Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Anders Aasrum Milje

Detecting Malicious Python Packages

In the Python Package Index (PyPI)

Master's thesis in Master in Information Security
Supervisor: Katrin Franke
June 2022

NTNU
Kunnskap for en bedre verden

Anders Aasrum Milje

Detecting Malicious Python Packages

In the Python Package Index (PyPI)

Master's thesis in Master in Information Security
Supervisor: Katrin Franke
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Abstract

Sharing open-source software is a common practice in large parts of modern software development. Sharing software saves countless resources for developers, not only because it cuts down on development time but also because it is more robust. Not every developer has the expertise to implement advanced algorithms, for example, machine learning libraries. Often this re-use of software is more reliable due to the large number of persons who contribute and verify that functionality works as intended. On the flip side of the coin, hostile actors could potentially hijack or upload new software packages injected with malicious code. Malicious contributions that are not discovered in a timely manner in such packages could cause huge amounts of damage, especially if the package is popular.

This thesis aims to improve on state-of-the-art detection of malicious packages in the Python Package Index (PyPI). We did this by developing a new method for static analysis of potential malicious packages, as well as providing an overview of the current state of security in PyPI, and software package managers in general. We did this by building on recent research in the field of software supply-chain security and package managers. We then test an implementation of the method on a dataset containing both malicious and benign software packages. We use genetic algorithms to optimize parameter sets, which result in two sets of parameters where we see potential. The first set of parameters resulted in a 60% F-score and 222 alerts when experimenting on our dataset of 441 benign and 97 malicious packages. With the same dataset, the second parameter set results in an 8-9% F-score but generates zero alerts. We conclude that our first parameters are suited for small-to-medium scale analyses of one or more Python packages and their dependencies. Our analysis of the second set indicates it could be suitable for large-scale scanning of malicious packages, for example, in PyPI. Still, more research is needed on this topic. As the last contribution, we provide our implementation of this new method as open-source software [1].

Sammendrag

Deling av åpen kildekode er en vanlig praksis i store deler av moderne programvareutvikling. Å dele programvare sparer utallige ressurser for utviklere, ikke bare fordi dette reduserer utviklingstid, men også en kan være mer sikker på at koden er robust. Ikke alle utviklere har ekspertisen til å implementere avanserte algoritmer, for eksempel maskinlærings-biblioteker. Ofte er denne gjenbruken av programvare mer pålitelig, på grunn av det store antallet personer som bidrar og verifiserer at funksjonaliteten fungerer slik som forventet. På baksiden av medaljen kan fiendtlige aktører enten kapre eller laste opp nye programvarepakker som er injisert med ondsinnet kode. Ondsinnet kode som ikke blir oppdaget tidlig nok kan forårsake enorme mengder skade; spesielt hvis pakken er populær.

Målet med denne oppgaven er å forbedre den nyeste forskningen på detektering av skadelige pakker i Python sitt pakke-register (PyPI). Vi gjorde dette ved å utvikle en ny metode for statisk analyse av potensielle skadelige programvarepakker, samt gi en oversikt over den nåværende sikkerhetstilstanden til PyPI, og i andre pakke-registre for programvare generelt. Vi gjorde dette ved å bygge på nyere forskning innen verdikjede-sikkerhet (supply-chain security) i åpen kildekode og sikkerhet i pakke-register for programvare. Vi tester deretter vår implementasjon av den nye metoden på et datasett som inneholder både skadelig og godartet programvarepakker. Vi bruker genetiske algoritmer for å optimalisere parametersett, som resulterer i to sett med parametere som vi ser potensiale i. Det første settet med parametere resulterer i en 60% F-score og 222 varsler som genereres når vi utfører eksperimenter på datasettet vårt med 441 godartede og 97 ondsinnede pakker. Det andre parametersettet, med samme datasett, resulterer i en 8-9% F-score, men genererer null varsler. Vi konkluderer med at våre første parametere er egnet for små- til middels-skala analyse av en eller flere Python-pakker, og pakker de avhenger av. Vår analyse av det andre settet indikerer at det kan være egnet for analyser i stor skala, for eksempel til skanning etter ondsinnede pakker i PyPI, men mer forskning er nødvendig på dette området. Som et siste bidrag deler vi vår implementasjon av denne nye metoden som åpen kildekode [1].

Acknowledgements

I want to express my special thanks to all the people who supported me throughout my degree and my thesis.

I want to thank Prof. Katrin Franke for her valuable guidance and feedback throughout the thesis and for helping shape my focus. This thesis would not be without her interest in the topic, for which I am grateful. I also want to thank the lecturers and staff for their time and knowledge they have shared throughout our courses.

I especially want to thank my wonderful girlfriend, Elisabeth Søvik, for always supporting me, being understanding, and providing feedback on my work. I truly appreciate it. I want to thank my good friend Bartosz Tracz for always taking the time to discuss what I am currently working on, as well as my good friends and roommates Isaksen, Jannis and Trym. I would like to thank my mom, my sisters, and the whole family for love and support. Last but not least, I want to thank my fellow students and friends for feedback, discussions, and support over the course of my degree.

The last few years have been difficult for many, so I appreciate you all for making mine easier.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

This chapter will introduce how software developers use open source software registries today, then go on to introduce the problem of thwarting malicious activity and keeping these registries safe to use.

1.1 Background

If you yourself are a developer or know someone who is, there is a big chance they have been using other people's code in their work. The modern approach to software development revolves a lot around using shared code from other software developers or engineers for example. The reasoning behind using existing code is that it saves time. This shared code could be a library for machine learning, advanced mathematics, linguistics, some complex data structure, or something entirely else. Sharing pre-existing code between developers saves countless hours, not just because it saves development time, but also because not all developers and engineers have the expertise to implement machine learning algorithms correctly for example. This shared code is often maintained either by programmers working at companies or by volunteers who spend their spare time giving back to the community. If you pick a reasonably popular, well-backed, or long-running project, chances are this code has a high guarantee of working as expected. It might even come with documentation, tutorials, and a support forum. Figure 1.1 shows a simple fictitious example of how dependencies of a project might be structured. The examples are meant to illustrate the point that only using a couple of dependencies for your work could include many more than at first expected.

All these upsides contribute to being more productive as a developer, with hopefully less time spent debugging advanced functionality.

For the past decade, there has been an emerging phenomenon called *social coding*[2]. One popular example of a social coding platform is GitHub [**github**]. GitHub allows any user to register and upload code, where the goal is usually to be able to share this with other developers. If you can use this code is a matter of licensing, though, which is out of the scope of this thesis. This platform allowed the bar to share code to be lowered substantially. You no longer had to set up a

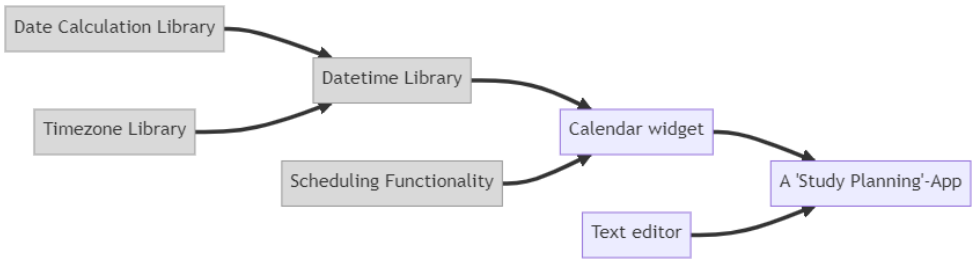

Figure 1.1: A fictitious example of dependencies in a *Study Planning* app. The grayed-out dependencies are ones that might be hidden from the developer because of their depth.

website, chat group, or email list to distribute or share your code [1]. Today it is also very simple to share a piece you just developed online.

In June of 2019, one of the largest registries for software packages registered over 1.3 million uploaded packages [3]. In this registry, you can find functionality for a good chunk of what you want to accomplish in your work. It is easy to use code you just found, but we should stop considering the consequences of using it just before we integrate it into our project. Further, in the thesis, we will discuss the downsides of having such a plethora of code and functionality at your fingertips, raise some awareness, and show what is being done to make this plethora of code safer to use.

The increased usage and reliance upon such code-sharing platforms, often called registries, while also asking the question if it is *reliable* also introduces the problem of *trust* and *security* of its usage. Code shared in these registries is often called *packages* or *modules*. A package can be a small piece of code, a library, or an entire solution ready for use. This means the size of a package can vary substantially, yet they are all equally easy to download and use in your work. Packages such as these are no stranger to vulnerabilities or bugs just like any other software.

The downside to using packages without proper forethought is that they might have security vulnerabilities, or that their handling of those security vulnerabilities is not up to the standards required of your work. We often refer to packages used in existing work as *dependencies*. Dependencies, just like your own software, could include bugs or security vulnerabilities of varying severity [2]. There are also examples of dependencies being deliberately uploaded with malicious content, or have the dependencies compromised and injected with malicious code [3].

A common view on security in open source is that more eyes mean fewer vulnerabilities. This is called Linus' Law and was investigated by Prana et al. in their

[1] Or sort cards and punch it into a machine for that matter

[2] https://arstechnica.com/information-technology/2021/09/npm-package-with-3-million-weekly-downloads-had-a-severe-vulnerability/

[3] https://www.vice.com/en/article/dypeek/open-source-sabotage-node-ipc-wipe-russia-belraus-computers

recent work where they found no correlation between better or worse handling of security vulnerabilities based on a number of contributors [4]. Their contributions will be highlighted more in the next chapter.

A different study found that security regarding such dependencies is often enforced if it is a company policy, or if the company has a pre-approved library open-source library they require developers to use [5]. Prana et al. also recommend an approach of having a list of approved libraries the developers can use in their work. This may of course be difficult in some scenarios, and dependency management has its own research.

A common word for talking about dependencies in the project is *the supply chain*. A *supply chain attack* is sometimes used to refer to compromised packages. The European Union Agency for Cybersecurity (ENISA) describes in their threat landscape report what constitutes such a supply chain attack [6]. The key takeaway here is that such an attack requires two attacks; (1) one on the supplier, and (2) one on the customer. They also attributed around 50% of attacks to advanced persistent threats (APT), which are typically regarded as state-sponsored or well-resourced criminal enterprises [7, p. 341].

In the next part, we will discuss how these concerns around security and the use of open-source dependencies relate to the work in this thesis.

1.2 Motivation

While there is research in the area of dependency management and for example how long vulnerabilities are undetected in open source code, there is a growing interest in looking at dependencies with malicious intent. The motivation for this work is to look at the problem of security vulnerabilities through the lens of it being malicious code. The difference between a vulnerability and malicious code is only the *intention*.

Malicious code will from this point onward in this thesis indicate that the code is malicious by intent, and not by accident, such as a bug made by a developer. If a bug is introduced with the intention of being malicious, then it will be regarded as *malicious* by our definition.

We will therefore in this thesis explore the process of detecting potentially malicious packages in the Python Package Index (PyPI). This repository is the official one for the Python programming language. With over 370'000 projects contributed by over 580'000 users [8], having over 500 million downloads every day [9], and a full clone of all released packages amounting to 10.9 terabytes of storage [10], it can be regarded as one of the largest repositories used by developers.

Franke et al. in the book *Digital Forensics* [11, p. 313] highlights the issue of the huge amount of unstructured data digital investigations face today as a key challenge. We can draw some parallels from digital needle-in-a-haystack problems, with the task at hand in this thesis, even though this project does not investigate a police case or similar, there are some parallels when it comes to the sheer amount of data and the difficulties of sifting through it.

Now knowing more about the subject of inquiry, we will next talk about the impact our work has with this motivation in mind.

1.3 Expected Impact

Through the work in this thesis, we aim to advance the state of the art in detecting malicious packages using static analysis. We expect to show an experimental design to alert on malicious activity inside packages while testing using a relatively small dataset.

The second goal is also to present findings from the literature and recent studies on vulnerabilities in open source registries, as well as the existing research on packages or dependencies that are regarded as malicious and the current techniques for finding such code.

The last goal of this work is to provide an easy-to-use and extendable implementation for developers and security analysts to use. We want to provide this implementation as open source. The future work section of this thesis will discuss improvements that could be made to this, as well as other techniques and ideas that would be interesting to explore in the future. Our impact is made through answering our research questions.

Research Questions

Tackling the problem of knowing whether to trust this piece of code on the internet in your own software is difficult, and this project expands on existing research done in this field. To find solutions to this problem, this thesis explores the following research questions:

- RQ_1 What is the current state of security for the python package index (PyPI)?
- RQ_2 How can we improve the state-of-the-art detection of malicious code in the Python package index (PyPI)?
- RQ_3 How can we give developers more autonomy in ensuring that their applications are safe from compromised dependencies?

1.4 Overview

After this introduction to prepare the reader for the coming chapters, we introduce some background on the ecosystem we are working in, as well as some concepts that are important to cover such as what a package is. The chapters have been written in such a manner that reading it from start to finish will be coherent, yet any single chapter can be read without all the prior ones, though they will contain both back and forward references.

Further, in the methodology section, we will introduce *what* we did and *why*. Next, we showcase *how* the methodology was implemented and present the experimental results.

The rest of the thesis focuses on discussing the results and their limitations, as well as giving a conclusion on what we have done. The very last section is on what we think efforts should be focused on in the future. Here we will present what we think will further the implementation made for this thesis.

At the end of the thesis, you will find an appendix with content that was not suitable for any other chapter, or extra information that might be of interest. All appendices are referenced somewhere in the main parts of the thesis.

Next, we will jump straight into the state of the art.

Chapter 2

State of the Art

Below we have divided the chapter into two parts. First introduces theory which is necessary to understand the previous work in the field, as well as to help in understanding the methodology later. This includes a bit about the Python programming language and the ecosystem that surrounds it.

The second part of the chapter presents previous work done in the field of finding malicious packages in both PyPI (more on this below) and other open-source registries, as well as presents different techniques used to find malicious packages and provides.

2.1 Theoretical Foundations

This section provides background on what a package is, what place this has in package managers and the difference between different package managers. Further, we introduce the Python programming language and its official package manager. Then close on some theories related to malware detection and analysis.

2.1.1 Python

Python is the language in which the malware in this report is written. It is an interpreted high-level dynamically typed programming language. Dynamically typed means it can figure out the type of a variable based on its contents, in contrast to strongly typed languages where you have to define an int variable for integer content. Therefore, it is an advantage to understand a bit about what the Python programming language is and how it works.

Another critical design decision is necessary to cover; the Python programming language uses indentation to determine scope by default. This means text representation programs often follow a pattern because it *needs indentation* to structure itself. Code can sometimes be *minified*, which is compressed to for example a single line of code. Python does have functionality for this, and supports semicolons ending statements, but is not used much.

The simple syntax makes it the first language to learn for many students and interested hobbyists. Python is used for both web applications, data science, and scripting[1].

The Python programming language is governed by the Python Software Foundation (PSF) as a non-profit corporation. The PSF is sponsored by large corporations such as Microsoft, NVIDIA, Huawei, Meta/Facebook, etc [12].

If you want to share Python code via the open-source registry, you would probably use the Python Package Index (PyPI). In this registry, we regard a piece of code for a *package*, described further below.

2.1.2 Packages

Most programmers are familiar with the concept of libraries or modules, a way of encapsulating functionality such as classes, interfaces, and functions in a manner that makes it easier to reuse in other projects. A package is a collection of libraries, executable programs, or scripts that are useful for others. A package, therefore, does not have to be a library intended for re-use. It could be a command-line application for converting audio for example.

2.1.3 Package Managers

Those familiar with Unix-like operating systems such as Debian, and CentOS might be familiar with package managers such as *apt* or *yum*. Packages found in these package registries provide both libraries and fully-fledged programs such as Firefox. Let us call these user-space package managers (USPM). This thesis will focus on programming ecosystem package managers (PEPM). The main difference between package managers in programming ecosystems is that is programming language agnostic. While PEPMs are solely for that specific operating system.

Some package managers are what could be called official. For example, the Ubuntu Foundation distributes both the operating system and controls the apt package manager, which is why you would call apt the official package manager for the Ubuntu distribution. An example of a non-official package manager is Homebrew [13], which is not officially controlled by the same governing entity (in this case Apple) that controls the operating system it runs on.

This report works with what we call PyPI, which is described in the next section. The official package manager for Python, called *pip*, works by downloading packages from PyPI.

Python Package Index (PyPI)

The Python Package Index (PyPI) is governed by the Python Package Authority (PyPA), which is a working group under the Python Software Foundation (PSF) that works on tools used for packaging Python modules and code [10].

[1] Automating tasks or doing simple operations not worthy of a full software solution.

2.1.4 Compiled vs. Interpreted Malware Analysis

To better understand the types of malware, and the techniques used in this report we will present the difference between analyzing *interpreted* and *compiled* malware.

We will firstly begin covering *compiled* malware. Firstly, when code is compiled it loses its text representation, which is the most important difference for us. In short is malicious code being compiled to machine code, or bytecode meant to run in some higher-level virtual machine or runtime. Some examples of machine code compiled languages are C, C++, and Go. Examples of languages compiled to bytecode could be Java, which is compiled to the Java Virtual Machine (JVM), or C#, which is compiled to the .NET runtime. This generally means the compiled code can run right on your computer. It might need a runtime or some other libraries installed for it to work, though.

In comparison, we have *interpreted* malware. This is code that has no need to be compiled and will be interpreted by another program when it is time to run. Popular examples of this are JavaScript and Python. The finished code is left in its text representation[2]. The Python language requires you to use the *Python compiler* to basically compile the language as it is run. Figre 2.1 shows the execution model for a compiled and interpreted language.

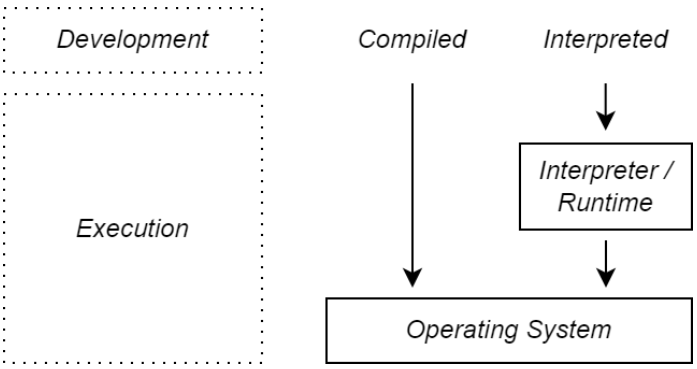

Figure 2.1: A graph showing how an interpreted language requires another dependency for being able to execute on an operating system. A compiled language is able to run directly on the (supported) operating system.

When malware is compiled to machine code, for example, you lose the plain text code, which would make the program easier to understand. Doing a static analysis of such malware entails reading assembly code. Doing such static analysis of malware in an interpreted language like JavaScript or Python would be reading the actual code being run. This code could of course be obfuscated in some shape or form to hinder analysis. The dynamic analysis would look similar; looking at what artifacts the execution of the code would manifest on a computer.

Python is considered an interpreted language, but it does have a *bytecode* ele-

[2]Not accounting for minification or obfuscation done when releasing the code.

ment to it, which is important to cover.

How the Python Interpreter Interprets

To introduce bytecode for Python we first have to take a quick detour to investigate how Python code is being run.

When a Python program is being run by the interpreter, the program used to execute program programs, the code is first parsed into tokens through lexical analysis. This process converts information like 'class HelloWorld', into the tokens LOAD_GLOBAL and CALL_FUNCTION, for example. The next step would be to create an abstract syntax tree from the list of tokens created in the previous step.

Abstract Syntax Tree (AST)

An abstract syntax tree (AST) is very useful for understanding a piece of code from a computational perspective. The code is reduced into its most, well, abstract form. It is a standardized representation of Python code that is agnostic of code structure and naming schemes.

In chapter 3 we will discuss how we traversed such an AST.

2.1.5 Python Bytecode

After the Python interpreter has converted text into an AST, it then converts this tree into a bytecode representation. This is similar to assembly, where you have rudimentary instructions for operations such as loading a variable, storing a variable, and adding or multiplying. This bytecode is then run by the python interpreter at runtime. Figure 2.2 shows the higher-level concepts of the Python interpreter.

Python is most often shared in its text representation, but the official python interpreter does have a caching function that stores the *compiled* code locally. The language also has integrated functionality to compile and run new code at runtime inside a program. This means that loading Python code from an already running Python program is easy to achieve. In the section regarding datasets, we will see why knowing this is possible is important.

Now that we have covered a bit about how the ecosystem for Python works, and how the code is run, we will look into some relevant literature on malware.

2.1.6 Types of Malware

Sikorski and Honing in their book *Practical Malware Analysis* [14, p. 3] explain the most common terms for malware. Since the release of the book, there have been some new additions of malware, such as *ransomware* and *coin-miner*. The authors mention that during analysis you should not get too invested in picking the proper category for your malware, but for us, it gives us an easier way of talking about threats and explaining our dataset.

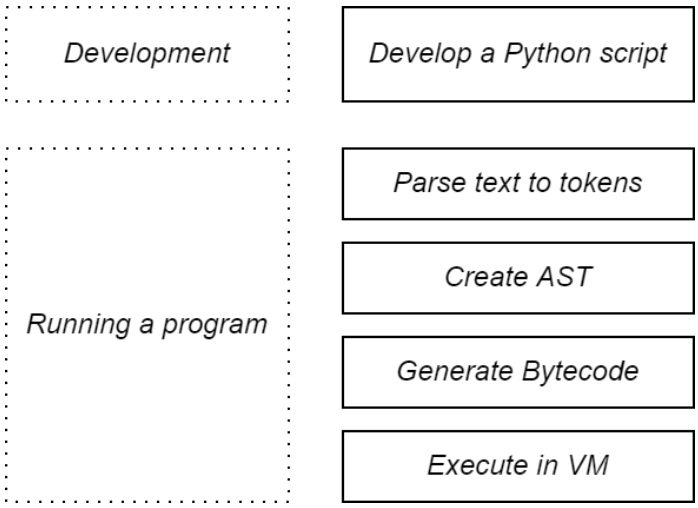

Figure 2.2: A diagram showing the higher level steps of the Python interpreter when you run a Python program from its text representation.

The first term we want to explain is a *trojan* or *backdoor*, this type of malware is a type that installs itself on a computer to give (illicit) remote access to an attacker. A backdoor could be a misconfiguration in software that allows access, such as standard passwords for logins, and a trojan would be some sort of malicious program that grants remote access through its own functionality.

The next term we want to describe is *downloader*. This is a definition of a malicious program whose intention is to download another malware. The downloader itself does not have the malicious functionality you would expect from a trojan, and its only purpose is to fetch a malicious payload and somehow install or execute it.

The last definition we will present is *information-stealing malware*, this type of malware does not have the fully-fledged functionality to take remote control of your computer, but instead collects and ex-filtrates certain information it is programmed to find. This could be local documents, network information, locally stored cryptographic keys, or anything else the program might have access to.

The last part of our theory quickly covers how malware is generally detected and analyzed.

2.1.7 Malware Detection

Before we can analyze malware we first have to detect it. Malware detection is the act of trying to detect malicious code on a computer or in a network. This might be done in several ways depending on if you are monitoring a computer, network, or both. There are two theoretical definitions we will use to separate malware detection. Stallings describes these terms in their book on network security essentials [7, p. 382].

The term *signature detection*, or *rule-based detection*, works by writing rules

or searches to find similar-looking pieces of programs or network traffic. This is often done by finding unique strings in known malware, such as IP addresses or domains used for communication, or messages inside the program. We can also match on a set of unique bytes that would identify a specific program containing malware. The same would apply to network traffic, where an analyst would try to find unique features for the network traffic a certain malware would produce when communicating with the outside. Two downsides of rule-based detection are that even simple text obfuscation can counter it and that the malware has to be known ahead of time[3].

The second term *anomaly detection*, or sometimes *statistical anomaly detection*, is used to describe the act of detecting malware based on it doing things that are not normal in your network or on a computer. This type of detection could detect unknown malware, in contrast to rule-based detection, which is a big advantage. Such detection can either use thresholds to determine when some activity is suspicious enough to get flagged for analysis or create an *activity profile* for a user, network, or program, and statistically determine when that activity falls out of its normal operation.

In the methodology chapter, we will see how this report uses both rule-based and anomaly detection with thresholds to try to detect malicious code. First, we will shortly explain some malware analysis techniques.

2.1.8 Malware Analysis

The book *Practical Malware Analysis* by Sikorski and Honing gives a great introduction to malware analysis techniques [14]. We will introduce the relevant terms from it such that we can better explain the methodology and reasoning in this paper.

The two most basic terms we will introduce are *static* and *dynamic* analysis. *Static* analysis entails looking at the program to understand it without running it. The book describes this in the context of analyzing *compiled malware*. As previously mentioned our Python malware is not compiled when shared, and is, therefore (mostly) in its text representation form, so this works a bit differently for us, but the core concept is the same; the program is not run during static analysis. This is also the technique used for this report, but more on that in a later chapter.

The second technique called *dynamic* analysis executes the program to figure out how it actually behaves when executed. During execution an analyst would look at certain parts of the computer, such as the file system, to determine if any new files were generated by the program. The dynamic analysis also fully omits any obfuscation used in code to make it harder to understand during static analysis. The downside is that dynamic analysis actually requires one to run the program on a computer in a controlled environment. Some malware also employs techniques to stop execution if it detects it is running in a closed environment,

[3]You can, of course, create rules for malware you think will arrive in the future, but few of us possess the ability to foresee the future with such accuracy.

which would be used for analysis. This report does not perform dynamic analysis, yet we will discuss it in related and future work.

Malware analysis is itself a field of study, as a consequence, the nuances of basic and static analysis and their up-and downsides are omitted from this section, as it would require more room in this report than we have time for. The previous book on *Practical Malware Analysis* goes deeper into this topic if you feel interested in learning more.

Finishing on the theory we move on to present related work.

2.2 Literature Analysis and Related Work

This section will present previous and related work in the field. The work is mostly centered around the security of large open-source registries like PyPI and NPM[4]. We would still consider this an emerging field that is increasingly relevant.

The field of studying open source software has had a fair amount of published works in recent years; a lot of these focus on vulnerable packages, how long they are present, what variables might affect them, and if a user is affected by a security vulnerability [4, 15–18]. There is also a lot of research on detecting backdoors in PHP, but we only cite one paper that focuses on finding these in Python code [19]. This means there are not a lot of relevant works to choose from, and we try to cover all the published works that are relevant to us.

We used both the Google and Google Scholar search engines to get the biggest reach, and the keywords we used to locate our sources were "package managers", "supply chain attacks", "open source/open source registries", "dependencies", and "security testing". The two strings we used that yielded the most results were "security testing open source dependencies" and "detecting malicious code python".

After finding the initial papers we used their references to locate more papers within the field. When we started seeing the same papers several times, we began to feel like we had found the most impactful works. Since the field of supply chain attacks and open source dependencies has many recent additions, we recognize there might be newer works that we have missed in the process of writing this thesis.

We would also like to point out that most books that were cited in this thesis came from courses we have had over the past years.

2.2.1 Vulnerabilities and the Security of Open Source Registries

Decan et al. released a paper in 2018 where they did an empirical study of 400 security reports over a six-year period in the NPM registry. At this time the registry contained over 610'000 packages. Their analysis concluded that the number of vulnerabilities is growing over time and that they generally take a long time to get fixed by the developers. They mention that it takes about 52 months for 50%

[4]For the NodeJS-ecosystem - Meaning JavaScript and TypeScript mostly.

of all low severity vulnerabilities to be fixed. Other findings include that older packages (over 28 months old) get the most vulnerabilities reported. While these security issues get fixed pretty quickly after they are discovered, a large portion of packages is still vulnerable due to upstream packages (which means packages they themselves are dependent on). They mention the use of too restrictive version locking as something that might halt an automatic update of a dependency in a package to a safer version. One of their actionable recommendations is that developers report their vulnerabilities to a common database, such that others can check their own projects [15].

Pashchenko et al. bring up the discussion that a security vulnerability in a library should be considered an issue if it is not being used in your application. They argue a lot of resources might go to converting code to another version or library because the functionality that is not being used is a vulnerability. This also shares similarities with halted libraries, which are also discussed. Halted libraries are libraries that are no longer maintained. Any security vulnerability in these would mean one has to create their own fix to the issue or migrate to another library. In their analysis of 200 popular Java open-source libraries, they also found that developers are able to fix 82% of vulnerable dependencies in their projects [16].

Ponta et al. recognized the need for a tool to analyze if vulnerable parts of a library were actually reached by their own code. They found their developers, especially the security aware of them, were reluctant to use meta-data tools, as these can generate many false positives for vulnerabilities that are not used by their application. The same developers also feared using yet another static analysis tool. They then developed an internal tool that utilized both static and dynamic analysis to assess if a project used a vulnerable part of another dependency to give more precise feedback to the developers [17].

Prana et al. found that certain metrics from projects, such as activity level, popularity, and developer experience did not turn into worse or better handling of vulnerabilities in dependencies [4]. They investigate the claims that fewer or more developers on a project improve its security, and found that you are likely better off reducing the number of direct dependencies instead of recruiting more developers or personnel. A solution they present is to use a bigger library that does more and has a good track record of updating their code. They mention the use of software composition analysis (SCA), which can be used to identify dependencies of projects and whether or not they have security vulnerabilities reported in them.

Alfadel et al. did an empirical analysis of vulnerabilities in Python packages [18]. They found an increasing number of vulnerabilities being discovered, and that they can take up to 3 years before being discovered (on median), regardless of their severity. This matches well with the findings from Decan et al., which we introduced at the beginning of this section. They also mention a study where a high number of projects were actually safe because the project did not utilize the vulnerable code, which again brings up the discussion on vulnerability reporting and its preciseness of it. Further, they bring up DependaBot, which GitHub has

required. This is a bot that automatically checks dependency configurations of projects and updates them.

2.2.2 Typo- and Combo-Squatting for Dependency Confusion

One of the first works we found in the literature search for this project was a project by Tschache [20]. They performed a practical experiment by generating fake package names based on real ones using Levenshtein distance to trick unsuspecting developers to download their decoy packages. They uploaded their own fake package with a callback function, which was executed on installation by the developer. They found that 17'000 install would have executed their code, which could have potentially been malicious. They found that 50% of these installs were done with administrator rights and that much-used packages could be susceptible to typo-squatting attacks if they for example change their name between versions. Their work has been cited by many papers we have cited in this report. While Tschacher used *edit distance* to their own typo-squatted packages, Duan et al. [21] in their work used it to track down suspicious packages by measuring how many packages were alike official ones.

The problem that Python packages also had, and much still have, is the ability to execute code on installations. This is meant as a feature for compiling C-code extensions for example, but this can of course be used maliciously.

Vu et al. present methods to identify suspicious packages by measuring their likeness to built-in modules (which should not be on PyPI) and other popular packages. They talk about both typo-squatting, which we have already mentioned, and combo-squatting, which is a term for combining words in package names using hyphens, or rearranging words. They put attack vectors for supply chain attacks into two different categories, where one way is to hijack the author's credentials and upload a malicious version, while the other is to upload your package and somehow trick the users into downloading it [22].

2.2.3 Code Security Testing for Python

Although not literature, the nature of the project means we also need to get practical to solve the issue of finding malicious packages. This section is dedicated to some tools and services we have found that could increase the security of a project by comparing them on some accounts. This is also to give the reader an introduction to what already exists in the ecosystem for developers.

A third-party application and service called *Safety* will check your requirements (e.g. dependencies) in your Python project and report on known vulnerabilities in versions that have been added to their database. It is administered by a Canadian cyber security company specializing in supply chain security. The tool is solely based on metadata of dependencies and already known vulnerabilities [23]. *Requires.io* that provides the same kind of concept as a service [24]. DependaBot, which was mentioned earlier by Alfadel et al., is a bot now owned by GitHub to

automatically update dependencies in the project by creating *pull requests*[5][25].

Bandit is a static analysis tool for detecting security issues in Python code by analyzing the abstract syntax tree (AST). Bandit runs on plugins and then gives a report back to the developer or analyst [26]. They have a plugin system that makes it easy to extend. *Detect-secrets* is another command-line application to detect strings and *secrets*[6] in your project. This is useful to run before committing your code to a public repository, or when doing a security analysis of open-source software for example.

There are also services that specialize in providing security analysis of code as a service. Services such as *JFrog*[27] and *Snyk*[28] are some examples to name a few. Snyk operates in the same realm as DependaBot and requires.io (which we previously mentioned) to analyze dependencies of projects and to give feedback to the developer on vulnerabilities in these. JFrog has worked to automatically identify malicious packages on PyPI, and has released the results from these efforts as blog posts on their website [29–31]. Many of the packages they discovered are in the dataset we are using in this report.

2.2.4 Static Analysis to Detect Security Issues

We have already presented some solutions and programs that are freely available to use for security testing in Python but will do so for one more section. While we still are in the realm of (mostly) static analysis, we present the work of Micheelsen and Thalmann in their 2016 master thesis [32]. They created a static analysis tool called *Python Taint*[33]. Their tool bases itself on the theoretical foundations of control- and data flows inside Python programs to detect unsanitized inputs. This allowed them to be able to detect SQL injection vulnerabilities by static analysis. They did this by tracking *sources* and *sinks* to determine where user-submitted data was flowing into the application. If some user input flowed into a sink that was determined dangerous (e.g. an *eval* command) the tool gives feedback to the developer on this. The project is now discontinued, but a large social media corporation has incorporated the same concept in one of their *type checkers*[7]. Their implementation called *Pysa* allows security engineers to generically define sources and sinks from previously known security bugs, which can then be used to search the code base for similar data flows [34].

Vu et al. proposed a static analysis technique to detect injected code in published packages. The technique performs metadata mining to locate a public source code repository for a given published package and compares its contents. If there is new code in the published version that is not visible in the source code repository, it should be flagged for analysis or similar [35].

[5] A request to submit a code change in a repository on GitHub

[6] Recognizable tokens, passwords, etc. that are typically used for authentication which should not be publicly available.

[7] A static analysis tool to verify the correct use of types in languages that don't enforce this by default (in languages such as Python for example).

Fang et al. detected Python backdoors using combined features. They recognized that most of the research was for either PHP or for generic web shells and that there existed little research on malicious Python code. They proposed to classify these backdoors with a combination of features. They did so through the use of abstract syntax trees, text statistical features, and opcodes, among others. They also used TF-IDF on opcodes combined with n-gram. They discovered through their work that comprehensive features gave the best result. They trained a random forest classifier which gave them an accuracy of 97.70%. They weigh up the benefits of static analysis by omitting the high amount of resources required for deep learning or dynamic analysis [19].

2.2.5 Dynamic Analysis of Malicious Code

In this section, we discuss more works in the realm of dynamic analysis. Ponta et al., as mentioned previously, also used dynamic analysis for their analysis, combined with static analysis.

Sand, in their 2012 thesis, explored detecting system calls in the lower layers of the operating system to detect malware [36]. Their approach yielded good results using graph-based learning methods to classify malware by creating dependency graphs for API calls being made to the system. They found that memory values were the most reliable of the features they tested, which were also functions, among others. They tested how obfuscation then affected their dependency graphs, for which their implementation proved resilient in most cases, as the techniques implemented by the attackers were mostly developed for signature detection and static analysis.

Kim et al. in their 2012 paper proposed an approach of internal function hooking (IFH) to dynamically analyze JavaScript malware. Their approach is compared to the approach of API-hooking, where IFH can reach other, deeper, parts of the code. With IFH and a symbol table[8] from Microsoft, they were able to apply their framework to Microsoft Internet Explorer (IE) and analyze both obfuscated and non-obfuscated code being executed in the browser [37].

Vasilakis et al. published a paper in 2018 showing a prototype for executing partial parts of one's code in an isolated container. In their work, they used the Docker container solution to run the code of choice in isolation. While this technically is not the dynamic analysis of malware, it is the compartmentalization of code that could potentially be malicious. The idea is to offload the potential security risks of using a library by isolating it in an environment where only the output of a function, for example, is returned. The approach works in practice, while the downside will be how one deals with overhead [38].

Ohm et al. proposed a dynamic analysis tool called *Buildwatch* to analyze software dependencies for newly introduced artifacts. They recognize the need for a tool that connects to the continuous integration (CI) [9] process that is often in-

[8] In short, a dictionary that keeps track of locations of functions, classes, etc, in programs.

[9] Introducing automation of building, testing, and publishing software using services like GitHub,

volved with modern software development. The limitation of their implementation is using a sandbox that will have to run as a separate instance, instead of running directly in the CI pipeline [39].

2.2.6 Scanning Open Source Registries

Duan et al. [21] created an analysis pipeline called *MalOSS* to process over one million packages in three large and popular registries; PyPI (Python), NPM (JavaScript ecosystem), and RubyGems (Ruby). They found seven malicious packages in PyPI, 41 in NPM, and 291 in RubyGems. In doing their qualitative analysis of registries, they created a framework to understand better the needs of package maintainers, developers, and users. They saw the need for notifications services in package installer tools. When PyPI removes a package for any reason, the installer does not do any remediation to remove it. The developer must then use external tools for such. PyPI also had no official advisory DB; however, the previously mentioned unofficial project *safety*[**toolSafety**] can do this for you [10]. The registries did not implement automatic notification to package maintainers, developers, or users when a package was removed for any reason (such as malicious activity or breach of terms of service).

Jenkins, or Gitlab Runners

[10]https://github.com/pyupio/safety

Chapter 3

Methodology

This chapter will present the methodology used in this project. We describe how the dataset was used, the assumptions made, and the techniques employed.

First, we introduce what we mean by *malicious*, which is proceeded by the new detection model we have developed, which is later implemented in code. After this section, we introduce the dataset we tested our implementation on, plus the metrics and pre-processing employed to determine the performance.

3.1 Defining Maliciousness and Malicious Behavior

Our definition of malicious means that someone has published or written harmful code or exploited a vulnerability with *intent*. We note the difference between a *bug* and a *backdoor* as a bug being accidental, and the backdoor is being implemented with intention. Ohm et al. also note this difference [40]. For example, if a package with a critical security-related bug was released and wrecked havoc, it would not be malicious.

Since we cannot judge a single function to be malicious or benign without context, we have to look at malware through a more contextual lens. We divide the common malicious activity of malware into three main parts; network, system, and file system (as shown in 3.1). Figure 3.1 shows a simplified model of how a lot of malware operates, and serves as a demonstration of their behavior. The model therefore does not cover every element observed in all malware. This does not accurately describe all malware, but for our purpose, it works. The model describes general malware that communicates information through a network, attempts to steal information from the file system, and tries to access system settings for information, then ex-filtrates this information back to the attacker.

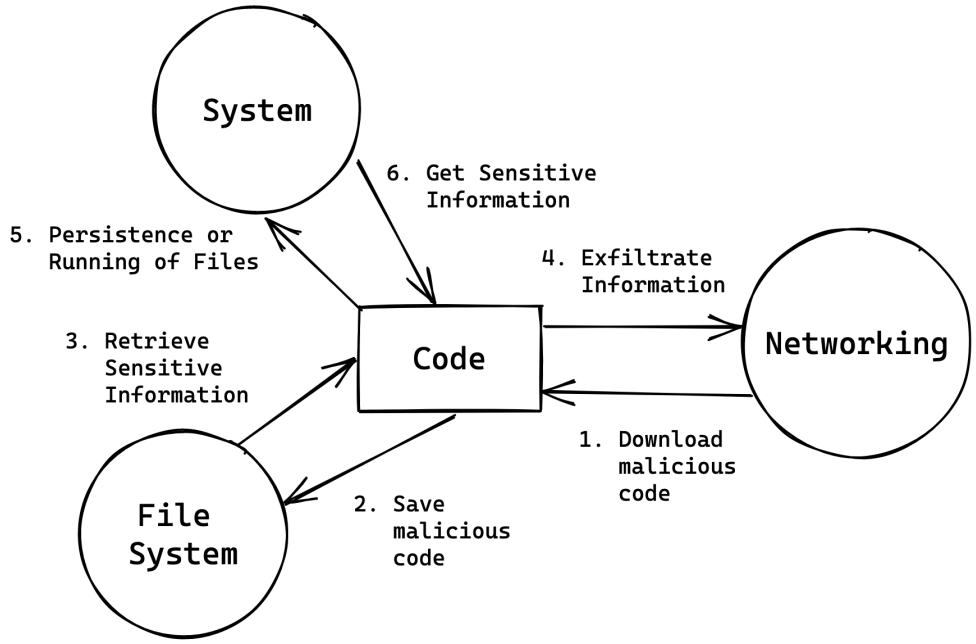

Figure 3.1: A graph we created to show a simplified model of how a lot of malware operates.

3.2 The Detection Model

The detection model we present utilizes our new method, which among others features, focuses on detecting suspicious functions and imported modules using static analysis. We also included rudimentary anomaly based detection, which looks for behavior such as importing modules inside functions. We ground this method in the malware behavior we defined in the previous section. Functions and imports on their own are not malicious, but their usage in combination might exert suspicious behavior not usually found in all benign Python programs. This is in large part inspired by the works of Fang et al. [19] (described in chapter 2.2.4). They received good results by combining features, which is what we aimed to incorporate into our model as well. One difference to note is that Fang et al. used bytecode, we used text representation.

The static analysis we performed was on the text-representation of Python source code, as described in the previous section 2.1.1 on Python. To facilitate our analysis, we first converted the text representation into an abstract syntax tree (AST) to extract information such as functions, imports, and strings. From the AST we also derive some anomalies. We will also show a practical example of an AST later in the chapter when we elaborate on obfuscation in Python.

In figure 3.2 we can see the general processing pipeline for our detection model. We collected various data, then matched it against rules we defined, look for anomalies, and finally try looking for matches with our canaries. Each match

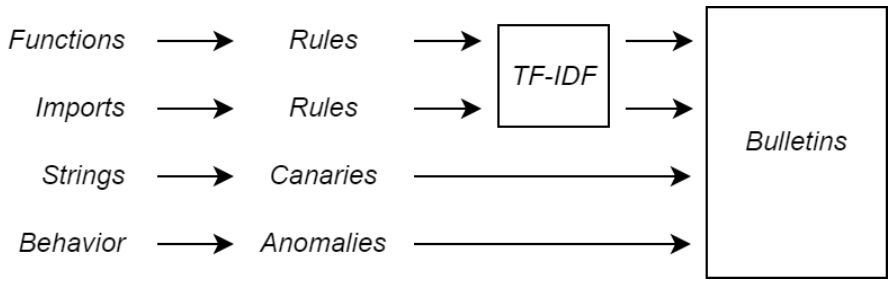

Figure 3.2: A graph showing the general processing pipeline of the detection model work on a conceptual level. Our data is matched against rules, anomaly detection, and canaries to create bulletins. Also, note that TF-IDF affects the bulletins generated for functions and imports.

results in what we call a bulletin, which we will describe in detail later. In short, a bulletin is an alert on some suspicious behavior or code.

As mentioned in the previous chapter on Python bytecode and interpreters; we do not use bytecode features. We used the AST to fetch information such as imports and functions while still keeping some important information about the program, such as structure. We attempted this approach since Python is very dependent on the indentation of its code. In figure 3.3 we can see the idea of how our data is processed from code to data. You can observe the first steps where the text representation is converted to an AST. We chose working with the abstract syntax tree (AST) instead of the bytecode, because compiling the Python code to bytecode would take longer. This is because it involves actual compilation with an interpreter, which we did not use.

After we have fetched functions and imports, we attempt to resolve variable assignments and import aliases. Resolving is something that is usually done by the compiler inside the Python interpreter, but since we did not use any interpreter and only parsed the AST, we had to perform this step ourselves in our implementation. We also performed some rudimentary concatenation of strings, which is an easy obfuscation technique seen in some simpler malware. From the AST we could also detect modules that are imported inside functions and classes, which we will discuss more later. With the data we gathered we implemented simple detection for libraries that are imported dynamically using the *importlib* library, and the *__import__* built-in function. If we look back at figure 3.3 we can see how these steps are combined. We performed the resolving as a post-processing step after we have gathered our data. This graph also mentions *partial evaluation*, which we go into later in this chapter.

The third research question in section 1.3 was formulated after conducting preliminary research on the different tools that are present for Python developers to detect security issues. While there are solutions to give feedback if developers use known outdated dependencies, we saw the possibility of improving on the tools that detect security issues in their Python code as well as being able to scan the PyPI registry. A previously mentioned project like Bandit [26] (see chapter

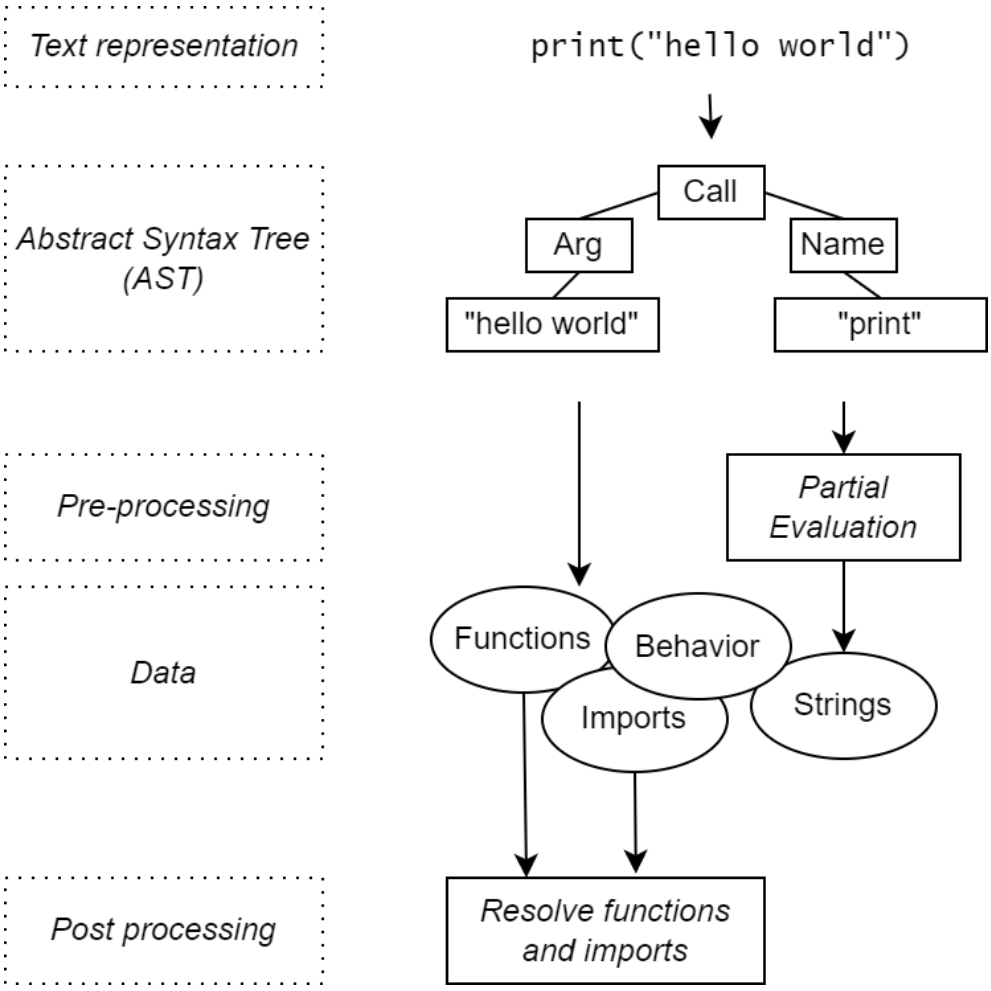

Figure 3.3: A graph showing how the data is processed on a more detailed level.

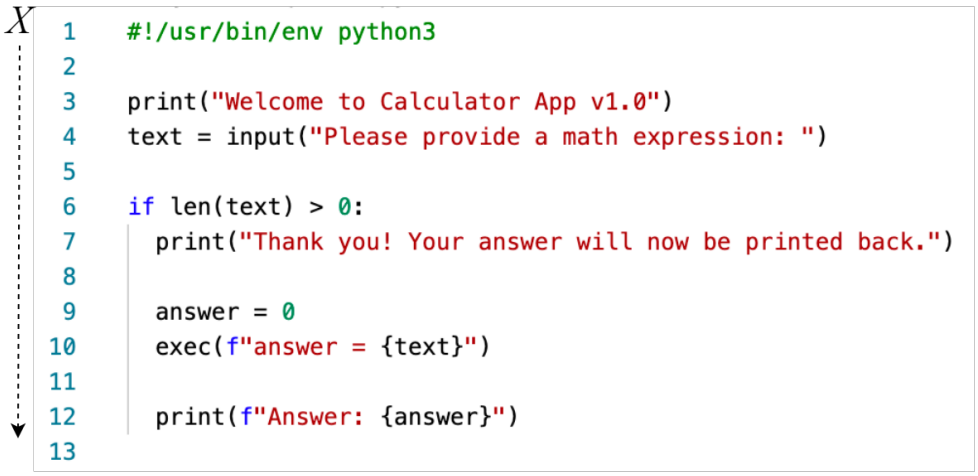

Figure 3.4: This graph shows how the *fields* are aligned vertically with the code. The fields follow the line numbers in other words. - The example code shown is a vulnerable calculator app. The applications is vulnerable due to the use of the *exec*-function to evaluate mathematical expression, as it will evaluate all input as code. The *exec*-function is a common function to look for when searching for malicious activity.

2.2.3) can be used to detect *exec*-calls in Python for example. The implementation in this paper was to some degree inspired by how Bandit works and reports on vulnerabilities.

We wanted to extend this idea of a rule-based static analyzer, but to improve on it to include dependencies of your current project as well. We wanted to continue the usual trend of giving the developer feedback on specific lines of code and why a certain alert was triggered. In other words, we did not want our method to be a classifier at this time. Bandit can check for rules in a current project, but, Fang et al.[19] also used matches against functions names and calls as features in their machine learning model. As it stands, the current implementation supports the functionality to analyse dependencies in packages, but this needs more work to be implemented correctly.

To be able to implement a static analyzer that scales with the amount of code and still gives relevant feedback, even when scanning dependencies, and combines both matching rules and anomalies, we wanted to experiment with a different weighting system. We implemented what we call *fields*, which will be described in the next section. These fields are basically functions on a two-dimensional graph, where the X-axis is the line of code, and the Y-axis is the value of the field. This concept is visualized in figure 3.4 and also includes a security vulnerability.

We wanted to create a versatile technique that could simultaneously give developers the autonomy to check their own code and dependencies for security vulnerabilities, while at the same time lowering the threshold for applying the same rules and detection mechanisms to the PyPI registry. For this thesis, we wanted

to test the implementation against the whole PyPI registry, but we could not get this ready due to time constraints. There was also sent an NSD application for downloading all of PyPI, which was granted and is mentioned later.

3.2.1 Fields & Weights

As mentioned in section 2.1.1 on Python, the language requires indentation to tell the compiler when code is moving in and out of scopes and functions. In other languages, this might be done using curly braces for example. Python also has support for semicolons to put multiple statements[1] on a single line, but in most Python code we have observed this is rarely used.

We will now explain how the fields and weights in our method work; If we look back to our graph (figure 3.4) showing the X-axis, we can think of the level of suspiciousness/risk as defined by $F(x) = y$, where x is defined as the *line of code (loc)*, and the output of the function y is defined as the *suspiciousness/risk*. The higher the y value the more we assume that line x has suspicious or risky code. When a line of code triggers one of our rules, a *bulletin* is created at the line-of-code. We defined the interval for x, or loc, as $1 \leq loc \leq lines$. This means x can only be as big as the file has lines of code.

The value of risk, or suspiciousness, that we added to the field as a curve or bump, is controlled by a function which we defined. The curve or bump we created has its median value at the location of the bulletin (at the line of code our rule triggered). We made this risk value a curve, so it would overlap with code around the trigger spot. The hypothesis was that malicious around it might trigger to. Several curved in a tight spot would amplify it, increasing the bump or curve. This again increases the level of risk at that line in the Python file.

To create a symmetrical adjustable curve defined by a formula, we used a probability density function (PDF). We choose a PDF that described a normal distribution, because we knew this could be symmetrical and would give us a good and adjustable area for potential overlapping of other curved and bumps. In figure 3.6 we can see such a normal distributions with different parameters for different shapes. The following equation describes such a distribution [41, p. 99]:

$$PDF_{norm}(x|\mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{\frac{1}{2}}} exp(-\frac{1}{2\sigma^2}(x-\mu)^2) \tag{3.1}$$

Using a PDF as shown in equation 3.1 we denoted the mean μ as the line number of the suspicious code. This gives us a curve at the center of the line that triggered, and gives us a symmetrical area on either side of the line. The variance σ^2 is a configurable variable to adjust how easily we want the curved or bumps to overlap in our file. For example, ff we have curves with high peaks, we need the bulletins to trigger close togheter for them to get amplified. For this thesis, the variance was configured as $\sigma^2 = 5$, because it seemed like a good middle ground between too little and too much peak in our curves.

[1] an expression such as x = 1, or calling a function

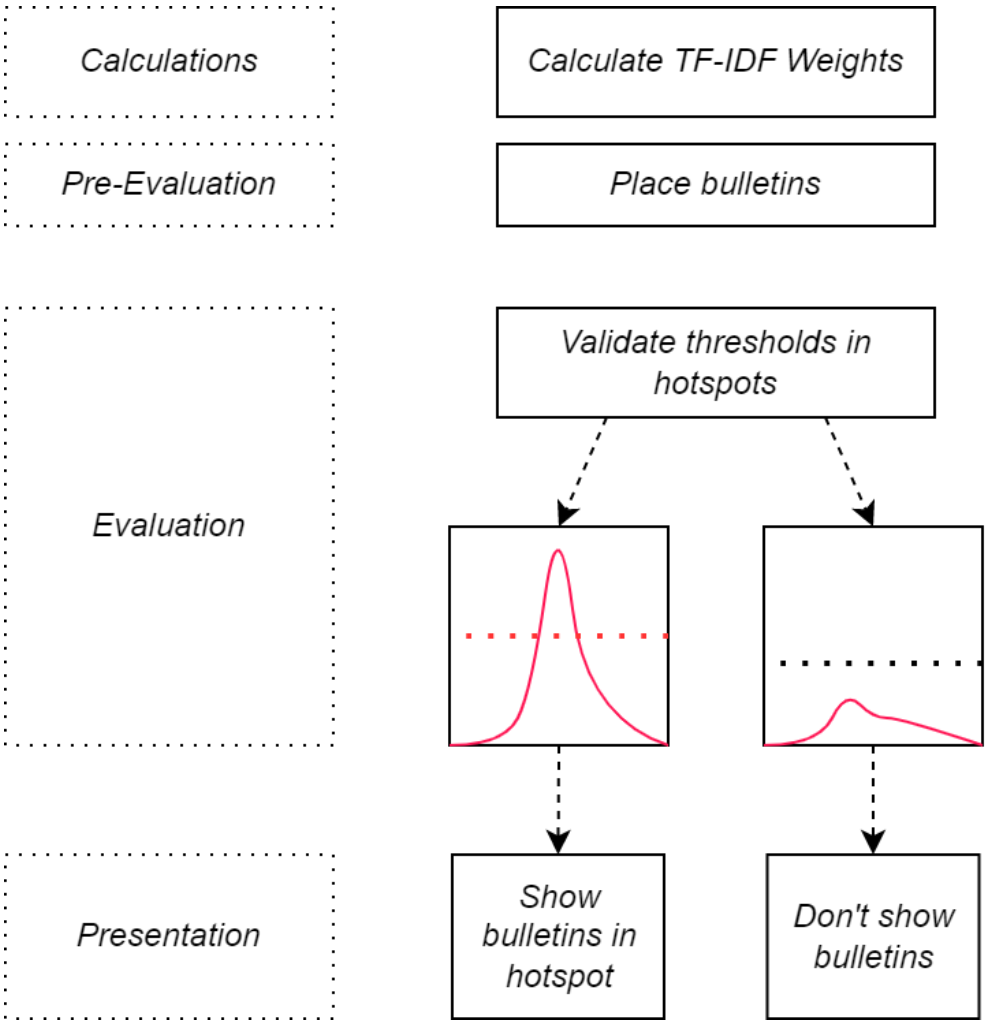

Figure 3.5: A graph showing the evaluation process after the bulletins have been created. Here we can observe that all bulletins are generated, and then later decided if they are shown or not based on the peak of the hotspots. Notice how the first two steps here are conceptually identical to the last two in figure 3.2.

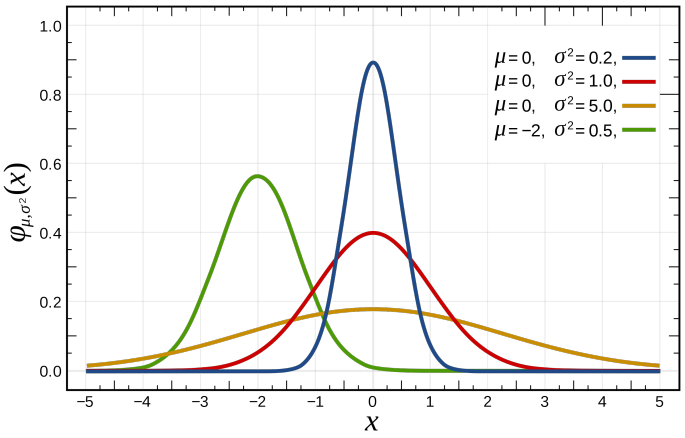

Figure 3.6: Normal Distribution Probability Density Functions (PDFs) - Source: Wikimedia Commons contributors [42]

The hypothesis is that code might get more suspicious the closer we get to the mean of the normal distribution. When a bulletin is created from a rule that triggered on a line of code, a curve is generated around this line (with parameters $\mu = linenumber$ and $\sigma^2 = 5$). The variance here is set to 5, as previously mentioned, but this could be changed. The different bulletins created the same curves, which were added to the field but could also be amplified by other factors such as TF-IDF values or multipliers from rules.

The idea of using a probability density functions was that we mimic waves. Two waves overlapping would increase the amplitude, which for us meant a higher peak if two bulletins are generated on the same line or very close togheter. The closer two bulletins are placed together, the higher the peak. For a single field we combined all the curves with the equation below. Where n is the line number. This question does not take into account *field weights*, yet.

$$F(x) = \sum_{n=0}^{N} PDF_n(x) \tag{3.2}$$

$$a(x) = F_{func}(x)w_1 + F_{im}(x)w_2 + F_{str}(x)w_3 + F_b(x)w_4 \tag{3.3}$$

We grouped the fields into four categories; (1) functions, (2) imports, (3) strings, and (4) behavior. After all the bulletins were placed on their respective fields, we combined the four categories into a *single field or final field*. This process used weights on each of the fields. This gave us the ability to control how much a certain field was included in the *final field*. As described in equation 3.3, we can tell the relationship between these four categories. The fields are represented as $f(x)$ with the subscript defining what category they belong to. Each field had their own weight. The single field or final field function is defined as $a(x)$. The weights are defined as any real number in the interval $[0, 2]$, but can also be outside of this. For example, if a rule detects a suspicious imported module, a bulletin is

placed at the line of the import, from which values from the PDF are calculated and added to the *imports* field. The imports field is later adjusted by the weight and added to the final field. These weights are configurable and play a critical role in our optimization strategy, which we go into later.

When all the functions and fields are combined we are left with a single function that tells us the level of suspiciousness or risk at any point in our given file. We use suspiciousness and risk interchangeably. Both are indicators that code is not benign. With this combined function, we can adjust the threshold on certain rules to be displayed or not, such that any peak below 0.5 would not be shown for example. This is set on a per-rule basis. The implementation also supports the configuration of a global threshold, but this is not used in the experiments shown in this thesis.

Figure 3.5 shows the general concept of the evaluation process. All the bulletins are placed, which then adds the PDF curve to the respective field. Do note that we also applied TF-IDF to the PDF curve as a multiplier before it was added to the field. As mentioned earlier, the fields are then combined into a single function using the user-specified or optimized weights. Further, we called the area under the peaks generated by the curves *hotspots*. A *hotspot* is then defined as an area of high suspiciousness, and as a way of clustering potential maliciousness. It is these hotspots that create the local threshold for the rules we have. If the hotspot has a peak that is high enough, then the bulletins generated by the rules in that location will be shown. If the peak was not high enough, the threshold for the generated rule will not be shown. Note that not all rules have the same threshold. This means a high enough peak for one rule to show in the same hotspot, which might not mean another rule considers the peak high enough. This is how we get the fuzzy behavior for how each of the rules is displayed.

The reason for choosing to work with fields in this manner is because we wanted to have human controllable thresholds for tailoring the detection to your specific needs. In the end, the program does the binary choice of displaying an alert or not, but the decision process is not binary and could be considered fuzzier. This decision process of combining fields and thresholds to guide the decision process with overlapping functions is in due part inspired by fuzzy logic. With human-readable weights, we are able to also easily observe what weights are being optimized. This gave us an opportunity to reflect on why the optimization valued functions over imports. We also experimented with weights as a way of eliminating false positives, so that we could potentially run our implementation on the PyPI registry.

In theory, we could in the future apply a neural network to this process, but for this thesis we explored the possibilities of using more primitive solutions first, because we were curious if it would work, and if so, how well. By using weights for each of our fields and optimizing these based on a dataset with malicious and benign code, we did to some degree create a very simplified one-layer neural network with four neurons. We did not use back-propagation for the optimization, though. Later we will discuss how we used genetic algorithms for that process. Do

note that this detection model was not supposed to be a classifier.

Hotspots

As we mentioned briefly in the previous section, we use the term *hotspots* for areas under the curve. The reason we chose this clustering technique is that it came naturally when implementing the fields, which allowed us to cluster functionality too. Another approach could have been an unsupervised clustering method on a one-dimensional axis (the lines of code where the bulletins are placed). We implemented hotspots in our model to test their feasibility. In future work we could compare it to more conventional clustering algorithms to see which methods works or performs better. We keep performance in the back of our minds when implementing this, as these hotspots are generated for every file in every project, but this is not a priority at this moment but is a consideration we made.

Figure 3.5 shows an example of creating a hotspot. This example only shows one, but imagine the two graphs were a part of the same graph. This would then generate two hotspots.

TF-IDF

The next section will outline how we experimented with *term frequency-inverse document frequency*(TD-IDF) [43, 44] in order to lower the number of false positives, and attempt to reject code that is not unique. We did this with the assumption that if a big library or framework frequently uses an import or module a lot across the whole project, then it is probably not malicious and should count less towards the decision to alert it. For this, we were inspired by Fang et al. [19], who used TF-IDF on opcodes (also mentioned in related work). We were also aware TF-IDF can be used for information retrieval in documents. We then hypothesized we could apply this to functions and imports instead of opcodes, and aimed to test the feasibility of this. The intention was to find functions and imports that are rarely used, with the hopes that malicious code uses functionality not commonly found in the library elsewhere. The equations 3.4, 3.5, and 3.6 show how we calculated TF-IDF for function calls. The same applied to imports.

$$TF = \frac{CountOf\,ThisCallInThisFile}{NumberOf\,CallsInThisFile} \tag{3.4}$$

$$IDF = log_e(\frac{NumberOf\,CallsInThisFile}{FilesWithThisCall}) \tag{3.5}$$

$$TFIDF = TF * IDF \tag{3.6}$$

In the actual implementation, we calculated the TF-IDF value for each unique function call and import done. We then stored this value and used a weight to

control its usage of this. We did this by interpolating between the calculated TF-IDF value and the actual value of the PDF-curve. The interpolation percentage was decided by the weight, which could be set in the implementation configuration. The interval of the weight was originally defined as the interval $[0, 1]$, but it could also be more or less than this. The weight were set to a default value of 1.0, but these weights were also optimized along with the field weights. We discuss more about this later in the chapter.

3.2.2 Rules

Since we wanted to look for specific functions, such as those working with networking and the operating system, we needed to specify which ones we were looking for. We also had to consider that some functions and imports are more interesting to us than others. Therefore we needed to implement a simple rule system where we could define both functions and imports, as well as set the thresholds for specific ones.

We then created a system where we defined either an import or a function call. These rules were then used to string match for an identifier after pre-processing the data from the abstract syntax tree (AST). Later in the chapter we provide further explanation of what this entails. Rules also incorporated a *functionality* aspect for future work on the implementation, with the possibility to identify a certain rule as a *file system* operation for example. This functionality aspect was not integrated into the detection model due to time constraints, but could be in the future. Below we show how rules were defined.

```
Module(Functionality, Identifier, Name (optional), Description (optional))
Function(Functionality, Identifier, Name (optional), Description (optional))
```

Code listing 3.1: The structure of a rule

Using this as a template we could then define a group of rules and set the threshold for the whole group. We also gave it a name so we could recognize it more easily. The rule format also supported comments. The file format was implemented using Rust Object Notation (RON) [45], which is a variation of the common JSON format but is specific to the Rust programming language. RON is easily converted into Rust native data structures, which was what we used in our implementation. Below is an example of two groups that define rules with different thresholds:

```
( name: "Basic module rules", threshold: 0.30, rules: [
    Module(Process, "subprocess", None, None),
( name: "Basic function rules", threshold: 0.20, rules: [
    Function(System, "exec", None, None),
])
```

Code listing 3.2: An excerpt of two rules we defined for our default rule set.

The thresholds in the above excerpt are 0.30 and 0.20, and were set from experience with the fields. In general, a lower threshold means the rules will trigger

more often. We created a default set of rules which were included in the implementation, yet these are very customizable and can be improved in the future. The default rule set is largely based on a list from Fang et al. [19] which they used in their classifier. A full list of the rules used in the default configuration is included in the appendix B.

3.2.3 Pre-processing of the Abstract Syntax Tree

Since we did not have an interpreter or compiler to perform all the pre-processing that would normally done to the AST, we did it ourselves. One of the first things we needed to implement was import and function aliasing. This is a Python language feature that allows us to shorten an imports name for more convenient use when coding. Doing this pre-processing in our implementation is not complicated but did require some work. It is mostly done through hashmap and dictionary data structures.

One reason we used the AST instead of compiling code was because it was quicker. We deemd it was worth it to use simple time-saving steps in the pre-processing stage because we wanted to perform a large amount of analysis.

Import aliasing

Here is an example of the import aliasing we mentioned earlier, using the popular package *numpy*:

```
import numpy as np
a = np.array([1, 2, 3])
```

Code listing 3.3: An example of numpy code using aliasing.

A rule matching for *numpy.array* would simply fail without pre-processing. A rule for *np.array* in this scenario would work, but we had to take into account that not everyone uses the same alias. Therefore we needed to support the resolving of aliases, as this is a very commonly used language feature. The above example would look like this with our implementation after pre-processing:

```
import numpy
a = numpy.array([1, 2, 3])
```

Code listing 3.4: Example of numpy code without alasing.

Now our rule for *numpy.array* would trigger a bulletin in this location.

Function aliasing

The next example shows the same concept but with function aliasing. While similar, this is used for importing single functions from a library:

```
from numpy import array as arr
a = arr([1, 2, 3])
```

Code listing 3.5: An example of function aliasing

The above example would look like this with our implementation after pre-processing:

```
import numpy
a = numpy.array([1, 2, 3])
```

Which is similar to what has been shown earlier.

3.2.4 The Approach to Obfuscation

The next process after this was tackling matching on strings and variables of interest. This could be done through a simple text search on the text representation of the Python file, but in some cases, the malware uses obfuscation to try to subvert static analysis. We also did not want to use Regular Expressions, which are common for searching text, due to concerns of performance at the scale we wanted to deploy our software. In our implementation, we used two simple techniques to try and defeat some obfuscation; partial evaluation, and text-encoding rainbow tables, which we call canaries.

Partial Evaluation

Partial evaluation is a process of picking specific parts of a program and then evaluating only that part as if the program was running. This is useful if you want to see the result of a string operation that might be obfuscating some data. If we continued implementing partial evaluation, until we implemented all the features in the language, we would eventually have made a compiler or full interpreter, which was not our intention. We used this process to aid in the static analysis and to give us more information to work with. This process can be useful for simple operations such as string concatenation, which some malware use to (barely) obfuscate strings. Other operations we observed in malware are indexing, slicing, reversing, and replacing strings.

Consider the following example of Python code that prints a simple string:

```
print("hello world")
```

Code listing 3.6: A simple *Hello World* example.

When this example is turned into an AST from text representation it is a rudimentary task to extract the variable and use it in our implementation. We simply looked up all the *Constant*-nodes and extracted their value. A simple text search would also match if we search in our file. In figure 3.7 we can see the AST generated for the code example above. The string is stored as a constant in the AST and is easy to extract.

If we then introduce the previous example but with simple string concatenation, the AST is transformed into a bigger tree:

```
print("hello " + "wo" + "rld")
```

Code listing 3.7: A simple Python program concatenating a string.

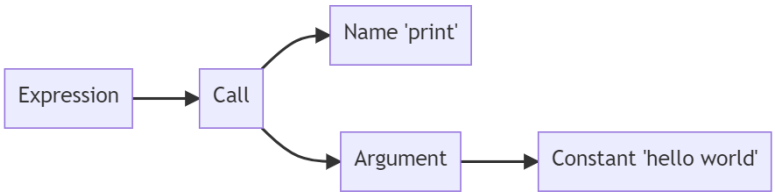

Figure 3.7: An example showing 'print("hello world")' when converted into an AST for the Python interpreter.

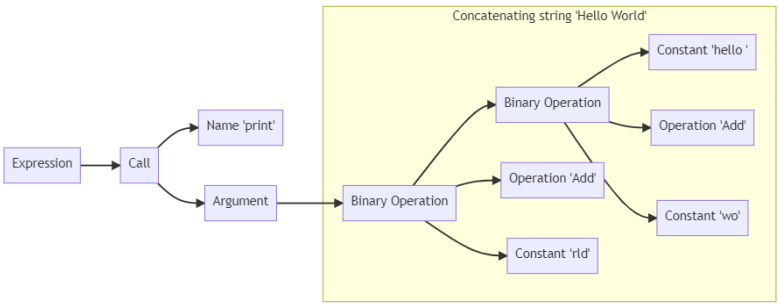

Figure 3.8: An example showing how simple string concatenation is converted into an AST. This example is based on the Python code 'print("hello " + "wo" + "rld")'

Figure 3.8 shows the above code example converted to an AST representation. The area highlighted in yellow is responsible for the concatenation of the three separate constants. In our implementation we implemented simple operations like these by performing binary operations on strings. If the constant was a number or other type than a string, the operation was aborted and the value left as is. In this example, we can no longer just look through the *Constant*-nodes. We also had to fetch them in the right order by reading the *Binary Operation*-nodes.

Canaries

Canaries[2] is the term we use in this thesis for text-encoding rainbow tables. Since different types of encoding, specifically Base64, is very easy to use in Python it is used quite a lot as the encoding for sending more malicious Python for the program to execute. Our hypothesis for why this is used so frequently is that sending a whole file as a single line of Base64 is quite robust, hides the actual payload from plain sight, and is easily decoded. This is a method we have seen being used by the information security community.

[2]Inspired by how they were used detect to carbon monoxide when coal miners were underground.

As mentioned we wanted to detect if a program has Base64-encoded code, which would be indeed very suspicious from the examples we have seen. The approach we took used the fact that Base64 is an encoding and not an encryption algorithm. The idea was to have a file of pre-computed strings to match against constants found in a Python file. If the pre-computed string matched, it was supposed to signal we found a program, URL, or other object encoded using Base64. The canaries are agnostic to how they were used to search a file or code, but we paired this with our partial evaluation and search *Constant*-nodes instead of full-text search.

We based our implementation on the fact that the *Base*-family of encodings (Bse64, Base32, Base16) are text-encoding algorithms. The same input will always generate the same output. We then created pre-encoded strings using a list of common words that start a payload. These common words could for example be *import*, or *https://* if it's an encoded URL or domain we want to catch. The full list of common words we used for our canaries in the implementation of our method can be found in appendix C.

Before we calculated the canary we had to be aware of the padding for the encoding, since they work in multiples. If an input string is less than a certain multiple, the encoding pads the tail end of the string so it becomes a multiple. This is important because we matched on sub-strings, which were a part of other bigger encoded strings. We, therefore, needed a cut-off point, which meant up to that cut-off, the two encoded strings had to be identical. Base64 for example pads the end of a string so that it is a multiple of four. We did not have to to know this for us to create the canaries, but it is good to be aware of. Here is an example to illustrate how we can ignore this padding length; The term *import* when converted to Base64 would be *aW1wb3J0*. If we used this in the example below, we would indeed find a match, because of its length it generates an encoded string with a length that is a multiple of four:

```
import            -> [aW1wb3J0]
import subprocess -> [aW1wb3J0]IHN1YnByb2Nlc3M=

Exact match!
```

Code listing 3.8: An example of the word *import* being used as a canary to match on a Base64-encoded string.

Consider the text *http*, which converts to *aHR0cA==*. If we now want to search for in our example above, we can see we do not get an exact match, which is what we want. We could of course configure some form of partial matching, but for performance reasons and doing these thousands of times, we wanted exact matching.

```
http                -> [aHR0cA==]
https://www.ntnu.no/ -> [aHR0c]HM6Ly93d3cubnRudS5uby8=

No exact match.
```

Code listing 3.9: An example of a canary not matching due to padding in the encoding.

Our wanted outcome was to have the least common full match at the beginning of any encoded string starting with *http*. To do this we calculated two strings, where one was padded with characters (in this case with A's). We then took the common part of the beginning of the two strings and created a cut-off point. This common string is what we save in our file.

```
http         -> aHR0c|A==
httpAAAAAAAA -> aHR0c|HM6Ly93d3cubnRudS5uby8=
                     ^
                     | Cutoff point
```

Code listing 3.10: An example showing how we decide the cut-off point for canaries.

We then stored the result in a file where we could reverse lookup the encoded string *aHR0c* to the common word *http*. We did this with many different variations of encoding, made a list of words we thought are common, and created a script to generate this list for us. As previously mentioned, the implementation used a default list of common words to generate canaries with can be found in appendix C. We do not include the full list of generated canaries in this thesis as it would be too large to include. However, the Python-script used to generate these are open-source and available for anyone to use [46].

3.2.5 Experiments

We wanted to measure the performance of our techniques by performing a set of experiments. These were designed to give us an idea of how well the implementation would do when scanning a huge set of packages. In the section on datasets later in the chapter, we will discuss what criteria we had for this.

First off we performed a couple of control experiments to create a baseline for our performance. We wanted to know the performance of our method when all the weights were default values (1.0). Another control experiment wanted to perform were to investigate the maximum detection rate. This can be done by optimizing the weights by only using F-score as a fitness value.

Since the experiments were designed to measure how scalable our implementation is, we wanted to perform three core experiments, with some variation depending on what we saw the need for.

The two most rudimentary experiments we wanted to perform were one set with TF-IDF enabled, and one with it disabled. This was so we could compare the differences, and see if we could see a noticeable difference. TF-IDF was used to prevent false positives, so we wanted to see how this worked on pure function calls and not opcodes, such as Fang et al. did [19]. We also wanted to incorporate a train-test splitting on the dataset to prevent overfitting. We therefore performed experiments with such a split, and ones without. This was to see if we could spot any difference. If we trained and tested the weight sets on the same dataset, we were afraid we might overfit the weights to the dataset. This is why we performed

experiments with and without training and testing datasets, so we could compare and see if this had any noticeable effect.

In addition to testing out enabling and disabling TF-IDF, we also wanted to test a variation on the IDF function. This is because we know TF-IDF is primarily used for information retrieval in documents, and was not made for lowering false positive rates when looking at suspicious function calls and imports in Python code. We then wanted to experiment and see if changing the IDF function could give us good results.

The initial *inverse document frequency (IDF)*-function that we used was the most common we had seen, and in short we calculated this using the equation below (this is the same equation 3.5 presented previously):

$$IDF = log_e(\frac{NumberOfCallsInThisFile}{FilesWithThisCall}) \tag{3.7}$$

We had a hypothesis that this function penalizes results in smaller packages with only one or a few files. Since the idea of TF-IDF is to help lower the false positive rate in huge projects with hundreds or thousands of files. To mitigate this we experimented by removing the inverse part of the equation, effectively turning it into a term frequency function instead. The new equation included the aspect of how many other files include a given function call. The idea was; the more a function was called in other files in a project, the lower the calculated TF-IDF value would be. The new equation, which we call *document frequency (DF)*, is shown in equation 3.8:

$$DF = \frac{FilesWithThisCall}{NumberOfCallsInThisFile} \tag{3.8}$$

$$TFDF = TF * DF \tag{3.9}$$

We then calculated TF-DF using equation 3.9. We experimented with this new function in chapter 4.

Next, the parameter optimization step is discussed, which was a critical part of our experiments. After the parameter optimization section we show then elaborate on how the fitness function was designed and used to perform this optimization.

Parameter Optimization

Through the thesis we also wanted to optimize the weights our implementation uses by applying machine learning. In table 3.1 we see the different weights we optimized. The weights were of course agnostic to the machine learning algorithm used to optimize the weights, and any algorithm that is suitable should in theory, work. There could of course be differences in the optimization they generate. This depends on the experimental setup, and on how much feedback the machine learning algorithm needs. Our experiments took quite some time to finish, which

#	Weight	Default value
1	Field Weight Functions	1.0
2	Field Weight Imports	1.0
3	Field Weight Behavior	1.0
4	Field Weight Strings	1.0
5	TF-IDF Weight Functions	1.0
6	TF-IDF Weight Imports	1.0

Table 3.1: A table showing the different weights we can optimize. We split them into two parts; the ones for the fields and the ones for the TF-IDF weighting.

meant we wanted an algorithm that did not need to run analyses on our dataset an unnecessary amount of times.

Fitness Function

Our fitness function can be seen in equation 3.10. This is the function we used in PyGAD to optimize the weights. The function was designed to be maximized, meaning we wanted the highest possible fitness value. In theory, we get a high value with a low amount of bulletins and a high F-score. We added a very small constant to the bulletin count in case it becomes zero. We wanted the F-measure to be as high as possible (close to 1.0) and the bulletin count to be as low as possible (close to zero). We multiplied by 10000 in order to get a more tangible fitness value to work with, as working with too many decimal places proved to be less robust. This may have been due to floating point precision errors, but we also wanted a number that was human-readable where possible.

$$Fitness = 10000 \frac{FMeasure}{BulletinCount + 0.00000001} \tag{3.10}$$

3.3 Dataset

The dataset for this thesis was a critical component where much time was spent, and was one of the initial roadblocks for the project. Initially, we planned to divide the data we gathered into four different categories: (1) malicious code found openly online, (2) datasets found publicly or gathered from other researchers, (3) custom made malicious samples, and last (4) benign code used as a control. It was not clear at the beginning of the project if there was going to be machine learning involved, but we knew we needed malicious samples, which were the hardest to acquire.

3.3.1 Malicious packages

Due to the fact that the field of supply chain attacks and malicious packages in open source registries is very recent, there is no fully publicly available dataset. This is probably due to the ethical reasons of accessibly sharing malware, which can be used as derivative malware in the same registries. We were preferably on the lookout for a dataset containing actual malicious packages from PyPI. These are different from normal malicious Python code, which could a backdoor for example, since packages published on PyPI have a specific structure that all packages must follow. This would make the dataset more accurate. Our plan was then to attempt to acquire a dataset from the literature we have been reading. In chapter 4 under experiments, we will describe how we acquired such a dataset.

3.3.2 Benign packages

For the benign packages, we decided to go for a collection of the most downloaded packages from PyPI. The idea is that these are generally safe to use, and we make the assumption these are not malicious packages (they can of course include security vulnerabilities). The reasoning behind this is that the packages are pretty widely used and any malicious activity would hopefully be detected quickly by users. In chapter 4 we describe how we downloaded the such packages.

3.4 Metrics

In this section, we discuss the different metrics we used to measure the performance of our implementation. As we mentioned at the beginning of this chapter, we created bulletins based on certain features such as rules of anomalies. These were then evaluated by the hotspot, which decided if the bulletins are suspicious enough to be shown to the user (here is also a mode in our implementation to override this mechanism, which will all alerts if enabled). This means bulletins are the measure we wanted to keep track of, and we wanted to see how well we placed bulletins on the specific lines of code we associate with malicious behavior in our dataset. We wanted to have a low number of bulletins when we looked at benign code, and a high number of bulletins when we looked at malicious code. Since the model allows for configurable weights, we can experiment with adjusting it to be more or less sensitive about features. We did not do this by hand, though. We optimized the weights using machine learning.

Since our bulletins were created for a specific line of code, we knew what lines of code specifically were malicious in our dataset. Only then could we be certain that the implementation generates the expected bulletins in the correct location and when it mistakes benign code for being suspicious. This labeling of the dataset is shown in the experiment section of chapter 4.

We kept track of how well bulletins were created using simple metrics that are commonly in confusion matrices. From the simple measures used in the confusion

Classification	**Description**
True Positive (TP)	Any bulletins that trigger inside a line of code (LOC)
False Positive (FP)	Any bulletins that trigger but are not inside any LOC
True Negative (TN)	Every line that correctly *does not* get a bulletin
False Negative (FN)	Any LOCs that does not have any bulletins

Table 3.2: The different classifications we use and what they mean in our context.

matrix, we calculated accuracy, recall, precision, and an F-measure [47, p. 68].

In table 3.2 we explain how each of the classifications corresponds to our scenarios. In practice we did not use the false negative (FN) classification, as this would just be any line that did not get a bulletin made for it.

We also tracked a time variable for our experiments, but these were in the scale of hours. This meant we did not give much consideration to accurately timing the speed of the operations since these small changes would not matter at our current scale. We were more interested in just comparing the experiment times to see if one was faster than the other.

3.5 NSD Application

For the thesis, we also submitted an NSD application to get approval to create a mirror of PyPI for research purposes. The aspiration at the beginning of the project was to test the implementation against the PyPI registry if it proved feasible. The application was later approved in case we got the time to do so.

The submission of the application was a discussion point, due to the fact that anyone can mirror the PyPI registry with open and publicly available software. The intention of the mirror would be to analyze the code, but in the process, we would also get information about the author as collateral information. We wished to not collect due to privacy of the authors, and because of GDPR rules here in the EU. The technical aspect of removing author information from all the packages from PyPI would be unfeasible and decided in the end to just not use it, but the argument for not removing it is that users submit packages to this registry willingly, and also submit their information knowing it will be public. There could of course be cases where users do not know this information (name and e-mail for the most part) will be public.

Chapter 4

Experimental Design and Results

Now that we have presented our methodology, we will elaborate on the experimental setup, the implementation, and the results. The experiments intend to determine if it is feasible to run our implementation on the entire PyPI registry and how well our detection model works.

For the experimental setup read on to the next section. If you want to go straight to the implementation jump forward to section 4.2, or to go directly to the results from the experiments jump to section 4.3.

4.1 Experiment Setup

Our fitness function discussed in chapter 3 has the need for two metrics we are interested in tracking; bulletin count of benign packages, and an F-score for all the malicious packages. We run two tests separately to acquire these, one test on the benign dataset, and one on the malicious dataset. These two tests were run separately, so the dataset was not mixed. The way we did this was by creating a small test framework in Python in which we could pragmatically define our experiments. We could now automatically execute experiments, which would be used by the genetic algorithm to determine the best samples in the population. We set up the genetic algorithm to use the weights described in the table 3.1 as genes.

A note on the performance of a single *run*. A single run is what we consider to be one pass over all the benign packages, and one pass over all the malicious packages. A single run acquires the fitness by combining the results from these two, and this could take several minutes depending on the features used. And a full experiment could take upwards of six hours to complete a full set of 10 generations with the current configuration. The TF-IDF calculation significantly increases the time to analyze all the packages, which is why it would be beneficial to partially disable it, or not at all. This is also why we want to compare TF-IDF runs and non-TF-IDF runs.

The genetic algorithm is set to run for 10 generations, but reports back the

best solution and other metrics at generations 1, 5, and 10. This means the results from the generations are all from the same starting pool, only more evolved.

The malicious packages were run with our program in a virtual machine (VM) in case the malicious packages somehow got activated in the process of testing. The implementation should not pose any risk to the host machine when analyzing malicious code, but we were concerned our test framework would accidentally execute malicious Python. Windows Defender was also very persistent in deleting parts of our dataset, and we did not want to create an exclusion rule for it. It is possible the anti-virus had not reacted if it was still compressed (they are downloaded in compressed form).

Through the use of a shared folder, we transferred results and code back and forth between the virtual machine. We compiled the implementation for Linux in the virtual machine without any modification. This proved that the program works for both Windows and Linux operating systems. It has also been tested on macOS. The test framework, which we mentioned earlier, would communicate with VirtualBox to run a specific script that starts the experiments and writes the results to a shared folder. This is similar to what Sand did in their thesis, where they had scripts on the host machine run commands in their virtual machine [36].

The following code is what we used to control scripts inside the virtual machine:

```
def run_in_virtualbox(args):
  out = subprocess.check_output(["VBoxManage.exe",
    "--nologo", "guestcontrol", "kali-linux-2022.1-virtualbox-amd64",
    "run",
    "--username", "kali", "--password", "**********", "--wait-stdout",
    "--", *args]).decode("utf-8")
  return out

# Execute ‘run.sh‘ inside the VM and capture the terminal output.
output = run_in_virtualbox(["/usr/bin/zsh", "/home/kali/run.sh"])
```

Code listing 4.1: A snippet of code used to control our experiments inside the virtual machine from our host machine.

The process is fully automatic, and as previously mentioned, the experiments were then pre-programmed and left to run unattended to calculate the results. This could take many hours and be mostly done overnight.

The experiments were set up on a personal computer running Windows 10 with an AMD Ryzen 5 3600X 6-Core processor and 32GB of RAM. We did not utilize a GPU or other hardware acceleration in these experiments. In practice, the implementation did not use much RAM at all but could utilize a lot of CPU at certain times during the experiments. A limitation in the implementation is that it runs entirely on the CPU, instead of running on the GPU, which would have been much faster.

4.1.1 Genetic Algorithms for Optimization

For the parameter optimization we chose to utilize genetic algorithms [47, p. 146].

As previously mentioned in the method chapter, we wanted to optimize six weights that are used in our implementation. These weights are used for combining fields and adjusting the influence of our TF-IDF values. By default, all these weights are set to 1.0 and we deem the interval $[0, 2]$ of as the most reasonable.

We did not implement our own genetic algorithm, instead, we used a publicly available library called *PyGAD* [48]. This library had extensive options we could experiment with, but we decided to use the default configuration. The mutation step is set to select 10% of the population randomly. Only one point is used in the crossover step. A method called steady-state selection was chosen for picking parents, and only one parent was kept. The number of genes was set to six, as this is the number of weights we wanted to optimize for. The full configuration can be found in appendix D.

4.1.2 Acquiring the Datasets

This section will go into in detail on how we acquired the datasets we used, how we prepared them, and then lastly, how we used them.

We first searched for malicious packages online (category 1 dataset, see chapter 3.3), yet this proved not to be very beneficial. There are lists of known previous malicious packages that have been present on PyPI. Ohm et al. curated a list of such packages [40]. News outlets also sometimes report on the findings of these. Most of these known malicious packages are taken off the registry though, as they should. We did have some success in finding a PyPI mirror with some malicious packages, but downloading them revealed another problem because the malware would be run on installation even with the *download* flag enabled. Ohm et al. also note this and recommend using *Wheels*, which is another packaging method for Python where no code is executed upon installation. As a word of warning, we therefore strongly suggest using a virtual machine if you want to attempt to download known malicious packages from PyPI or a mirror of it.

We tried to gather individual examples of malicious Python packages, but most were proof-of-concept code found on GitHub. The actual malicious packages from PyPI that have been reported over the years have, as we mentioned, been taken off the official registry after being reported. If you manage to find a mirror that does not delete packages and only synchronizes with new ones, you could get "lucky" and be able to download the previously detected malicious packages.

We contacted Ohm et al. [40] by email, who as previously mentioned curated a list of real malicious packages found in several package managers, including PyPI. They thankfully gave us access to their dataset, which we will elaborate more in the next section. This covered our second category, which was locating a dataset. We then quickly discarded the idea of finding malicious code ourselves, as the dataset we just received was just what we were looking for. We also discarded the idea of creating an abundance of custom-made malware, as we did not have the resources for this and in the end, would only be derivatives of known malware. We will also go into the last category of benign code, which thankfully was more

straightforward.

Malicious Dataset

For our dataset, we directly contacted Ohm et al. as they could upon request grant permission for research purposes. Their dataset is called *Backstabbers Knife Collection* and is described in depth in their paper [40]. Their collection includes malicious packages from four large open-source registries; NPM, RubyGems, Maven, and PyPI. We have personally inspected all the samples from PyPI in the dataset, and regard them all as malicious. The dataset has grown over time, meaning it has more packages now than when their paper was released. We, therefore, note that not all malicious package from PyPI in this thesis is included in the results of their paper.

Across RubyGems, NPM and PyPI, Ohm et al. found that malicious packages were available for 209 days on average before being made unavailable. They make due note of the package managers running arbitrary code on installation of a package as a reason for the increasing number of malware, and that 56% of packages they analyzed initiate their routines on installation.

The dataset at the time of our download included 97 malicious packages from PyPI. The packages were extracted to be ready for analysis and usage. This dataset is what we used for our second (2) category, which was datasets found online or given by other researchers. The labeling of the dataset required us to mark every line that was deemed malicious, we could then see that the average malicious package had 56.42 lines of code associated with its malicious functionality. The median value was 20, and the standard deviation was 163.66. We go into detail on how we labeled the dataset in a later section.

Benign Dataset

We decided to download the top 360 most downloaded packages at the beginning of February 2022. In practice, the download of this includes dependencies of those packages as well, which made our total number of benign packages 442. The list of these packages was acquired via Google BigQuery, as PyPI uploads statistics here. A tool called *pypinfo* [49] integrates with BigQuery, which allows us to perform queries (for a cost).

Pre-processing of the Dataset

Before we could use the data we had gathered, we had to pre-process it. In this section we will discuss how we prepared the compressed data, how we labeled the dataset, and how we decided which released version of the malicious packages to use.

Both the malicious and benign packages were compressed when downloaded. Packages are by default compressed when they are distributed by the package manager. This is either in a Zip or Tar format, which are both widely used. We

started by decompressing the packages to make it easier to browse through them, but also to lower the overhead when doing analysis. This meant we did not have to decompress the packages in memory for each analysis. This does take up more space, but it really is not a problem if you do not try to scan the whole PyPI registry. This means the current implementation does not incorporate decompressing of packages in memory, but this could be implemented in the future.

As mentioned in the previous section on metrics, we need to be able to tell what lines of code in the sample are malicious to be able to tell how well the implementation did in our experiments. We achieved this by labeling the dataset of malicious packages we got from Ohm et al. This proved difficult at times. Some samples only had a few lines of code that were malicious, while others were an entire file with a full implementation of a backdoor or key-logger. In the latter example, we choose to label the entire file as malicious, since circling in on one part was not feasible. This is a limitation of this approach. We choose to also label supporting code, such as import statements for modules used in the malicious code. We choose not to label imports that were a part of the original package. The following listing is an example of how this was labeled:

```
malicious-package,/setup.py:4-9!23-43;/folder/test.py:1-3!19-19
```

Code listing 4.2: An example of how malicious lines of code are labeled for a single packages.

Since a package could have several malicious files, we split them by a semi-colon. The lines of code were written as inclusive, which means 4 to 9 means inclusive 4 to inclusive 9. We used an exclamation mark to separate code ranges for a single file, and files were defined from the root of their package. For example, the */setup.py*-file is located at the top level of the package.

In the dataset, we also get all published versions of a package. For this pre-processing, we decided to only use the latest version of published malicious packages. This was because a lot of the previous revisions used identical code. Since each package had different numbers of published versions, it was decided to only use the latest to prevent potential over-fitting, since a lot of the versions would have almost the same code. We do note that there were also packages that were alike in their malicious payload, which could also enable some over-fitting.

We also choose to label dependencies in files such as the *setup.py*-files that we knew were malicious. This is because we wanted to be able to show bulletins created in dependencies. This could also be integrated with databases that contain known malicious package names and versions. An example of this is Safety [23], which we presented in chapter 2. Technically the implementation as it stands does support detecting dependencies of a package but lacks the integration to forward bulletins for that dependency (e.g. lookup in a database, or through another API). We elaborate on this in future work.

Splitting the Dataset for Testing and Training

To not over-fit the weights in the machine learning process, we also incorporated a training and testing split of our dataset. There are some considerations we had to do when doing this. A single *run*, which will be defined at the start of the next section, can take several minutes, and a full experiment usually takes upwards of six hours. We, therefore, did not use a more thorough method of say K-Fold splitting, where we would use a $K = 5$ for example. This would be too resource-intensive for us at this point. We would recommend optimizing the experiments and then trying out K-Fold for optimization in the future. This will be discussed more later in the discussion on *Experiment Speed* (section 5.3).

We, therefore, use a standard 80/20 split on the dataset. This means that when we employ this, we split the samples in the benign and malicious datasets to 80% for training, and 20% for testing. We will refer to this as an 80/20 split in the rest of the thesis.

4.2 Implementation

For this thesis, we implemented a program in the Rust programming language that utilizes the techniques we presented in our methodology chapter. This implementation is free and open-source on GitHub [1], as was a goal from the beginning of this thesis. In total it harbors around 2900 lines of code and is implemented as a cross-platform command-line tool. It can be compiled and run on Windows, Linux, and macOS.

The implementation has been programmed with the intention of it being extensible and future-proofed where we could. We, therefore, hope this implementation can be worked on in the future.

Choosing a Programming Language

From the start, the goal of the project was to make a static analyzer as a prototype, but also make it extensible and robust. The robustness was deemed important to ensure the correctness of the program. A first prototype was written in Python to test out some of the concepts, but this quickly showed that we needed a strongly-typed language. The structure of the program, and analyzing many packages as quickly as possible also meant multi-threading the implementation would perhaps yield a big impact. We also wanted the implementation to be as easy to use and run as possible, which is why it comes with a default configuration (the one tested later in this chapter). To make sure we could compile the implementation to any architecture and operating system we choose to go with a programming language that would support this.

The prototype implementation was therefore written in the Rust programming language. A strongly typed programming language that supports many architectures. We could also utilize a lot of the functional data-oriented principles

that Rust allows us to use. This allows for maintainable and understandable code, which would make contributions by others easier. Rust also has good support for parallelization in both features and libraries available.

This was the first big project we attempted in Rust, which did prove a challenge. Learning the language took time, but was also included in the planning of the thesis. In the end, we were happy with the result and the language.

RustPython for AST-parsing

We needed a library for parsing the Python AST from the text representation. This is due to the development time it would require to build such functionality for this thesis only. The RustPython project[50] was discovered at the beginning of the thesis and provides an open-source library for parsing and returning a full AST. This fitted well into our project since we wanted to integrate this functionality directly into our implementation. The library *rustpython-parser* has now become essential for our implementation. We do note that the RustPython project is still in development and is not yet production-ready, but for our implementation, it does exactly what we want.

AST-walker

The Python standard library [1] provides by default an *ast*-module [51]. We experimented with prototypes of our implementation using this module in the early stages of the thesis. This module provides functionality to visit and traverse specific nodes in the AST, which is useful to, for example, only visit function call nodes to collect their names. The downside was the RustPython-library did not have such functionality by default, and since we chose to use this library for the full implementation we had to implement an AST traverser/walker ourselves. This module we developed is provided in the open-source repository for the implementation [1] under the name *ast_walker*. The design goal was to mimic the Python module as closely as possible but in Rust.

[1] A set of libraries included by default with the programming language.

4.3 Results

In this section, we will explain the experiments we performed and the results we got. We try to determine the feasibility to use the implementation for the two operating modes we have in mind. Again, these modes are; (1) running the implementation on a single package and its dependencies, and (2) being able to scan huge amounts of packages to find the needle-in-a-haystack, which would be a malicious package. We will only perform experiments for the second mode, as we will try to infer the performance on a smaller amount of dependencies from these results.

Before we show the experiments, we will demonstrate the usage and application of the implementation on a single file. This is to demonstrate its features, and how the feedback to the user looks like.

4.3.1 The Impact of TF-IDF

During development we discovered how TF-IDF completely hid important bulletins in sample packages we were using as control. One of these packages were the *jeilyfish*-package (a part of the dataset we described earlier). With default weights set to 1.0 and TF-IDF enabled, this package got zero bulletins. If we did a single pass of this package again without TF-IDF, we received 12 bulletins.

4.3.2 Experimentation on a Single File

To demonstrate the abilities and limitations of the implementation, we present a program with different suspicious behaviors. For this test, we turned off the TF-IDF weighting for function calls and imports. This is because TF-IDF generally only works if we have a lot of files, but this will be discussed more later in the next section. If we use TF-IDF on such a small program, we would see a lot less bulletins (as presented in the above section). This is a limitation of enabling TF-IDF for samller files, but it is intended to lower false positives in larger dependencies.

Below we can see the code for the suspicious sample. Do note that this is for demonstration purposes, and that the program does not actually run. Except for the *test*-function which would import the *builtins* module in Python, and dynamically get the built in method *exec* from it using the *getattr* function. The function name is encoded using Base64. It would then decode a Base64 snippet of code that gets the hostname of the local machine and returns it.

```
import os
import socket
import os.path as awdawd
from importlib import import_module as im
from Crypto.Cipher import AES

def x():
  return "vY2tldDtwcmludCh"

def test():
```

```
  key = "aW1wb3J0IHN" + "zb2NrZXQuZ2V0" + "aG9zdG5hbWUoKSk="
  hello(key)

def hello(key_param):
  import base64
  import marshal
  m = im("buil" + "ti" + "ns")
  s = base64.b64decode(key_param).decode("utf-8")
  e = getattr(m, base64.b64decode("ZXhlYw"+"==").decode("utf-8"))
  e(s)

f = open("myfile.txt", "x")

exec("hello there")

test()
```

Code listing 4.3: A sample malicious file for testing.

The first most basic demonstration is detecting the use of *exec* in our program, which is used a lot in malicious Python code. Below you can see the output from the bulletin that was created:

```
 24| exec("hello there")
      ^The function 'exec' is often used in malicious activity
```

The implementation also marks imports that are often used in malicious activity, because we have defined it in the rule-set. The packages beneath are also used for totally benign purposes, which also illustrates that judging what is suspicious or not is difficult in static analysis.

```
1| import os
  ^The import 'os' is often used in malicious activity
2| import socket
  ^The import 'socket' is often used in malicious activity
```

In order to demonstrate the functionality for resolving aliases and partial evaluation, we show the results from a part of our program trying to import the *builtins* package, which can be used to dynamically get functions such as "exec", which we are indeed trying to do.

We see the usage of "im" as an alias to "import_module" is detected, and the partial string evaluation shows us that this package is the "builtins" package:

```
4| from importlib import import_module as im
  ^The import 'importlib' is often used in malicious activity

[...]

17| m = im("buil" + "ti" + "ns")
        ^Functionality was dynamically imported (at runtime). [...]
        ^The import 'builtins' is often used in malicious activity
```

The last example from this we want to show is the use of canaries. Below you can see the output from our implementation detecting the use of "import" as a Base64-encoded string. This could in future work be chained with string evaluation to automatically decode the string and to then recursively run analysis on that code again for example.

```
10| def test():
11|   key = "aW1wb3J0IHN" + "zb2NrZXQuZ2V0" + "aG9zdG5hbWUoKSk="
        ^Canary triggered: detected 'import' using transform 't_b64encode'
12|   hello(key)
```

To demonstrate the limitations of the canary detection using the AST, we present another line of code from our example. This line of code decodes "exec" from a Base64-encoded string, but no bulletin was created. We know this should have generated a bulletin because "exec" is in our canary list. This is probably due to arguments in functions not being recursively handled, which could be extended in the future, but is nevertheless important to demonstrate. This example in particular could be found using a simple regex search, but in some cases that might not be possible. Combining this with regular expression could be possible in the future.

```
19|   e = getattr(m, base64.b64decode("ZXhlYw"+"==").decode("utf-8"))
        ^The function 'base64.b64decode' is often used in malicious activity
```

4.3.3 Experiments on the Dataset

Since the implementation allows for adjusting weights for fields and TF-IDF values, we wanted to try and optimize these weights using machine learning. To do this we used genetic algorithms, which we along with the fitness functions described in chapter 3. The results are presented in this section, and partially elaborated on.

The full discussion on the results, and conclusion about what experiments were better, please see chapter 5. This chapter will only present the findings.

This section does not show all the optimized weights for each of the generation for clarity of reading, however, to see full combination of weights and results for each experiment, please see section E in the appendix. The tables there also provide more decimals for each weight, in case you want to use them in the implementation.

Ex.	Gen.	TF-IDF	Dataset	Description
0	-	Calls/Imports	All	Control with default weights.
0.1	-	Disabled	All	Default weights, no TF-IDF.
0.2	-	Calls/Imports	Train/Test	Only optimized for F-score.
1	1, 5, 10	Calls/Imports	Train/Test	Train/Test (80/20) split.
2	1, 5, 10	Calls/Imports	All	All samples (no train/test).
3	1, 5, 10	Calls/Imports	Train/Test 100 RND	100 random of benign samples.
4	1, 5, 10	Disabled	Train/Test	Train/Test (80/20) split.
5	1, 5, 10	Disabled	All	All samples.
6	1, 5, 10	Disabled	Train/Test 100 RND	100 random of benign samples.
7	1, 5, 10	Calls	Train/Test	Disabled for imports.

Table 4.1: An overview of our concrete experiments. Experiments 1 through 3 are **with TF-IDF**, and experiments 4 through 6 are **with TF-IDF disabled**. Experiment 7 is an extra experiment we perform.

In table 4.1 we can see an overview of the experiment we performed. Below we describe more in detail what each of the experiments do. We also perform the same experiments again later in the chapter with a slightly different implementation of TF-IDF, but more on that towards the end of the chapter.

Experiment 0, 0.1 (along with **Experiment 0.2** described next) were our control experiments. These were used as baselines for the rest of the experiments. They measure F-score, bulletin count, and a fitness score just like the other experiments and were meant give us a control to match against the other results. The weights were set to 1.0 for these.

Experiment 0.2 was performed to tell us the maximum detection rate for our implementation. This is as discussed in 3.2.5, where we mentioned we wanted find a maximum F-score. To do this we optimized for only the F-score by simply

returning it in the fitness function, since higher values equals better fitness. The higher the F-score the better. The difference from the normal fitness function is that we then omit the bulletin count.

Experiment 1 is meant to be a standard 80/20 train-test split. We do this to discourage overfitting the weights to our results, we did also consider using K-Fold for this, but as we will discuss in section 5.2, we choose not to do so for this thesis.

Experiment 2 is made to see what would happen if we optimize the weights without using a train/test split, and the last one is a combination with train/test and a selection of 100 random benign packages of the 441 benign packages we have downloaded.

Experiment 3 was made to test how the weights fair against being optimized with approximately a third of the benign dataset. Each run picks 100 random benign packages of the 441. These 100 are then used in a train/test split. The malicious dataset stays the same, but is also affected by the train/test split. This was to see how the performance scales between this experiment and the full-test experiment without train-test splitting.

Experiment 4, 5, and 6 are almost equal to **experiment 1, 2 and 3** respectively. The critical change is the usage of TF-IDF, which was disabled for both imports and functions. As mentioned in section 3.2.5, we wanted to also test how the TF-IDF did on function calls and imports, since Fang et al. used this on opcodes [19].

Experiment 7 was created because the results from the previous tests prompted us to at the time to experiment more with the combination of TF-IDF. This is the same as experiment 1, but without TF-IDF for imports. We will elaborate on why when we present the results, which will also be discussed in chapter 5.

We start of by showing the results from the control experiments.

4.3.4 Control Experiments

Experiment 0 and 0.1: The first two control experiments were simple runs, where we only set the default weights. When we say we did a **full run**, we mean a single pass over the full dataset of benign and malicious packages. The results from this can be seen in table 4.2. This was on the whole dataset of both malicious and benign results (what we denote as "all" in our tables). For **experiment 0** we get an F-score of 0.44 and a bulletin count of 2172, which results in about 4.9 bulletins per package on average (with 441 benign packages in our dataset as previously mentioned). This results shows us that this run did not generate bulletins for half of the requested malicious code.

To reiterate; The results we are looking for have a **high F-score** (which are practically percentages, but are displayed as decimals), a **low bulletin count**, and a **high as possible fitness**.

Experiment	TF-IDF	F-score	Bulletins	Fitness
Ex. 0	Yes	0.44	**2172**	**2.04**
Ex. 0.1	No	**0.78**	9662	0.81

Table 4.2: Control experiment using all weights set to 1.0. - **Ex. 0 has TF-IDF and Ex. 0.1 does not.**

Experiment 0.2: The next control was optimized only using F-score as fitness. This experiment also used TF-IDF and went through 10 generations of evolution.

Table 4.3 shows the results from this run with the genetic algorithm. The values **highlighted in bold are the best ones** for the column they are in. How we chose the *best* values are discussed more in detail in chapter 5.

Gen.	F-score	Training Fitness	Testing Fitness	Time
1	0.83	0.78	**4.64**	14m
5	**0.88**	0.78	4.12	38m
10	0.86	**0.81**	3.69	1h 7m

Table 4.3: Control **experiment 0.1** - Optimized for F-score only (TF-IDF on function calls and imports, train/test split, only malicious samples).

The time column in 4.3 show how long it took to get to that specific generation. That the time for the 5th generation is the *time it took to get to that evolution*. It is *not* the time between evolution's. For example, the F-score only optimization took 1 hour and 7 minutes to complete all its generations from 1 to 10.

Figure 4.1 shows the fitness and F-score from the three control experiments on a single graph. We can see that the control experiment with no TF-IDF is comparable to the experiment where we only optimize for F-score.

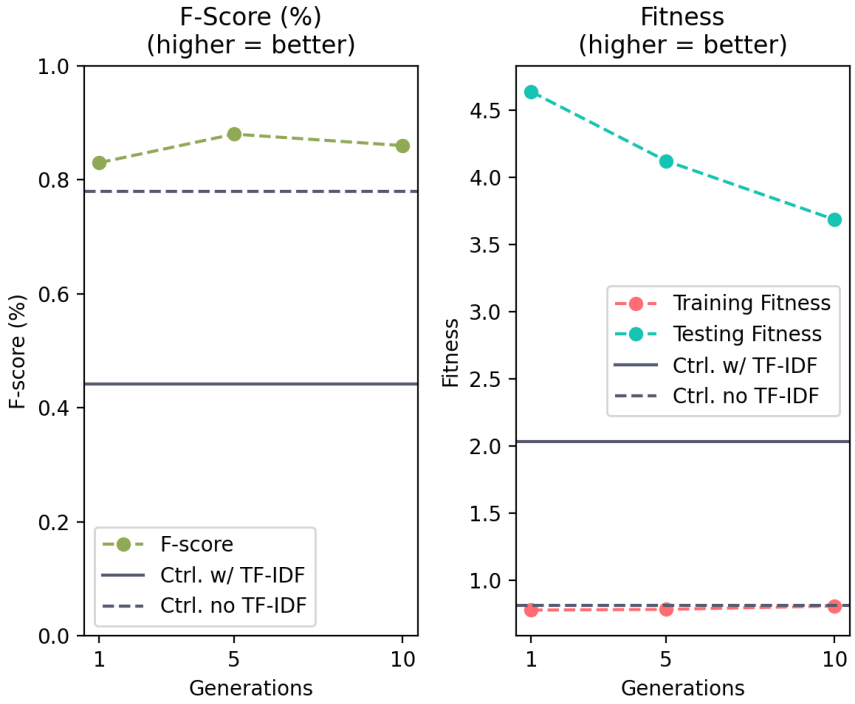

Figure 4.1: Control experiments - F-score and fitness for the control experiments compared.

In summary: We can tell from figure 4.1 that the F-score performance of the F-score-only optimized weights are comparable to the control experiment with no TF-IDF (as mentioned all weights were set to 1.0 for experiment 0 and 0.1). The F-score is almost halved by the introduction of TF-IDF, but we can see in table 4.2 that the bulletin count goes from 9662 to 2172 with TF-IDF enabled.

4.3.5 Experiments with TF-IDF

Next, we will show the results from the TF-IDF **experiments 1 through 3**. They are shown in table 4.4 below. The weights for these experiments can be found in table 4.6, but will make more sense when these are discussed and analyzed in chapter 5.

The experiments used a total time of 16 hours and 54 minutes to complete. As previously mentioned we highlight the best values in their column by marking it in **bold**. Do note that the *bulletin count* is read from the testing run. This means the 1st experiment in generation 5 got 291 bulletins and during that run got 26.39 as a fitness score. It is critical these results are viewed with the context of the full runs, so the experiments have a common testing ground (table 4.5).

Ex.	Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
	1	0.37	326	3.22	11.47	1h 26m
1	**5**	**0.77**	**291**	7.90	**26.39**	3h 43m
	10	0.67	301	**8.11**	22.08	6h 26m
	1	0.46	1191	3.87	3.86	1h 44m
2	5	0.56	606	10.14	9.21	4h 41m
	10	**0.58**	**339**	**17.11**	**17.05**	8h 9m
	1	0.51	174	62.61	29.16	36m
3	5	0.43	235	82.17	18.09	1h 23m
	10	**0.52**	**119**	**84.61**	**43.57**	2h 19m

Table 4.4: Results from **experiment 1 to 3** (as shown in or overview table 4.1) - The best results for each column are **highlighted in bold** for each experiment. - The generations highlighted in cyan were deemed the best for that experiment.

The cyan-colored generations were chosen to be evaluated through a *full run*. As mentioned in the introduction to this section, a *full run* is what we call a run of a set of weights across all the benign and malicious samples we have. This

Ex.	Gen.	F-score	Bulletins	Fitness
0	-	0.44	2172	2.04
0.1	-	0.78	9662	0.81
1	5th	**0.67**	1136	5.89
2	10th	0.58	339	17.05
3	**10th**	0.62	**222**	**27.78**

Table 4.5: Full run with the best weights from **experiment 1, 2 and 3 + controls 0 and 0.1** - A graph of this can be seen in figure 4.2 - The best results from experiment 1, 2 and 3 are **highlighted in bold**.

gives the weights a common testing dataset when they have been optimized with different ones. In table 4.5 we can see the results from the weights we regarded as the best from **experiment 1, 2, and 3**. The control experiments are included as comparison.

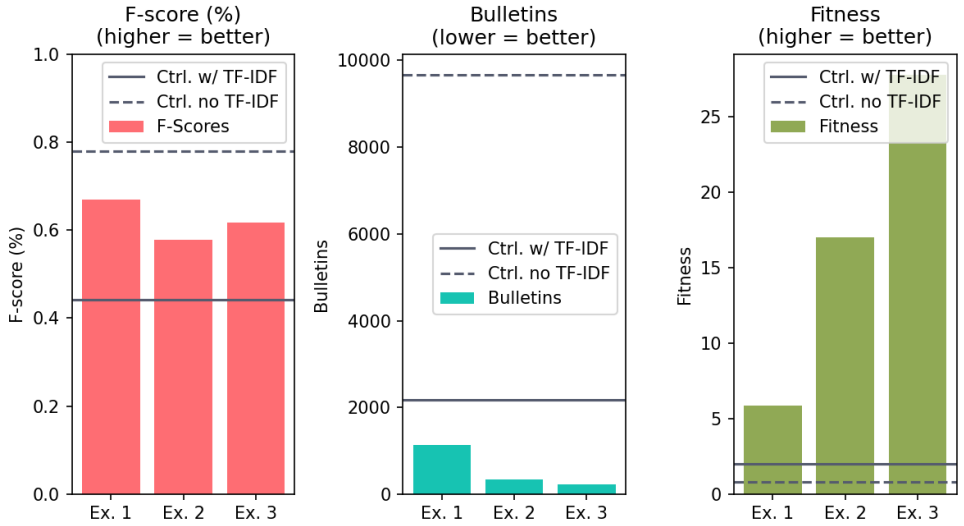

Figure 4.2: Experiment 1 to 3 Best Weights Full Run - F-Score, bulletin count, and fitness are shown for a full run for the best weights from each experiments (highlighter in 4.4. We can tell that experiment 3's weights performed the best.

These results were visualized in figure 4.2. We also overlay the first two controls (omitting the F-score-only control) on the graph for comparing the results from the optimization. Do note the bulletin count is from the *testing* split in the experiments that use this, which are experiment 1 and 3. Experiment 2 uses all the samples, which means the best generation here is considered a *full run*. It was a simple task to pick out the best generation within a given experiment just based on the other best values.

In summary: Experiment 1 to 3 performed right in the middle of our controls in terms of F-score, hovering at around 60%. The bulletin count was consistently and considerably better than the controls, and any of the weights would outperform it. The fitness values (right in the figure 4.2) reflect this good combination of F-score and bulletin count in experiment 3. Experiment 3 was the one with 100 random benign packages, instead of the full samples of 441. The time for this experiment was also 1/4 of experiment 2, which was trained on all samples.

Ex.	Gen.	FW Func.	FW Im.	FW Be.	FW Str.	TW Func.	TW Im.
1	1	0.21	0.85	-0.13	0.72	1.26	1.49
	5	0.21	0.85	-0.13	1.90	1.22	0.29
	10	0.21	0.85	-0.13	2.10	1.22	0.29
2	1	0.87	0.68	0.61	1.05	0.59	0.27
	5	0.13	0.68	0.48	1.05	0.76	0.09
	10	0.13	0.68	-0.60	1.05	0.01	0.09
3	1	0.09	0.73	0.90	1.18	0.87	0.35
	5	0.19	0.70	0.90	1.18	0.87	0.71
	10	0.09	0.65	-0.09	-0.10	0.87	0.03

Table 4.6: Optimized weights from **experiment 1 to 3** (from table 4.4)

4.3.6 Experiments without TF-IDF

This next part will focus on doing the same experiments as 1 to 3, but with TF-IDF disabled. The overview for all the experiments can once again be found in 4.1. Do note that due to technical reasons we still feed the genetic algorithm with six genes, even though the last two weights for TF-IDF will have no impact on the results.

For the control experiments we refer back to table 4.2 in the previous section. THe control with no TF-IDF resulted in 78% as F-score, 9662 bulletins, and the fitness value was considerably lower than the other control with TF-IDF (due to the much increased bulletin count).

Ex.	Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
	1	0.17	117	163.6	14.11	34m
4	5	**0.23**	66	216.9	34.86	1h 26m
	10	0.21	**0**	**7.7e+10**	**2.1e+11**	2h 30m
	1	**0.13**	450	2.90	2.90	0h 39m
5	5	0.08	64	12.02	12.02	1h 42m
	10	0.08	**0**	**7.7e+10**	**7.7e+10**	2h 56m
	1	**0.39**	156	3017.1	**24.81**	0h 17m
6	5	0.0	**141**	9.5e+10	0.0	0h 41m
	10	0.17	**141**	**1.3e+11**	11.71	1h 13m

Table 4.7: Results from **experiment 4 to 6** (as shown in or overview table 4.1) - The best results for each column are **highlighted in bold** for each experiment. - The generations highlighted in cyan were deemed the best for that experiment.

The results from the no TF-IDF experiments can be seen in table 4.7. As we have mentioned, these experiments are partially the same as experiment 1 to 3, but they have TF-IDF disabled.

We immediately see that these results differ a lot from the TF-IDF experiments. The F-score is generally much lower, ranging from a maximum value of 39% all the way down to 0% (worse than both our controls), but bulletin count is consistently much lower than our previous experiments. This count reaching all the way down to zero at the 10th generation in both experiment 5 and 6.

We only performed a full run for the 10th generation for experiment 4, because the 6th experiment did not have a clear best generation, and the best weights in

F-score	Bulletins	Fitness
0.09	0	9.5e+10

Table 4.8: Experiment 4 10th generation - All samples run

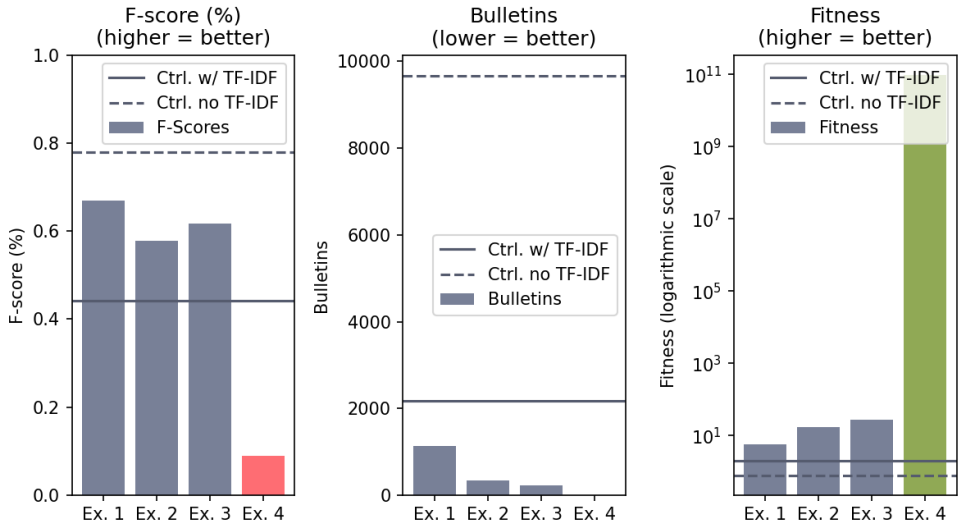

Figure 4.3: Experiment 4 Best Weights (10th generation) Full Run - compared to previous results - F-Score, bulletin count, and fitness are shown for a full run for the best weights from each experiments. - The new additions are colored, while the previous ones are grayed out

experiment 5 would be worse than those in experiment 4. The results from this full run can be seen in table 4.8. In figure 4.3 we can see this result compared to the previous full runs of the experiments 1 to 3. The bulletin count of this experiment is not visible because it is zero. Also note that the fitness graph is a logarithmic scale to be able to compare them visually. The new additions are colored, while the previous ones are grayed out.

The weights for these experiments can be found in table 4.9, but will make more sense when these are discussed and analyzed in chapter 5.

These experiments also used a lot less time on average. The maximum time required for an experiment for this group was 2 hours and 56 minutes. This experiment with all the samples does go through considerably more benign packages. The quickest of the experiments were experiment 6, which only used 1 hour and 13 minutes to complete.

In summary: The TF-IDF experiments, as illustrated by table 4.7 and figure 4.3, show us that we can get results with a considerably lower amount of bulletins compared to our previous experiments, but that this comes at the cost of F-score.

Ex.	Gen.	FW Func.	FW Im.	FW Be.	FW Str.	TW Func.	TW Im.
	1	0.52	0.29	0.31	1.06	0.22	0.46
4	5	-0.46	0.55	0.31	0.40	-0.01	1.58
	10	0.36	-0.45	0.31	1.92	0.11	0.97
	1	0.05	0.13	1.54	1.54	0.05	0.55
5	5	0.05	0.13	0.98	1.54	0.05	0.55
	10	-0.27	0.02	0.33	2.07	0.05	0.55
	1	0.53	0.58	0.10	1.61	0.43	0.85
6	5	0.53	-0.32	0.10	0.72	0.98	0.64
	10	0.55	-0.36	0.25	1.06	0.98	1.65

Table 4.9: Optimized weights from **experiment 4 to 6** (from table 4.7) - We can discard the last two weights, as they are TF-IDF weights.

4.3.7 Experimenting with TF-IDF Disabled for Imports

During the experimentation of the first three experiments, we saw a trend in the weights that were generated. The weights themselves can be seen in table 4.6 in the section on experimenting with TF-IDF. This trend was that TF-IDF on imports tended to be low. This prompted us to perform an **Experiment 7 with TF-IDF disabled for imports**, but enabled for function calls. This experiment can also be seen in our overview table 4.1. This meant running 10 generations with a train/test dataset.

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.21	154	4.91	13.91	1h 14m
5	0.66	122	15.86	53.91	3h 15m
10	**0.66**	**90**	**23.33**	**73.08**	5h 37m

Table 4.10: Results from **Experiment 7 - Import TF-IDF disabled** (train/test split).

Table 4.10 show the results from this experiment. We did see a good result with 0.66 in F-score and 90 bulletins in total on the 10th generation. We do have to be aware that this result is from the test-portion of the dataset. To get a better idea if this result would be consistently good, we perform a full run using the weights from the 10th generation.

F-score	Bulletins	Fitness
0.63	230	27.62

Table 4.11: All samples run with 10th generation weights from *Experiment 7 - Import TF-IDF disabled*.

In table 4.11 we can see the results from a full run using the 10th generation of our experiment above. The result we got is comparable to the one in table 4.10 above. Figure 4.4 visualizes this result in comparison to the previous results. The previous results are grayed out for clarity, while our new result is colorized. From this graph we can easily tell that the performance of this experiment closely matches that of experiment 3 (TF-IDF, train/test, 100 random benign packages).

In summary: This experiment with TF-IDF disabled for imports yielded similar results to an experiment with TF-IDF enabled for both imports and calls. Specifically, this experiment (7) was similar to experiment 3 in performance, with our full run getting 63% F-score and 230 bulletins, and experiment 3 getting 0.62% and 222 bulletins. This results are discussed further in chapter 5.

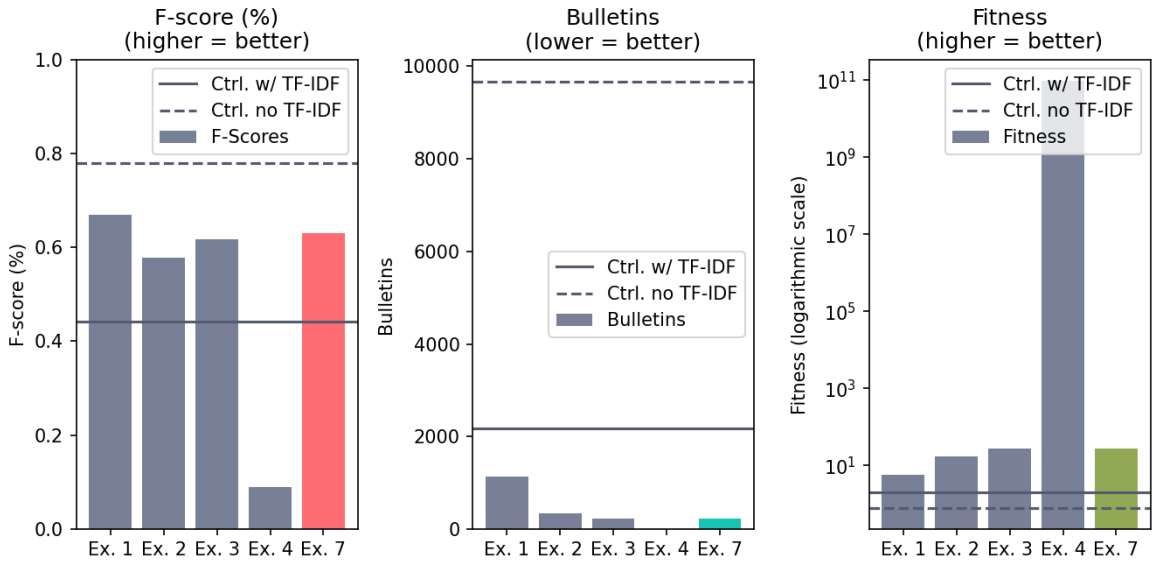

Figure 4.4: Experiment 7 Best Weights (10th generation) Full Run - compared to previous results - F-Score, bulletin count, and fitness are shown for a *full run* for the best weights from each experiments.

Gen.	FW Func.	FW Im.	FW Be.	FW Str.	TW Func.	TW Im.
1	0.48	0.42	0.46	1.53	0.97	0.28
5	0.01	0.66	0.46	1.76	0.97	0.28
10	0.01	0.65	-0.46	1.76	0.97	0.28

Table 4.12: Weights for **Experiment 7 - Import TF-IDF disabled** (train/test split) - The TF-IDF weight for imports can be ignored here, as it is not used.

4.3.8 Changing the IDF-function

We wanted to perform one more experiment where we changed the IDF-function. As mentioned in section 3.2.5, we modified the TF-IDF function by replacing the IDF function with one we called document frequency (DF) (equation 3.9 in chapter 3). The intention was to see if we could mitigate penalizing results in smaller packages.

We ran **Experiment 1, 2 and 3 again** with this modified function. The results can be found in table 4.13. Do note again that this experiment used a train/test split. But the results were promising. The 5th generation in **Experiment 1 w/ DF** outperformed generation 1 and 10 in terms of F-score, suggestion that it might perhaps over-fit if we ran it for longer.

Ex.	Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1 w/ DF	1	0.4	11.0	13.04	363.6	1h 49m
	5	**0.62**	33.0	16.16	188.1	4h 1m
	10	0.35	**9.0**	**21.87**	**391.1**	6h 40m
2 w/ DF	1	**0.23**	65.0	35.05	35.05	1h 53m
	5	0.08	4.0	256.4	192.3	4h 56m
	10	0.08	**0.0**	**7.7e+10**	**7.7e+10**	8h 22m
3 w/ DF	**1**	**0.25**	**8.0**	**3.9e+10**	**307.0**	0h 33m
	5	**0.25**	**8.0**	**3.9e+10**	**307.0**	1h 17m
	10	**0.25**	13.0	**3.9e+10**	188.9	2h 1m

Table 4.13: Results from **experiment 1 to 3 with DF instead of IDF** (as shown in or overview table 4.1) - The best results for each column are **highlighted in bold** for each experiment.

The results from the experiments were varied. We decided to therefore perform two full runs using two set of weights from *Ex. 1 w/ DF* and *Ex. 3 w/ DF*. This meant we could compare it to the results from *Ex. 2 w/ DF*. The results from this can be seen in table 4.14. Here we observed the weights have a good F-score for *Ex. 1 w/ DF*, but with a bulletin count of 8663 it lowered the fitness values

Ex.	Gen.	F-score	Bulletins	Fitness
Ex. 1 w/ DF	5	**0.76**	8663	0.87
Ex. 2 w/ DF	10	0.08	**0**	**7.7e+10**
Ex. 3 w/ DF	1, 5	0.06	51	11.42

Table 4.14: All samples run with weights from *Ex.1 (5th gen.) and Ex. 3 (1st and 5th gen) changing IDF to DF*. We also include *Ex. 2 w/ DF* since it can be compared to a full run.

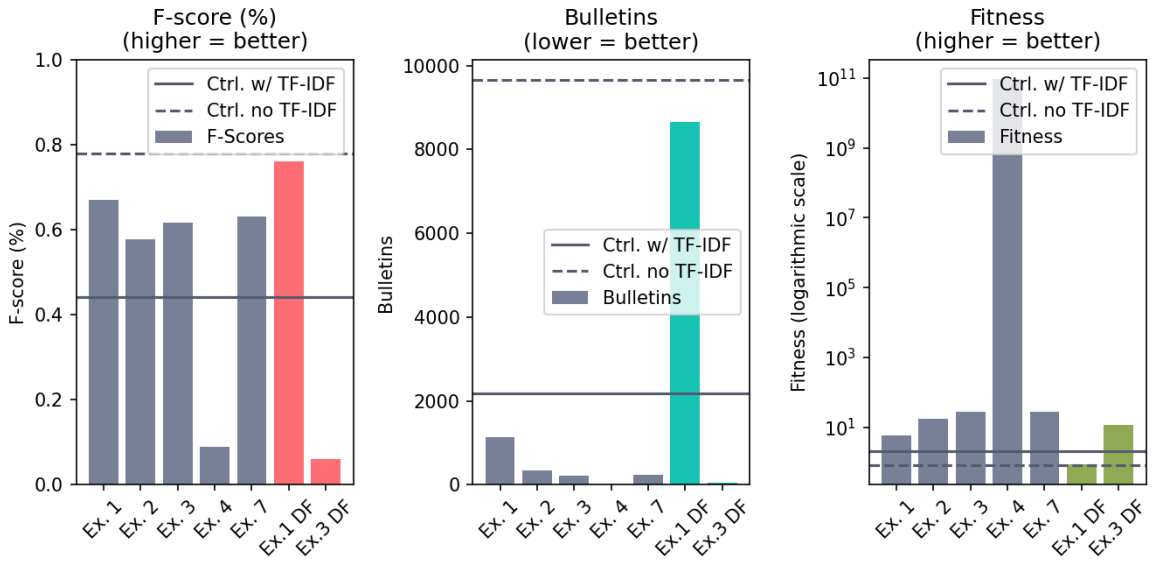

Figure 4.5: Experiment 1 and 3, w/ DF, Best Weights Full Run - compared to previous results - F-Score, bulletin count, and fitness are shown for a *full run* for the best weights from each experiments.

drastically. Both generation 1 and 5 in *Ex. 3 w/ DF* had the same weights, as seen in table 4.15, which is why they are both highlighted. We also observed that *Ex. 2 w/ DF* is almost identical in performance when we compared it to experiment 4 and 5. Both had 8% F-score and 0 bulletins.

Figure 4.5 compared the full run result from our chosen experiments. From this graph we saw that higher F-score strongly correlated with a higher bulletin count. *Ex.3 w/ DF* matched with experiment 4 in terms of F-score and bulletin count, but we notice the fitness values are widely different. We do note that the full run for experiment 4 had 0 bulletins, while *Ex.3 w/ DF* got 15.

In summary: Our experiments with the TF-DF function enabled for imports and functions gave us results where two experiments exhibited high correlation between F-score and bulletin count, which is not what we wanted. *Ex. 2 w/ DF* on the other hand, had almost the same results as experiment 4 and 5 (no TF-IDF functions and imports).

Ex.	Gen.	FW Func.	FW Im.	FW Be.	FW Str.	TW Func.	TW Im.
1 w/ DF	1	0.32	1.06	0.23	1.88	1.16	1.09
	5	0.32	1.26	-0.50	1.88	1.16	1.09
	10	1.41	0.63	0.44	1.88	1.16	1.09
2 w/ DF	1	1.61	0.54	0.42	1.44	0.94	1.07
	5	-0.02	-0.27	0.31	1.44	0.94	1.07
	10	-0.02	0.23	0.31	1.44	0.94	0.0
3 w/ DF	**1**	0.57	0.98	0.70	0.65	1.59	1.60
	5	0.57	0.98	0.70	0.65	1.59	1.60
	10	0.57	1.11	0.70	0.11	1.59	1.60

Table 4.15: Optimized weights from **experiment 1 to 3 with DF instead of IDF** (from table 4.13)

4.3.9 Experimenting with alternate DF and disabled imports

Due to the results in the previous section, as shown in figure 4.5, we wanted try combining experiments. We decided to run an experiment with TF-DF-function but disabled it for imports, just like experiment 7 (see overview table 4.1).

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	**0.73**	662	4.15	11.04	1h 20m
5	0.21	**0**	**3.9e+10**	2.1e+11	3h 27m
10	0.35	**0**	**3.9e+10**	**3.5e+11**	6h 1m

Table 4.16: Results from 10th generation weights in **Ex. 7 w/ DF on Calls**

F-score	Bulletins	Fitness
0.11	0	1.1e+11

Table 4.17: All samples run with 10th generation weights from *Ex. 7 w/ DF on Calls*.

The results from this experiment can be seen in table 4.16. We see the last two generations have a very high fitness value for the testing split.

We also perform a full run for the best weights from the 10th generation in the experiment. The results can be seen in table 4.17 and as we can see the trend of 0 bulletins is continuing, but the F-score is only slightly better than the previous result. The run is visualized in figure 4.6 where it is compared to previous full runs. To see previous results please see our previous sections.

This is the most promising results where we have got 0 bulletins. The results from this full test closely match experiment 4, which was the train/test experiment were TF-IDF was disabled for both imports and functions (as seen in our overview 4.1), only that the F-score is slightly better. As for the difference between the experiments; we have the DF instead of IDF-function, and TF-DF then enabled on functions.

In summary: This experiment was performed with the modified DF-function, but only for function calls. The experiment was run on our train/test dataset, and the results were promising. A full run showed us that the 10th generation weights got an F-score of 11% and a bulletin count of 0. This experiment closely matched the results of experiment 4.

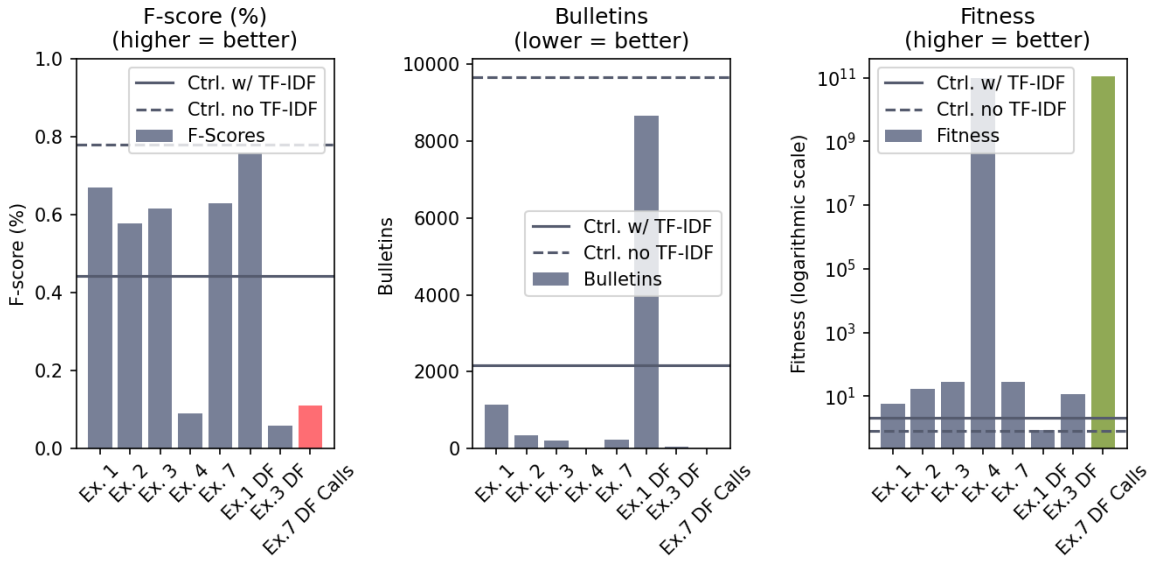

Figure 4.6: Experiment 1 and 3, w/ DF, Best Weights Full Run - compared to previous results - F-Score, bulletin count, and fitness are shown for a *full run* for the best weights from each experiments.

Gen.	FW Func.	FW Im.	FW Be.	FW Str.	TW Func.	TW Im.
1	0.30	0.84	0.36	1.48	1.72	0.65
5	0.44	0.12	0.36	2.07	1.72	0.65
10	0.72	-0.03	0.36	2.53	1.72	1.40

Table 4.18: Weights from **Ex. 7 w/ DF on Calls** (train/test split).

Chapter 5

Discussion and Conclusion

This chapter discusses the previously shown results while also analyzing them. We also reflect on the decision talking about the validity and the limitations of our approach. This is also the chapter that addresses the research questions.

Through the thesis we got a lot of points to discuss and reflect on, therefore this chapter is divided into sections. First section 5.1 covers the results from the previous chapter, section 5.2 then discusses the experimental design. The last two sections 5.3 and 5.4 cover the limitations and validity of our results and methods. Lastly, we present our conclusion of the thesis in section 5.5.

5.1 Experimental Results

This section will discuss the results from chapter 4. This is done in the same order they appear in the results, and each section provides a summary for each part.

TF-IDF

The first three experiments we performed with TF-IDF yielded good results compared to our controls, as we mention in the summary. The results can, again, be seen in figure 4.2. Here we can see these experiments performed consistently better than the control with TF-IDF, and the bulletin count was also considerably lower. The fitness values for these first three experiments show an inverse correlation between F-score and bulletin count, which is what we wished to see (as mentioned this is because we want a high F-score and a low bulletin count). These results, specifically from the 10th generation of experiment 3, are so far the best overall set of weights. This result gave us an F-score of around 60%, and the bulletin count was 222, which compared to the control result of 2172 is almost ten times better. This could with some work function well on smaller scale analysis, for example, one package and a selection of their direct dependencies (RQ_3).

Now that we have seen the results from these experiments, we can also look at the weights to see if we see any particular similarities between the three experiments. The weights can be seen in table 4.6, and we this since they all performed

reasonably well. They do tell us at least three interesting behaviors; (1) All weights penalize the field weight for the functions, reducing it to between 0.09 and 0.21 for their last generations (just a reminder the default weights are 1.0). (2) We see the field weight for imports being reduced, but not too much. (3) We can also tell the TF-IDF weights for imports were reduced drastically for all experiments 10th generation. In practice this means we do not use the calculated TF-IDF value, practically disabling it.

We are also aware we might see different results from these weights if we run the optimization process several times, and run a full run for those as well, which we could not do due to the speed of our experiments (which we discuss later in this chapter).

We picked the *best* generation for each experiment to perform a full run. The intention of the full run was to give all experiments a common testing ground. The way we chose the best ones was by looking at a combination of bulletin count and F-score and trying to find a balance between them. The fitness score was not always reliable in showing an equal prioritization of the two in our opinion. We speculate this could be improved by adjusting the fitness function.

In summary: The first three experiments showed an inverse correlation between bulletin count and F-score, meaning we observe a low bulletin count when we have a high F-score. The best set of weights gave us 60% F-score and 222 bulletins. We can also tell the TF-IDF weights for imports were reduced drastically for all experiments 10th generation.

No TF-IDF

The results from the non-TF-IDF tests were a lot more varied. While experiments 1 to 3 were all pretty consistent, experiments 4 to 6 were sacrificing F-score for bulletin count much more than we had anticipated. The maximum F-score value from all the generations was 39%, which is worse than both the controls. The worst F-score was 0%, which still had bulletins. We decided the best result in terms of the trade-off between F-score and bulletin count was experiment 4, generation 10, with 21% F-score and 0 bulletins. In the actual full run, we got a 9% F-score and 0 bulletins. This full run result closely matches the 10th generation of experiment 5. We note that experiment 2 and 5 practically only do full runs to train and test their weights. This means experiment 5 results are comparable to full runs of experiment 4 [1]. By seeing the result for experiment 4 and 5 match in this manner, we could perhaps draw a conclusion that train/test-splitting does not impact optimization, but more research on this is needed.

In terms of bulletin count, experiment 4 and 5 is the best we have got so far and actually has the potential to be feasible to scan the entire PyPI registry. If we could detect even 8-9% of all malicious packages we scan through, it would still be fewer malicious packages in PyPI. Even though the F-score is low we have to look at it in the context of over 380'000 packages, which is what PyPI currently has.

[1] Which is why experiment 5's 10th generation is highlighted in cyan as well in the result tables.

With no TF-IDF, a single run over all the 441 benign packages takes approximately 100 seconds, which is almost 10 packages per second. We can then estimate we would spend 38000 seconds, or roughly 10 and a half hours to scan PyPI using one running instance of the implementation. This does not account for the time it takes to extract the Zip file in memory. In comparison, the same run with TF-IDF for imports and calls takes 243 seconds or roughly 4.1 packages per second. Meaning a full PyPI scan would take almost 26 hours.

These results without TF-IDF were also a lot quicker than the ones with TF-IDF. In a future section on speed and performance (5.1) we will discuss this further, but in general, the experiments with TF-IDF disabled were 65% faster than those with it enabled. This would again make it more feasible for large-scale deployment (RQ_2).

The weights for these experiments (shown in table 4.9 in the previous chapter) are not as consistent as the previous ones we analyzed. We ignore the TF-IDF weights for experiments 4 to 6, but include them for completeness. We can at least look into the weights for our best generation, which was the 10th one in experiment 4 and 5; This experiment got 9%- and 8%F-score, respectively, and zero bulletins as previously mentioned. Their weights do not match, except for the field weights for the behaviors and the strings/canaries. Those are almost a complete match, which is interesting. The field weights for functions and imports are pretty far apart, but both get penalized quite a lot.

The fitness value in these experiments also skyrocketed. This is likely because of our fitness function and the small value we add to the bulletin count in case it it zero to stop divide-by-zero exceptions. We also want 0 bulletins, which is why we do not discard results and add a small value. The function worked exactly as designed (shown in equation 3.10, chapter 3.2.5), but we did not expect it to go this high. Here is an example with a very medium and low bulletin count, showing us how the fitness value increases drastically:

$$Fitness = 10000 \frac{FMeasure}{BulletinCount + 0.00000001} \tag{5.1}$$

$$\tag{5.2}$$

$$27 = 10000 \frac{0.60}{222 + 0.00000001} \tag{5.3}$$

$$\tag{5.4}$$

$$600000000000 = 10000 \frac{0.60}{0 + 0.00000001} \tag{5.5}$$

In the future, it might be reasonable to add a *max()* function to the fitness function, where instead of the small value we add, pick the maximum value of the bulletin count and a value of 1.

In summary: Experiment 1, 2, and 3 were performed anew with TF-IDF disabled. The results showed a huge decrease in time for the experiments to complete. In general the experiments with TF-IDF disabled were 65% faster than those

with it enabled. Experiment 4 and 5 got 9%- and 8%F-score, respectively, and zero bulletins, which is so far the best result in regards to bulletin count.

TF-IDF Disabled for Imports

As mentioned previously in the discussion, the optimized weights for experiment 3 reduced the TF-IDF weight for imports drastically, practically disabling it. Because of this, we performed experiment 7, which disabled TF-IDF for imports. Figure 4.4 in results showed the full run for this experiment compared to the previous ones, which clearly showed us the results are almost identical to experiment 3. This was expected but proved we did not need half of the TF-IDF calculations for practically the same results. If also want to compare times; All 10 generations in experiment 7 spent 5 hours and 37 minutes, and all 10 generations in experiment 1 took 6 hours and 26 minutes. We use experiment 1 for comparison since both used train/test datasets. From these values, we can see our experiment with TF-IDF disabled spent 1 hour less in terms of time, which is a 12.7% improvement in time.

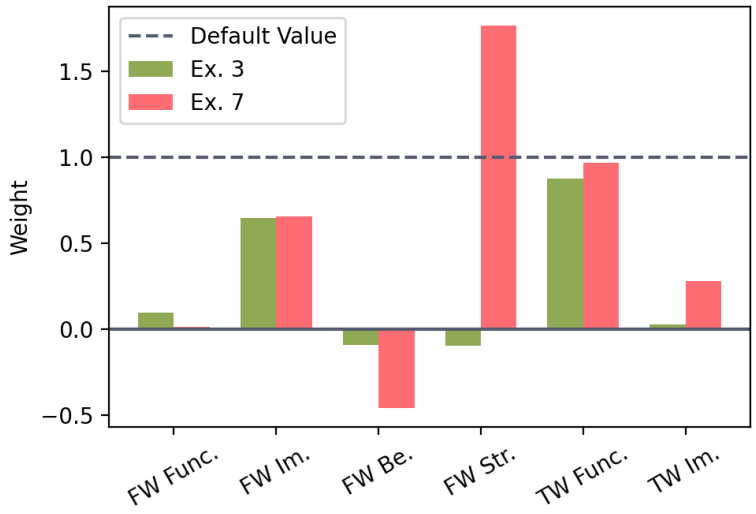

Figure 5.1: Weight comparison of **Ex. 3** and **Ex. 7** (both are the 10th generation weights from their respective experiments.)

We wanted to compare weights from experiments 3 and 7 since they are closely matched, which we can see graphed in figure 5.1. Again, we can ignore the TF-IDF weight for the imports from experiment 7 (right-most red bar in the graph). We see the field weight for functions, imports, and partially the behavior, closely matches. TF-IDF weights also closely match. The close match of the TF-IDF Import field weights matching even when it does not affect the performance of the fitness could be a coincidence, but this remains an open question.

We observe that the field weight for strings does not match. This suggests the dataset does not contain many or any, Base64 encoded payloads that it could find.

In summary: Experiment 3 (TF-IDF import and functions) and 7 (TF-IDF functions) show similar results, which is expected, since Experiment 3 penalized TF-IDF weights for imports. Disabling this function for this experiment yielded a 12.7% improvement in time. It also helps to prove we can disable this functionality and get similar results.

Changing the DF Function

We also performed experiments where we changed the IDF function in TF-IDF to a new function, *DF*. The results from these experiments are displayed in table 4.13. As we mentioned in the results chapter, *experiment 2 w/ DF* closely matched experiments 4 and 5 in terms of F-score and bulletin count. This might suggest TF-DF exhibits some of the same results at experiments without TF-IDF. This is interesting because the results from *Ex. 1 w/ DF* and *Ex. 3 w/ DF* had a high correlation between F-score and bulletin count. This is the inverse of what we wanted, and if we needed results with a high coverage rate (high F-score and very high bulletin count), we have would have run with default weights and TF-IDF disabled (see control experiments 4.5).

When looking at the weights for *Ex. 2 w/ DF* in table 4.15, we see TF-DF weight for imports being reduced to zero, meaning TF-IDF for imports become practically disabled. We also observe the field weight for functions being reduced to close to zero. This also means bulletins on functions in practice would have almost no effect on the result. Field weights for imports and behavior have also been heavily penalized compared to the default value of 1.0. This is a trend we have been seeing from table 4.6 (Ex. 1 to 3) and 4.12 (Ex. 7 TF-IDF Calls Only).

In summary: Results from *Ex. 2 w/ DF* match closely with experiments 4 and 5. For the weights belonging to *Ex. 2 w/ DF* we observe the field weight for functions being reduced to close to zero. Meaning functions in practice have no effect. We also see TF-DF weight for imports being reduced to zero, meaning TF-IDF for imports becomes practically disabled.

DF function Disabled for Imports

As we saw a trend of TF-IDF weights being severely reduced in several experiments, we decided to combine this with the modified TF-DF function. The results can be seen in 4.16chapter 4, and as mentioned in the summary; this experiment closely matched experiment 4. While experiment 4 got 9% F-score (table 4.8), this experiment achieved 11% in F-score. Without further re-runs and validations we will withhold a certain conclusion this is a 2% improvement. More testing is needed.

In figure 5.2 we compare the weights from the two experiments (as seen in table 4.9 and 4.18). Experiment 4 is displayed in green, and *Ex. 7 w/ DF Calls Only* in red/pink. We can see some similarities, for example in the field weight for

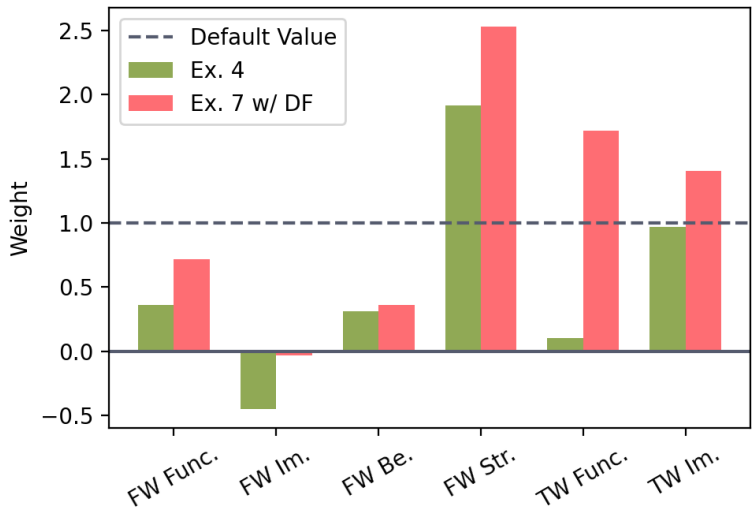

Figure 5.2: Weight comparison of **Ex. 4** and **Ex. 7 w/ DF Calls Only** (both are the 10th generation weights from their respective experiments.)

functions and behavior. The field weight for imports does to some degree match. Since experiment 4 did not utilize TF-IDF, we cannot directly compare those values. This means the only valid TF-IDF weight to look at is the one for function calls. By (assumed) coincidence the TF-IDF value on experiment 4 was also close to zero, which would have practically disabled it as well. Since this was the biggest change, we could speculate the TF-IDF for function calls increased the F-score by a couple of percent, but more testing is needed. Currently, the TF-IDF weight for *Ex. 7 w/ DF Calls Only* amplifies the TF-IDF value by about 1.7.

In terms of time used, this experiment used 6 hours and 1 minute. Compared to experiment 4 (our closest match) which used 2 hours and 30 minutes. This means this new experiment spent 41% more time.

In summary: Due to a trend with the optimized weights practically disabling TF-IDF for imports, we performed a combined experiment with the new DF-function while also disabling TF-IDF for imports. The result from this experiment (*Ex. 7 w/ DF Calls Only*) closely matched the ones of experiment 4 (no TF-IDF, train/test dataset). While matching experiment 4 in terms of F-score and bulletin count, this experiment spent 41% more time.

Speed and a Note on Performance

We can tell by the experiment time on experiments *with* and *without* TF-IDF that this has a big impact on performance (tables 4.4 and 4.7). We compare the time it took for experiments 2 and 5 to finish, as they both trained and tested on the whole dataset; Experiment 2 used 8 hours and 9 minutes to finish, and experiment 5 used 2 hours and 56. This meant the experiments with TF-IDF disabled were 65% faster

than those with. This could perhaps be mitigated with optimization and profiling. This is also mentioned in future work (section 6.2.2). The process of calculating the TF-IDF requires one to know if any function or import has a presence in all files, which is a problem in handling data inside the application. Which means it's a practical component to it as well. We therefore cannot be certain if it's the sheer number of calculations themselves or the implementations handling of data that is the cause for these extended testing times.

From sample testing we have performed, it seems the TF-IDF struggles with larger packages, for example, the *ansible* package, which contains over 4000 Python files. Other packages take a very short time to analyze from our experience. We could ignore certain package sizes in our analysis to speed up the experiments. We could ignore larger packages in our experiments, but even though the average length of malware in our dataset is not high, it can be inside any package in theory (as presented in section 4.1.2on the malicious dataset).

The reason performance matters for us is because we want to be able to scan hundreds of packages in the shortest amount of time. This is to both (1) enable quick analysis of a package and all its dependencies, and (2) to make it feasible to scan all of PyPI as quickly as possible.

5.2 Experimental Design

We picked the *best* generation for each experiment to perform a full run. The intention of the full run was to give all experiments a common testing ground.

In the next version of our experiments, we would have performed a full run at the end of each generation, and stored it, so that we could compare all generations for all experiments more easily instead of picking the best ones and performing extra experiments on them. Another difference we would do with is to perform re-runs of experiments. This would allow us to get standard deviations from the results, which would mean we could call one result better than another with more confidence.

We also hypothesize the extra weights that are not used in the non-TF-IDF experiments have no impact on the performance, but in the future, we would change this so we could be certain. This would mean changing the machine learning setup to only use 4 or 5 genes, depending on if we have TF-IDF for both calls and functions. Removing the weights should not change the experiment time by a huge amount, as most time is spent doing the actual analysis using the implementation.

Time as a Metric of Performance

At the beginning of the experiment, we also tracked time variables for our experiments, including everything else mentioned in section 3.4 on *Metrics*. The idea was that we would see what made the implementation faster and slower. During the

project, this was deemed not feasible anymore, as the development of the implementation was across three different machines and inside virtual machines. This is because the stored data would then be inconsistent across platforms, and we realized optimizing the algorithms for the temporal aspect should be done later if the method is at all shown to work. The time perspective would be interesting to include, yet was not feasible for these experiments with the inconsistent setups.

Train/Test Splitting

In chapter 4 we present our experiments where we performed a train/test split of 80/20 (%). This is a split we have seen and thought would be reasonable. We also considered doing a K-Fold cross-validation with $K = 5$. This would mean us doing the same split of 80/20 five times, such that all samples, at some point, get to be both a test- and train-sample. The reason we choose not to use K-Fold in our experiments is that this would require perhaps five times the time it already takes to run one test. As we saw from the results from experiment 1 in table **??**, one run of 80/20-splitting with 10 generations took approximately six and a half hours. This same run with K-Fold would then take around 32 hours. At this time this would not be reasonable with respect to time. Before doing this we would perform some optimization on both the implementation, the dataset, and the experiment framework. The test framework is not multi-threaded due to I/O limitations at this moment but could be fixed in the future. This means we only run one test at a time, which is not efficient, but the time required to fix this cannot be justified before this report is due.

5.3 Limitations

Another limitation we found was that TF-IDF in some cases hides critical bulletins in malicious packages. It is used to lower false positives, so we can argue it works as expected, but this is not good when scanning a small package, or a single file. This is why we again performed tests to uncover the impact of this.

Implementing Other Languages

Out method which we presented in chapter 3 uses the fact that Python code need indentation in order to parse its syntax. This is not true for many other languages, such as JavaScript. This means this method at this moment is very specific to Python, but more research could be done to apply it to other programming languages.

Experiment Speed and Overfitting Weights

The rate at which we could perform the experiments was pretty low. This is on the same topic as also previously discussed, except this is regarding the experiments as a whole and not the specific implementation of TF-IDF. The experiments

themselves took upwards of six to eight hours, as shown in table 4.4 and 4.7. These experiments were for 10 generations, with a starting population of 8. The bottleneck was the evolution process that needed to run the weights for both the benign and malicious samples to return a fitness. This could have been multi-threaded with more time. Each gene (set of weights) in the pool for the genetic algorithm at a given generation could be run at the same time since the genes are not dependent on each other. The benign and malicious tests could also be run in parallel, but running through the benign samples is the most time-consuming part, so this last parallelization would probably make little difference.

We can tell by the experiment time on experiments *with* and *without* TF-IDF that this has a big impact on performance.

We were afraid the weights might be overfitted, meaning we trained to well on the dataset we gave them and would not be generally applicable to other datasets, which is the reason we have a training/testing split. We wanted to implement K-Fold for the dataset, so we could have even better fitness values for the genetic algorithm, but since the experiments took so long to run, this would not be feasible. We also wanted to run the experiments several times, but we had trouble running more experiments at a time at the beginning of the testing phase, which resulted in the limitation of only being able to run one test at a time. As discussed in future work; the experiment framework and the speed of the tests would be a good place to put optimization efforts if one decides to continue that route.

Tracking Bulletins

The way our experiments currently track bulletins is by counting the total amount of bulletins that are reported for all packages. This method has its drawbacks, and is currently a limitation. It is useful to have a single statistic which we can use in our fitness function, but also knowing how many packages got zero bulletins would be interesting. Looking at package counts for individual packages would not be feasible, but calculating certain descriptive statistics from those counts would be interesting. With different types of bulletin tracking, we could, for example, optimize for alerting on as few packages as possible and see what type of results this would give us.

5.4 Validity

Mistakes in the Implementation

We need the implementation to be correctly implemented in order to receive valid results. We recognize that mistakes in the implementation of the method we have developed are possible. Depending on the mistake, the results to no longer be valid and would have to be re-run.

Labeling of the Dataset

The labeling of the dataset, as described in chapter 3, is essential to accurately measuring what is malicious. In this process, one has to make assumptions and choices about *what* lines of code contribute to the malicious activity. In some cases, this can be difficult. An example is an import dealing with network functionality, which in this example is used throughout the whole program, but is also used by the malicious payload. The question is if this line where this import is loaded should be marked as malicious or not. This is an example, but we did have the same situations when labeling our dataset. Most times we ended up marking such imports as malicious, but it also depends on the malware. The difficulty is in ensuring we cover all the malicious code, while still not marking too much benign code, which would then contribute to more false positives since the machine learning algorithm follows this as malicious.

F-Score vs. Accuracy

It is important to note that this implementation is a classifier, it is a tool that provides static security analysis for Python packages and their dependencies. This implementation has the goal of trying to bring better detection of unwanted code into the hands of normal developers while trying to not make the barrier of entry too high. It also aims to be versatile while experimenting with new techniques to give feedback on suspicious and malicious code. Therefore the tests and experiments cannot be viewed as measuring a classification problem.

As mentioned in the methodology chapter; we utilize F-score to measure how well the implementation generates *bulletins* for the lines of code we have labeled. We incentivize the GA with our fitness function (described in the methodology chapter 3.2.5) to do this, while still optimizing for bulletin count. The F-score practically translates to how much of the labeled code has got a bulletin generated for it. Think of it as perhaps as how much area have we covered compared to what is desired (which is of course 100%).

At the beginning of the project, we used accuracy to measure how well we detected labeled lines of code but switched to F-score after it seemed like a good idea due to it using precision and recall. As an afterthought, this might skew the performance of the implementation because it is not a classifier. We want to measure the area of labeled code we cover, and for that accuracy may be better than F-score.

5.5 Conclusion

In conclusion, the goal of this thesis was to find a method of analyzing a package and all its dependencies efficiently, as well as being able to deploy it to scan large amounts of packages in the shortest possible time.

To do this we looked into the small, but growing, literature on supply-chain attacks, vulnerabilities in dependencies, and malicious packages in package managers. We read up on works by Duan et al.[21] who analyzed package managers NPM, PyPI, and RubyGems for malicious packages, and Ohm et al.,[40] who curated and analyzed a collection of known malicious packages in several package managers including NPM and PyPI.

We were inspired to develop a new method by building on the malware analysis research by Sand [36] and Fang et al. [19], as well as the static control-flow analysis done by Micheelsen and Thalmann [32].

This new method utilizes static analysis which involves fields, weights, and bulletins. This approach analyzes the AST of Python programs in order to extract function calls, imported modules, variables, and certain behaviors. We then match this information against a set of defined rules. When a rule is triggered it adds a risk value to the fields, on the line of code is triggered. When a risk value goes above a threshold, one or more bulletins (alerts) are returned to the developer. The developer then has to then judge whether or not the bulletins actually show malicious activity and how they would act on it.

We then performed a series of experiments to test our new method on a dataset of 441 benign packages and 97 malicious ones. Inspired by Fang et al.[19] we also added TF-IDF to lower the amount of alerting false positives. The experiments utilized genetic algorithms to optimize weights, which controlled how much the risk value mattered when comparing it to a threshold. These experiments resulted in two sets of weights which we deemed the best; the first set resulted in a 60% F-score and 222 bulletins when experimenting with our dataset of 441 benign packages and 97 malicious packages. The second set resulted in an 8-9% in F-score, but zero bulletins. We conclude the first set is more applicable for smaller to medium scale analysis, while the second set might be applicable to larger-scale analysis (of all packages in PyPI), but more research is needed.

Lastly, we open-sourced two projects from this thesis; the first project is the implementation of our new method [1], and the second is the program to generate the canaries we have previously described [46].

Chapter 6

Future Work

This chapter will focus on what we would improve in this thesis given more time and further work in the area. Since this report had a lot of focus on implementing an experimental technique this is going to be a lot of the focus and is elaborated on in the *Implementation*-section below. The topics in this chapter are as follows:

- Methodology & Experiments - *How our methodology and our experiments to test it could be improved.*
- Implementation - *How the implementation itself could be improved.*

6.1 Methodology & Experiments

As mentioned in the discussion, and later in this chapter, we wanted to optimize the experiment time. This is so we could speed up the process of gathering fitness data from our experiments. This would also enable us to perform K-Fold testing, which would hopefully help to optimize the weights for a much more general solution. We could also run the experiments several times to see if the optimization was consistent, or if it is strongly influenced by randomness.

We could also consider using more metrics for another implementation as discussed in the previous chapter. Using accuracy would be interesting to see correlated with our F-measure and other metrics as well. For this thesis, we chose F-score as previously mentioned for our metric, but this may not be the best solution. More research is needed here.

On the topic of metrics; we would also consider looking into synthesized data, as the number of real worlds samples is low. This could perhaps be improved by also utilizing synthesized data, but we could foresee this being a lot of work. Bigger datasets would hopefully enable us to have better results when using machine learning for optimization.

Turn it Into a Classifier

The implementation implemented in this thesis is not a classifier, as is discussed in chapter 5. For future work, we would be interested in experimenting with using the detection model and machine learning to create a binary classifier, which could hopefully be used for more semi-automatic flagging of malicious packages. The intention would be to run this on a large scale or to give more reliable information back to the developer on the dependencies they use.

6.2 Implementation

This section discusses future work and possible ideas for the implementation itself.

6.2.1 Big Data, Cloud, and Edge Computing

Our implementation is written in Rust, which also has the possibility to be compiled to Web Assembly (WASM). This opens the possibility to have the same command-line application run on the edge with little modification, in theory. This allows for maintaining a single code base that can be easily deployed to several operating systems, as well as the cloud.

The hope is to be able to utilize the cloud, or edge computing, to parallelize the operation of searching through packages. We would then require less local resources, and could possibly turn it into a REST-API, which would make it more feasible to use in automatic build pipelines (CI/CD).

6.2.2 Performance and Profiling

The implementation as it stands integrates all the parts it needs from the Rust ecosystem to lower the overhead of inter-process communication. This allows us to have more free control over the data we handle, and at the same time not sacrifice it for performance. The implementation does have a ways to go when it comes to performance. Profiling would be the first step in trying to optimize the existing implementation to run faster, which would allow for more packages per second.

In the same realm, we would discuss multi-threading or asynchronous operations. In Rust, there are good libraries and frameworks to implement this, and the language even has support to ensure no deadlocks will happen at compile-time, meaning there will be less time debugging deadlocks when doing multi-threading. This is also one of the reasons we chose Rust over other programming languages for the implementation in this report.

6.2.3 Caching of results

In chapter 3, our figure 3.5 shows the evaluation process after the bulletins have been placed. We believe that there is potential in perhaps caching results for up

to the *Pre-Evaluation* step. This means we could tweak variables and more quickly execute a search anew. The scenario would be searching through a large heap of packages, and want to adjust the weights to give more or fewer results, depending on what you have received back from the implementation.

This approach of caching has the downside of taking up **space**, while saving on **time**. The implementation as it stands spends more time and does not really take up any space except the packages it is analyzing.

6.2.4 Improved Rules

The rule system introduced in chapter 3 was quickly implemented and may not be the best for the task. In future work, we would explore other ways to create rules, thresholds, and test if the functionality aspect of rules as it stands today is useful in detecting malicious packages.

Doing research on what rules to include would be valuable. Better definitions that not only looked for concrete functions and imports, but could perhaps look for combinations, and chain rules togheter. Maybe even more information could be useful. The rule set implemented in this report was already inspired by the types found in intrusion detection systems (IDS), and malware signatures (e.g. YARA rules), and could perhaps borrow more concepts from these implementations and shape them for our needs.

6.2.5 Support for Different Python Versions

As mentioned in section 4.2 on the implementation, we only support newer versions of Python with the *rustpython-parser* library, and the middleware we use to transforming code to something this library understands is not a robust solution for the future. Since we already only do static analysis we do not necessarily need a correct parser for every version of Python, and would perhaps be good with a *general* parser that gave as much information as possible given any Python version. In a perfect scenario, we would have a parser for every version, but this is not feasible. Instead, we recommend trying to create a *soft parser* that is more lenient on syntax errors, either through ignoring certain lines of code, making assumptions, or others. The parser would only be used to alert the developer on malicious code, and the code would never be run, so this might be a good solution.

6.2.6 Dynamic Analysis Through a Modified Interpreter

As discussed in previous work, we would also like to combine static and dynamic analysis for even better performance. Static analysis could be susceptible to obfuscation, and we might detect code that is never even run, while dynamic analysis requires a more elaborate setup to safely analyze potential malware, but might not run the code we want to inspect due to anti-analysis techniques or system conditions.

The idea is to modify the Python interpreter in such a way that we emulate or block system calls to the file system or network, for example. This is largely similar in concept to Kim et al. (discussed in chapter 2.2.5). This could mean the malicious program would try to fetch a payload from some website, but instead, we block the request and serve it a junk payload. This technique of intercepting traffic and blocking it and serving your own payload back is not new in any way (it is often called a Man-in-the-middle (MITM) attack), Sikorski describes this as a method of malware analysis in their book [14]. The difference with this approach is to do all this in the same *program*, which would mean not running a virtual machine, on the host machine. This is the ideal scenario, which is of course not advised before thoroughly testing this modified interpreter. You would have to make sure you have hooked all the functions interacting with the host system (file system, network, system configuration, etc.). In future work we could for example fork the *RustPython* implementation and integrate it straight into our implementation. This would hopefully lower the overhead of the analysis, and at the same time give us data from dynamic analysis. This would of course be overly complicated if we wanted to only test a single malicious package, but we want to facilitate mass-processing in the shortest amount of time.

Bibliography

[1] A. Milje, *Scout: A static analysis security tool for python packages and dependencies.* https://github.com/Syntox32/scout, (Accessed on 05/30/2022).

[2] L. Dabbish, C. Stuart, J. Tsay and J. Herbsleb, 'Social coding in GitHub: Transparency and collaboration in an open software repository,' in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work - CSCW '12*, Seattle, Washington, USA: ACM Press, 2012, p. 1277, ISBN: 978-1-4503-1086-4. DOI: 10.1145/2145204.2145396. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2145204.2145396 (visited on 04/03/2022).

[3] 'NpmJS Blog - So long, and thanks for all the packages!,' [Online]. Available: https://blog.npmjs.org/post/615388323067854848/so-long-and-thanks-for-all-the-packages (visited on 20/04/2022).

[4] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma and D. Lo, 'Out of sight, out of mind? How vulnerable dependencies affect open-source projects,' *Empir Software Eng*, vol. 26, no. 4, p. 59, Jul. 2021, ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-021-09959-3. [Online]. Available: https://link.springer.com/10.1007/s10664-021-09959-3 (visited on 23/10/2021).

[5] I. Pashchenko, D.-L. Vu and F. Massacci, 'A Qualitative Study of Dependency Management and Its Security Implications,' in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, Virtual Event USA: ACM, 30th Oct. 2020, pp. 1513–1531, ISBN: 978-1-4503-7089-9. DOI: 10.1145/3372297.3417232. [Online]. Available: https://dl.acm.org/doi/10.1145/3372297.3417232 (visited on 25/11/2021).

[6] European Union Agency for Cybersecurity., *ENISA Threat Landscape for Supply Chain Attacks.* LU: Publications Office, 2021. [Online]. Available: https://data.europa.eu/doi/10.2824/168593 (visited on 06/10/2021).

[7] W. Stallings, *Network Security Essentials: Applications and Standards*, Sixth edition. Boston: Pearson, 2017, 445 pp., ISBN: 978-0-13-452733-8.

[8] 'Python Package Index (PyPI),' [Online]. Available: https://pypi.org/ (visited on 21/04/2022).

[9] 'PyPI Stats - Download stats for downloads across all packages on PyPI,' [Online]. Available: https://pypistats.org/packages/__all__ (visited on 21/04/2022).

[10] 'Python Package Index (PyPI) - Statistics,' [Online]. Available: https://pypi.org/stats/ (visited on 21/04/2022).

[11] A. Årnes, 'Digital Forensics,' p. 373, 2018.

[12] 'Python Package Index (PyPI) - Sponsors,' [Online]. Available: https://pypi.org/sponsors/ (visited on 01/12/2021).

[13] 'Homebrew - Third-party Package Manager for macOS (or Linux),' [Online]. Available: https://brew.sh/ (visited on 01/12/2021).

[14] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software*. San Francisco: No Starch Press, 2012, 766 pp., ISBN: 978-1-59327-290-6.

[15] A. Decan, T. Mens and E. Constantinou, 'On the impact of security vulnerabilities in the npm package dependency network,' in *Proceedings of the 15th International Conference on Mining Software Repositories*, Gothenburg Sweden: ACM, 28th May 2018, pp. 181–191, ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196401. [Online]. Available: https://dl.acm.org/doi/10.1145/3196398.3196401 (visited on 23/10/2021).

[16] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta and F. Massacci, 'Vulnerable open source dependencies: Counting those that matter,' in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Oulu Finland: ACM, 11th Oct. 2018, pp. 1–10, ISBN: 978-1-4503-5823-1. DOI: 10.1145/3239235.3268920. [Online]. Available: https://dl.acm.org/doi/10.1145/3239235.3268920 (visited on 24/10/2021).

[17] S. E. Ponta, H. Plate and A. Sabetta, 'Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software,' in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Madrid: IEEE, Sep. 2018, pp. 449–460, ISBN: 978-1-5386-7870-1. DOI: 10.1109/ICSME.2018.00054. [Online]. Available: https://ieeexplore.ieee.org/document/8530051/ (visited on 23/10/2021).

[18] M. Alfadel, D. E. Costa and E. Shihab, 'Empirical Analysis of Security Vulnerabilities in Python Packages,' in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Honolulu, HI, USA: IEEE, Mar. 2021, pp. 446–457, ISBN: 978-1-72819-630-5. DOI: 10.1109/SANER50967.2021.00048. [Online]. Available: https://ieeexplore.ieee.org/document/9425974/ (visited on 23/10/2021).

[19] Y. Fang, M. Xie and C. Huang, 'PBDT: Python Backdoor Detection Model Based on Combined Features,' *Security and Communication Networks*, vol. 2021, S. Nazir, Ed., pp. 1–13, 14th Sep. 2021, ISSN: 1939-0122, 1939-0114. DOI: 10.1155/2021/9923234. [Online]. Available: https://www.hindawi.com/journals/scn/2021/9923234/ (visited on 14/10/2021).

[20] N. P. Tschacher, 'Typosquatting in Programming Language Package Managers,' University of Hamburg, 2016. [Online]. Available: https://incolumitas.com/data/thesis.pdf (visited on 25/11/2021).

[21] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio and W. Lee, 'Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages,' 2nd Dec. 2020. arXiv: 2002.01139 [cs]. [Online]. Available: http://arxiv.org/abs/2002.01139 (visited on 25/11/2021).

[22] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate and A. Sabetta, 'Typosquatting and Combosquatting Attacks on the Python Ecosystem,' in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, Genoa, Italy: IEEE, Sep. 2020, pp. 509–514, ISBN: 978-1-72818-597-2. DOI: 10.1109/EuroSPW51379.2020.00074. [Online]. Available: https://ieeexplore.ieee.org/document/9229803/ (visited on 14/10/2021).

[23] 'Pyup - Safety,' [Online]. Available: https://github.com/pyupio/safety (visited on 10/05/2022).

[24] 'Requires.io,' [Online]. Available: https://requires.io/ (visited on 15/05/2022).

[25] 'Dependabot - Automated dependency updates built into GitHub,' [Online]. Available: https://github.com/dependabot (visited on 20/05/2022).

[26] 'Bandit - A tool designed to find common security issues in Python code,' [Online]. Available: https://github.com/PyCQA/bandit (visited on 20/05/2022).

[27] 'JFrog,' [Online]. Available: https://jfrog.com/ (visited on 20/05/2022).

[28] 'Snyk.io - Developer security platform,' [Online]. Available: https://snyk.io/ (visited on 20/05/2022).

[29] *Jfrog - malicious packages in pypi use stealthy exfiltration methods*, https://jfrog.com/blog/python-malware-imitates-signed-pypi-traffic-in-novel-exfiltration-technique/, (Accessed on 24/05/2022).

[30] *Jfrog - python developers are being targeted with malicious packages on pypi*, https://jfrog.com/blog/malicious-pypi-packages-stealing-credit-cards-injecting-code/, (Accessed on 24/05/2022).

[31] *Jfrog - malicious packages in pypi: 3 remote access trojans discovered*, https://jfrog.com/blog/jfrog-discloses-3-remote-access-trojans-in-pypi/, (Accessed on 24/05/2022).

[32] S. Micheelsen and B. Thalmann, 'A static analysis tool for detecting security vulnerabilities in python web applications,' Aalborg University, Aalborg University, 31st May 2016. [Online]. Available: https://projekter.aau.dk/projekter/files/239563289/final.pdf (visited on 18/10/2021).

[33] 'PyT - A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications,' [Online]. Available: https://github.com/python-security/pyt (visited on 24/05/2022).

[34] 'Pyre Check - Performant type-checking for Python.,' [Online]. Available: https://github.com/facebook/pyre-check (visited on 20/05/2022).

[35] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate and A. Sabetta, 'Towards Using Source Code Repositories to Identify Software Supply Chain Attacks,' in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, Virtual Event USA: ACM, 30th Oct. 2020, pp. 2093–2095, ISBN: 978-1-4503-7089-9. DOI: 10.1145/3372297.3420015. [Online]. Available: https://dl.acm.org/doi/10.1145/3372297.3420015 (visited on 02/10/2021).

[36] L. A. Sand, 'Information-based Dependency Matching For Behavioral Malware Analysis,' p. 138, 2012.

[37] H. C. Kim, 'JsSandbox: A Framework for Analyzing the Behavior of Malicious JavaScript Code using Internal Function Hooking,' *KSII TIIS*, 2012, ISSN: 19767277. DOI: 10.3837/tiis.2012.02.019. [Online]. Available: http://www.itiis.org/digital-library/manuscript/316 (visited on 17/01/2022).

[38] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon and J. M. Smith, 'BreakApp: Automated, Flexible Application Compartmentalization,' in *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2018, ISBN: 978-1-891562-49-5. DOI: 10.14722/ndss.2018.23131. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_08-3_Vasilakis_paper.pdf (visited on 26/11/2021).

[39] M. Ohm, A. Sykosch and M. Meier, 'Towards detection of software supply chain attacks by forensic artifacts,' in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, Virtual Event Ireland: ACM, 25th Aug. 2020, pp. 1–6, ISBN: 978-1-4503-8833-7. DOI: 10.1145/3407023.3409183. [Online]. Available: https://dl.acm.org/doi/10.1145/3407023.3409183 (visited on 25/11/2021).

[40] M. Ohm, H. Plate, A. Sykosch and M. Meier, 'Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks,' in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science, C. Maurice, L. Bilge, G. Stringhini and N. Neves, Eds., vol. 12223, Cham: Springer International Publishing, 2020, pp. 23–43, ISBN: 978-3-030-52682-5 978-3-030-52683-2. DOI: 10.1007/978-3-030-52683-2_2. [Online]. Available: http://link.springer.com/10.1007/978-3-030-52683-2_2 (visited on 08/10/2021).

[41] C. M. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. New York: Springer, 2006, 738 pp., ISBN: 978-0-387-31073-2.

[42] Wikipedia contributors, *Normal distribution — Wikipedia, the free encyclopedia*, [Online; accessed 25-May-2022], 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Normal_distribution&oldid=1087764287.

[43] Wikipedia contributors, 'Term Frequency - Inverse Document Frequency (TF-IDF) - Wikipedia,' *Wikipedia, The Free Encyclopedia*, [Online]. Available: https://en.wikipedia.org/w/index.php?title=Tf%E2%80%93idf&oldid=1071253989 (visited on 10/05/2022).

[44] H. C. Wu, R. W. P. Luk, K. F. Wong and K. L. Kwok, 'Interpreting TF-IDF term weights as making relevance decisions,' *ACM Trans. Inf. Syst.*, vol. 26, no. 3, pp. 1–37, Jun. 2008, ISSN: 1046-8188, 1558-2868. DOI: 10.1145/1361684.1361686. [Online]. Available: https://dl.acm.org/doi/10.1145/1361684.1361686 (visited on 10/05/2022).

[45] *Ron-rs/ron: Rusty object notation*, https://github.com/ron-rs/ron, (Accessed on 05/26/2022).

[46] A. Milje, *Syntox32/encodary: A script for generating rainbow tables for simple encodings and string transformations.* https://github.com/Syntox32/encodary, (Accessed on 05/30/2022).

[47] I. Kononenko and M. Kukar, *Machine Learning and Data Mining: Introduction to Principles and Algorithms*. 1st Mar. 2008.

[48] A. F. Gad, *Pygad: An intuitive genetic algorithm python library*, 2021. arXiv: 2106.06158 [cs.NE].

[49] 'GitHub - PyPInfo,' [Online]. Available: https://github.com/ofek/pypinfo (visited on 29/04/2022).

[50] *Rustpython: An open source python 3 interpreter written in rust*, https://rustpython.github.io/, (Accessed on 05/30/2022).

[51] *Ast — abstract syntax trees — python 3.10.4 documentation*, https://docs.python.org/3/library/ast.html#ast.walk, (Accessed on 05/30/2022).

Appendix A

How It's Made

For Writing and Research

- Overleaf
- Zotero
- Joplin
- Dropbox

For Programming

- Visual Studio Code
- Rust 1.61.0
- Rust-analyzer Extension
- LLDB for Rust Debugging
- Python 3.10

For Graphs and Diagrams

- excalidraw
- mermaid-js
- diagrams.net
- matplotlib

Appendix B

The Default Set of Rules

This is the default set of rules included in the implementation. This file can also be viewed in the GitHub repository for the implementation.

```
///
///
/// Rules are defined as follows:
///     ```
///     Module(Functionality, Identifier, Name (optional), Description (optional))
///     ```
/// You can choose between Module or Function.
///
/// `Name` and `Description` is currently not used and can be safely set to `None`.
///
///
([
    ( name: "Basic module rules", threshold: 0.30, rules: [
        // compression
        Module(Compression, "zlib", None, None),
        Module(Compression, "gzip", None, None),
        Module(Compression, "tarfile", None, None),
        // encoding
        Module(Encoding, "base64", None, None),
        Module(Encoding, "binascii", None, None),
        // hashing, signing, encryption
        Module(Encryption, "hashlib", None, None),
        Module(Encryption, "hashes", None, None),
        Module(Encryption, "Crypto.Util.Padding", None, None),
        Module(Encryption, "Crypto.Cipher", None, None),
        // networking
        Module(Network, "socket", None, None),
        Module(Network, "urllib2", None, None),
        Module(Network, "urllib", None, None),
        Module(Network, "urllib.request", None, None),
        Module(Network, "paramiko", None, None),
        Module(Network, "ftplib", None, None),
        Module(Network, "socketserver", None, None),
        Module(Network, "httplib", None, None),
        Module(Network, "scapy", None, None),
        // processes
        Module(Process, "subprocess", None, None),
        Module(Process, "commands", None, None),
        Module(Process, "pty", None, None),
```

```
        Module(Process, "threading", None, None),
        Module(Process, "select", None, None),
        Module(Process, "multiprocessing", None, None),
        Module(Process, "setproctitle", None, None),
        Module(Process, "shutil", None, None),
        Module(Process, "fcntl", None, None),
        // filesyten
        Module(FileSystem, "io", None, None),
        // System
        Module(System, "ctypes", None, None),
        Module(System, "platform", None, None),
        Module(System, "winreg", None, None),
        Module(System, "psutil", None, None),
        Module(System, "wmi", None, None),
        Module(System, "pynput", None, None),
        Module(System, "pwd", None, None),
        Module(System, "os", None, None),
    ]),
    ( name: "Misc. suspicious rules", threshold: 0.20, rules: [
        Module(NotSpecific, "builtins", None, None),

        Module(System, "importlib", None, None),
        Module(System, "marshal", None, None),
        Function(System, "marshal.load", None, None),
        Function(System, "marshal.loads", None, None),
        Module(System, "pytransform", None, None),
        Function(System, "pyarmor_runtime", None, None),
        Function(System, "__pyarmor__", None, None),
    ]),
    ( name: "Basic function rules", threshold: 0.20, rules: [
        // compression

        // encoding
        Function(Encoding, "b64decode", None, None),
        Function(Encoding, "b64encode", None, None),
        Function(Encoding, "EncodeAES", None, None),
        Function(Encoding, "DecodeAES", None, None),
        Function(Encoding, "encode_base64", None, None),
        // Function(Encoding, "OAEP", None, None),
        // Function(Encoding, "MGF1", None, None),

        // hashing, signing, encryption
        Function(Encryption, "encrypt", None, None),
        Function(Encryption, "decrypt", None, None),
        Function(Encryption, "AESGCM", None, None),
        Function(Encryption, "md5", None, None),
        Function(Encryption, "rc4", None, None),
        Function(Encryption, "SHA256", None, None),
        Function(Encryption, "sha1", None, None),
        // networking
        Function(Network, "urlopen", None, None),
        Function(Network, "socket", None, None),
        Function(Network, "bind", None, None),
        Function(Network, "setsockopt", None, None),
        Function(Network, "gethostbyname", None, None),
        Function(Network, "gethostname", None, None),
        Function(Network, "SSHClient", None, None),
        // processes
        Function(Process, "spawn", None, None),
        Function(Process, "Popen", None, None),
```

```
        Function(Process, "communicate", None, None),
        Function(Process, "daemon", None, None),
        Function(Process, "fork", None, None),
        Function(Process, "ThreadingTCPServer", None, None),
        Function(Process, "ThreadingUDPServer", None, None),
        Function(Process, "setproctitle", None, None),
        Function(Process, "CreateThread", None, None),
        // filesyten
        // Function(FileSystem, "open", None, None),
        // Function(FileSystem, "StringIO", None, None),
        // Function(FileSystem, "BytesIO", None, None),
        // System
        Function(System, "exec", None, None),
        Function(System, "execv", None, None),
        Function(System, "execvp", None, None),
        Function(System, "execfile", None, None),
        Function(System, "storbinary", None, None),
        Function(System, "system", None, None),
        // Function(Encoding, "getopt", None, None),
        // Function(Encoding, "getoutput", None, None),
        // Function(Encoding, "tcsetattr", None, None),
        Function(System, "command", None, None),
        Function(System, "exec_command", None, None),
        Function(System, "check_output", None, None),
        Function(System, "VirtualAlloc", None, None),
        Function(System, "sysinfo", None, None),
    ]),
])
```

Appendix C

Common Words Used in Creating Canaries

This is the full list of common words used to create canaries in our program.

```
import
__import__
base
eval
exec
base64
zlib
gzip
urllib
http://
https://
ftp://
ws://
wss://
b64decode
b32decode
b16decode
b85decode
decode
#!/usr/
#!/bin/
```

Appendix D

Genetic Algorithm Configuration

This is the full configuration we used in our genetic algorithm. The library is as mentioned PyGAD [48]. Do note that this code below is a snippet from the testing framework and is missing variables and context, but the important details of the configuration are present.

```
num_generations = [1, 5, 10]
function_inputs = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

num_parents_mating = 4

sol_per_pop = 8
num_genes = len(function_inputs)

init_range_low = 0.0
init_range_high = 2.0

parent_selection_type = "sss"
keep_parents = 1

crossover_type = "single_point"

mutation_type = "random"
mutation_percent_genes = 10

def on_generation(ga):
    if ga.generations_completed in num_generations:
        generations = ga.generations_completed
        solution, solution_fitness, _ = ga_instance.best_solution()
        total_time = time.perf_counter() - start_time

        # custom code to report results to a log file
        # also runs a test to gather F-score for generations 1, 5, and 10
        [...]

ga_instance = pygad.GA(num_generations=max(num_generations),
                num_parents_mating=num_parents_mating,
                fitness_func=fitness_func,
                sol_per_pop=sol_per_pop,
                num_genes=num_genes,
                init_range_low=init_range_low,
                init_range_high=init_range_high,
```

```
                parent_selection_type=parent_selection_type,
                keep_parents=keep_parents,
                crossover_type=crossover_type,
                mutation_type=mutation_type,
                mutation_percent_genes=mutation_percent_genes,
                on_generation=on_generation)
```

Appendix E

All Experiment Result Tables

F-score	Bulletins	Fitness
0.44213	2172	2.0356

Table E.1: The control **experiment 0** using all weights set to 1.0.

F-score	Bulletins	Fitness
0.78113	9662	0.8085

Table E.2: The control **experiment 0.1** using all weights set to 1.0 but with **TF-IDF disabled**.

Gen.	F-score	Training Fitness	Testing Fitness	Time
1	0.8301	0.7773	4.6401	14m
5	0.8802	0.7824	4.1248	38m
10	0.8601	0.8078	3.6867	1h 7m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	1.9191	1.2179	1.7402	1.008	0.9143	0.4415
5	2.1904	1.2179	1.7402	1.5566	-0.3746	1.1532
10	2.8935	1.2179	1.4287	1.5566	-0.3746	0.6519

Table E.3: The results for **experiment 0.2** which only uses the F-score from the malicious samples as output from the fitness function (TF-IDF on function calls and imports, train/test split).

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.37398	326	3.2212	11.4719	1h 26m
5	0.76800	291	7.9026	26.3918	3h 43m
10	0.66667	301	8.1081	22.0751	6h 26m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.2068	0.8546	-0.1262	0.7161	1.2623	1.4922
5	0.2068	0.8546	-0.1262	1.9047	1.2231	0.2900
10	0.2068	0.8546	-0.1262	2.0965	1.2231	0.2900

Table E.4: The results from **experiment 1** (TF-IDF on calls and imports, train/test split). The bulletin count is taken from the testing split of the dataset.

F-score	Bulletins	Fitness
0.66959	1136	5.8943

Table E.5: A run using the best weights from experiment 1.

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.45937	1191	3.8668	3.8571	1h 44m
5	0.55796	606	10.1426	9.2072	4h 41m
10	0.57813	339	17.1140	17.0540	8h 9m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.8687	0.6784	0.6106	1.0452	0.5876	0.2658
5	0.1325	0.6784	0.4750	1.0452	0.7573	0.0923
10	0.1325	0.6784	-0.5991	1.0452	0.0083	0.0923

Table E.6: Results from **experiment 2** (TF-IDF on calls and imports, all samples for training and testing).

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.50746	174	62.6133	29.1645	36m
5	0.42520	235	82.1745	18.0935	1h 23m
10	0.51852	119	84.6096	43.5730	2h 19m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.0926	0.7276	0.9004	1.1821	0.8733	0.3475
5	0.1862	0.6987	0.9004	1.1821	0.8733	0.7126
10	0.0926	0.6465	-0.0913	-0.0998	0.8733	0.0276

Table E.7: Results from **experiment 3** (TF-IDF on calls and imports, train/test split, 100 random benign packages).

F-score	Bulletins	Fitness
0.61678	222	27.7830

Table E.8: A run using the the weights from the 10th generation in experiment 3.

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.16514	117	163.5514	14.1143	34m
5	0.23009	66	216.9421	34.8619	1h 26m
10	0.21429	0	7.692e+10	2.143e+11	2h 30m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.5170	0.2858	0.3110	1.0584	0.2251	0.4617
5	-0.4558	0.5497	0.3110	0.3985	-0.012	1.5758
10	0.3601	-0.4456	0.3110	1.9159	0.1058	0.9668

Table E.9: Results from **experiment 4** (TF-IDF disabled, train/test split). The number of bulletins are from the testing split. We can discard the last two weights, as they are TF-IDF weights.

F-score	Bulletins	Fitness
0.09524	0	9.524e+10

Table E.10: Doing an all-sample run with the 10th generation weights from experiment 4.

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.13084	450	2.9076	2.9076	0h 39m
5	0.07692	64	12.0192	12.0192	1h 42m
10	0.07692	0	7.692e+10	7.692e+10	2h 56m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.0460	0.1298	1.5397	1.5371	0.0535	0.5487
5	0.0460	0.1298	0.9818	1.5371	0.0535	0.5487
10	-0.2693	0.0212	0.3311	2.0712	0.0535	0.5487

Table E.11: Results from **experiment 5** (TF-IDF disabled, all samples). We can discard the last two weights, as they are TF-IDF weights.

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.3871	156.0	3017.1428	24.8139	0h 17m
5	0.0	141.0	9.524e+10	0.0	0h 41m
10	0.16514	141.0	1.308e+11	11.7119	1h 13m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.5337	0.5787	0.1008	1.6059	0.4299	0.8532
5	0.5337	-0.3201	0.1008	0.7157	0.9821	0.6399
10	0.5513	-0.3596	0.2515	1.0618	0.9821	1.6551

Table E.12: Results from **experiment 6** (TF-IDF disabled, train/test split, 100 random benign packages). We can discard the last two weights, as they are TF-IDF weights.

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.21429	154.0	4.914	13.9147	1h 14m
5	0.65772	122.0	15.8576	53.9113	3h 15m
10	0.65772	90.0	23.3333	73.0798	5h 37m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.4785	0.4162	0.4643	1.5280	0.9669	0.2802
5	0.0112	0.6627	0.4643	1.7647	0.9669	0.2802
10	0.0112	0.6535	-0.4587	1.7647	0.9669	0.2802

Table E.13: Results from **experiment 1 - Import TF-IDF disabled** (train/test split).

F-score	Bulletins	Fitness
0.63531	230	27.62219

Table E.14: All samples run with weights from *experiment 1 - Import TF-IDF disabled*.

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.4	11.0	13.0378	363.6364	1h 49m
5	0.62069	33.0	16.1588	188.0878	4h 1m
10	0.35204	9.0	21.8659	391.1565	6h 40m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.3160	1.0592	0.2348	1.8756	1.1560	1.0869
5	0.3160	1.2579	-0.4985	1.8756	1.1560	1.0869
10	1.4071	0.6339	0.4356	1.8756	1.1560	1.0869

Table E.15: Results from **experiment 1 - changing IDF to DF** (TF-DF for calls and imports, train/test split).

F-score	Bulletins	Fitness
0.75484	8663	0.871336

Table E.16: All samples run with weights from *experiment 1 - changing IDF to DF*.

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.22781	65.0	35.0476	35.0476	1h 53m
5	0.07692	4.0	256.4103	192.3077	4h 56m
10	0.07692	0.0	7.692e+10	7.692e+10	8h 22m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	1.6151	0.5356	0.4187	1.4436	0.9394	1.0727
5	-0.0194	-0.2654	0.3103	1.4436	0.9394	1.0727
10	-0.0194	0.2300	0.3103	1.4436	0.9394	-0.000

Table E.17: Same as **Experiment 2 - changing IDF to DF** (all samples, TF-DF on imports and calls)

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.24561	8.0	3.922e+10	307.0175	0h 33m
5	0.24561	8.0	3.922e+10	307.0175	1h 17m
10	0.24561	13.0	3.922e+10	188.9339	2h 1m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.5701	0.9760	0.7011	0.6455	1.5932	1.6009
5	0.5701	0.9760	0.7011	0.6455	1.5932	1.6009
10	0.5701	1.1152	0.7011	0.1127	1.5932	1.6009

Table E.18: Same as **Experiment 3 - changing IDF to DF** (RND100 Train/Test, TF-DF on imports and calls)

F-score	Bulletins	Fitness
0.05825	51	11.42204

Table E.19: All samples with weights from GA 1 and 5 in **Experiment 3 - changing IDF to DF**

Gen.	F-score	Bulletins	Training Fitness	Testing Fitness	Time
1	0.73084	662.0	4.1544	11.0399	1h 20m
5	0.21429	0.0	3.922e+10	2.143e+11	3h 27m
10	0.35204	0.0	3.922e+10	3.520e+11	6h 1m

Gen.	FW Func.	FW Im.	FW Behavior	FW Str.	TW Func.	TW Im.
1	0.2986	0.8362	0.3634	1.4839	1.7156	0.6493
5	0.4354	0.1228	0.3634	2.0729	1.7156	0.6493
10	0.7177	-0.0322	0.3634	2.5269	1.7156	1.4042

Table E.20: Results from **experiment 1 - TF-DF disabled for imports** (train/test split).

F-score	Bulletins	Fitness
0.11182	0	1.118e+11

Table E.21: All samples run with weights from *experiment 1 - TF-DF* (disabled for imports).

Anders Aasrum Milje

Detecting Malicious Python Packages in PyPI

NTNU
Kunnskap for en bedre verden