utilized to evaluate the efficacy of input stream detection. Each of the aforementioned programs possessed a minimum of one input stream originating from a collection consisting of `stdin`, files, and arguments.

```
$ opencrs-surface detect --elf null_pointer_deref_args.elf
Several input mechanisms were detected for the given program:

 ──────────────────────────────────────
 | Stream                  | Present  |
 |─────────────────────────|──────────|
 | STDIN                   |   No     |
 | ARGUMENTS               |   Yes    |
 | FILES                   |   No     |
 | ENVIRONMENT_VARIABLE    |   No     |
 | NETWORKING              |   No     |
 ──────────────────────────────────────
```

Listing 7.4: Example of Approximating the Attack Surface

Table 2 presents a summary of the outcomes.

In two programs, the file and `stdin` streams were wrongly recognized as present. This phenomenon occurs due to the utilization of reading library calls by the programs, which lack specificity towards any particular one. It is noteworthy that these entities possess the capability to read from a file pointer, thereby enabling them to manage input from both `stdin` (the `0` descriptor) and files (the descriptor obtained through invocations such as `open` and `fopen`). For this reason, the modules tend to prioritize the promotion of false positives (streams that are not actually genuine but can be confirmed through the vulnerability detection module) over false negatives (streams that are authentic but remain unreported).

## 7.5 Vulnerability Detection

To identify vulnerabilities using the three implemented fuzzers, we selected three programs that experienced crashes due to distinct causes when provided with a particular input on supported input streams. The identified weaknesses or susceptibilities were:

- Tainted format string: If `%s` was used as an input format string, a `NULL` pointer dereferencing occurred. Another attack used `%n` to try to write to the same `NULL` pointer.
- Stack buffer overflow: If the input is long enough, the saved EBP and EIP registers on the stack are overwritten. The program will crash when it returns because the values are invalid.
- NULL pointer dereferencing.

```
$ opencrs-vulnerability fuzz --fuzzer FILES_AFLPLUSPLUS --stream FILES --elf tainted_format_string_files.elf
New proof of vulnerability was generated with the following payloads:
- For FILES:

00000000: 6E 25 6E 25                                              n%n%
```

Listing 7.5: Example of Generating a Proof of Vulnerability while Fuzzing

The input supplied by afl++, namely the `%n`, prompted the NULL pointer dereferencing and subsequent crash in this example. The fuzzer successfully created the crashing inputs in the other two cases, notably a long input for the executable prone to buffer overflow and another with `y` on the fifth place for the one vulnerable to NULL pointer dereferencing.

## 7.6 Automatic Exploit Generation

The automatic exploit generating module was tested for the final functionality with three binary given by Zeratool, all having `stdin` as an input stream and vulnerable to buffer overflow, but with various mitigations:

1. Two with NX, canaries, and PIE disabled; and
2. One with NX enabled + canaries and PIE disabled.

Furthermore, one executable contained a sensitive function that could be used in the exploit (as a substitute to, say, other symbols). In every example, the module effectively developed exploits.

```
$ opencrs-aeg exploit --exploiter ZERATOOL --elf bof_nx_32 --stream STDIN --mitigation NX --weakness\
 STACK_OUT_OF_BOUND_WRITE
The exploiter could generate an exploit with the outcome of SHELL and the following payloads:

- For STDIN:
 [...]
\x00\x00\x00\x00\x00\x00\x00\x00\x90\x90\x04\x08baaa\x1e\xa0\x04\x08\x00\x00\x00\x00\x00
 [...]
```

Listing 7.6: Example of Automatically Exploiting an Executable

# 8 CONCLUSION AND FUTURE WORK

## 8.1 Conclusion

Our objective was to initiate the implementation of four modules of a novel open source cyber reasoning system that could effectively utilize the technical advancements in binary analysis achieved since the Cyber Grand Challenge sponsored by DARPA and the present time. The paper provided a concise overview of the notion of a CRS, which was expounded upon in greater depth in a related dissertation [13]. The author provided a comprehensive overview of each module within the CRS, including the dataset, attack surface approximation, vulnerability detection, and automatic exploit generation. The presentation included the purpose of each module, its connections with other modules, theoretical considerations, architecture, and technical implementation.

The available interfaces for testing the modules were the command line interface and the set of exposed Python methods. The modules were tested manually using the former. The present thesis has introduced an open source dataset comprising vulnerable programs, a subset of which was utilized during the evaluation phase. Although the precision is not unary and the testing has uncovered certain limitations, we maintain confidence that the research presented in this thesis and [13] provides a robust foundation for future endeavors (as emphasized in the subsequent section of this chapter), whether undertaken by fellow students or members of the open source community.

## 8.2 Future Work

Due to the goal of creating a proof of concept for an open-source cyber reasoning system, we imposed the limitations presented in the chapter 2 such that our implementation could reach the desired maturity and wholeness. Firstly, the possible future improvements may target the fundamental way in which OpenCRS works:

- New executable formats: PE, Mach-O;
- New CPU architectures: `x86-64`, `amd64`;
- New input streams: network packets;
- New programming languages; and
- Unification of all modules' configuration into one file.

Besides this, the changes may improve the implemented module.

### 8.2.1 Dataset Module

- Static code analysis over the source files included in the test suites, such that the input streams are detected and used as labels in the dataset
- More tests in the suite created by us by importing the Zeratool's binaries and creating small and handmade executables (which can be used in the testing phases because of the execution speed)
- More test suits, such as `cb-multios`[1], a Linux port of the executables used in DARPA's Cyber Grand Challenge

### 8.2.2 Attack Surface Approximation Module

- Improve the accuracy of the binary matching heuristic by searching only in specific subsections of the executable
- Improve the accuracy of arguments fuzzing by studying other execution events available in QBDI
- New technique to detect the usage of an input stream, such as symbolically running the binary and intercepting the input-related (library or system) calls
- Pair-wise testing of arguments, for scenarios of dependency relations
- Migration from mounted Docker volumes to gRPC for the QBDI and Ghidra containers

### 8.2.3 Vulnerability Discovery Module

- Timeout or coverage convergence heuristic for stopping the fuzzing session, for example by leveraging Fuzz Introspector[2]
- Binary rewriting for adding Address Sanitizer, for example with RetroWrite[3]
- Migration from mounted Docker volumes to gRPC for the afl++ containers

[1] https://github.com/trailofbits/cb-multios

[2] https://github.com/ossf/fuzz-introspector

[3] https://github.com/HexHive/retrowrite

### 8.2.4 Automatic Exploit Generation

- Reverification of the created exploit in a sandboxed environment
- Exploitation via format string attacks, a technique that is implemented by the already-integrated `zeratool_lib`
- New exploitation techniques, eventually for other input streams too

# 9 BIBLIOGRAPHY

[1] *afl++*, https://github.com/AFLplusplus/AFLplusplus, Accessed: 2023-06-20.

[2] *american fuzzy lop*, https://github.com/google/AFL, Accessed: 2023-06-20.

[3] *attack_surface_approximation*, https://github.com/CyberReasoningSystem/attack_surface_approximation, Accessed: 2023-06-20.

[4] *automatic_exploit_generation*, https://github.com/CyberReasoningSystem/automatic_exploit_generation, Accessed: 2023-06-20.

[5] Thanassis Avgerinos et al., "Automatic exploit generation", în *Communications of the ACM* 57.2 (2014), pp. 74–84.

[6] Teresa Nicole Brooks, "Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems", în *Intelligent Computing: Proceedings of the 2018 Computing Conference, Volume 2*, Springer, 2019, pp. 1083–1102.

[7] *C Test Suite for Source Code Analyzer v2 - Vulnerable*, https://samate.nist.gov/SARD/test-suites/100, Accessed: 2023-06-20.

[8] Sang Kil Cha et al., "Unleashing mayhem on binary code", în *2012 IEEE Symposium on Security and Privacy*, IEEE, 2012, pp. 380–394.

[9] Bogdan-Andrei Cîrceanu, *Symbolic Execution as a Vulnerability Detection Technique Within a Cyber Reasoning System*, Bachelor's Thesis, 2023.

[10] Dan DaCosta et al., "Characterizing the 'security vulnerability likelihood' of software functions", în *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, 2003, pp. 266–274.

[11] *dataset*, https://github.com/CyberReasoningSystem/dataset, Accessed: 2023-06-20.

[12] Jiahao Fan et al., "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries", în *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 508–512, ISBN: 9781450375177, DOI: 10.1145/3379597.3387501, URL: https://doi.org/10.1145/3379597.3387501.

[13] Claudiu Ghenea, *OpenCRS: Cyber Reasoning Systems, Vulnerability Analytics and Automatic Patching in Binaries*, 2023.

[14] Anup K Ghosh, Viren Shah și Matt Schmid, "An approach for analyzing the Robustness of Windows NT Software", în *Proc. 21st National Information Systems Security Conference, Crystal City, VA, USA*, 1998, pp. 383–391.

[15] Abhilash Gupta, Rahul Gopinath și Andreas Zeller, “CLIFuzzer: mining grammars for command-line invocations”, în *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1667–1671.

[16] Kyriakos K Ispoglou et al., “Block oriented programming: Automating data-only attacks”, în *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1868–1882.

[17] Tiantian Ji et al., “The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques”, în *2018 IEEE third international conference on data science in cyberspace (DSC)*, IEEE, 2018, pp. 53–60.

[18] *Juliet C/C++ 1.3*, `https://samate.nist.gov/SARD/test-suites/112`, Accessed: 2023-06-20.

[19] *Juliet Test Suite for C/C++*, `https://github.com/arichardson/juliet-test-suite-c`, Accessed: 2023-06-20.

[20] Ahcheong Lee et al., “POWER: Program option-aware fuzzer for high bug detection ability”, în *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2022, pp. 220–231.

[21] Pratyusa K Manadhata et al., *An approach to measuring a system's attack surface*, rap. teh., CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2007.

[22] *Mayhem for Code*, `https://forallsecure.com/mayhem-for-code`, Accessed: 2023-06-20.

[23] *NIST's C Test Suite*, `https://github.com/CyberReasoningSystem/nist_c_test_suite`, Accessed: 2023-06-20.

[24] *opencrs_dataset*, `https://github.com/CyberReasoningSystem/opencrs_dataset`, Accessed: 2023-06-20.

[25] Ștefan Pană, *Exploit Detection using System Calls Tracing*, Bachelor's Thesis, 2023.

[26] Razvan Raducu et al., “Collecting Vulnerable Source Code from Open-Source Repositories for Dataset Generation”, în *Applied Sciences* 10.4 (2020), ISSN: 2076-3417, DOI: `10.3390/app10041270`, URL: `https://www.mdpi.com/2076-3417/10/4/1270`.

[27] *Shellphish*, `https://github.com/shellphish`, Accessed: 2023-06-20.

[28] Alexey V Vishnyakov și Alexey R Nurmukhametov, “Survey of methods for automated code-reuse exploit generation”, în *Programming and Computer Software* 47 (2021), pp. 271–297.

[29] *vulnerability_detection*, `https://github.com/CyberReasoningSystem/vulnerability_detection`, Accessed: 2023-06-20.

[30] Luhang Xu et al., “Automatic exploit generation for buffer overflow vulnerabilities”, în *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, IEEE, 2018, pp. 463–468.

[31] *Zeratool*, `https://github.com/ChrisTheCoolHut/Zeratool`, Accessed: 2023-06-20.

[32] *zeratool_lib*, `https://github.com/CyberReasoningSystem/zeratool_lib`, Accessed: 2023-06-20.