UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Effective Code Obfuscation for Python

Robin Acke

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Peter QUAX

BEGELEIDER :
De heer Wouter LEMOINE

2022
2023

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Effective Code Obfuscation for Python

Robin Acke

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

BEGELEIDER :

De heer Wouter LEMOINE

Universiteit Hasselt

Masterproef voorgedragen tot het behalen van de graad van master in de informatica

Effective Code Obfuscation for Python

Auteur:

Acke Robin

Promotor:

Prof. dr. Peter Quax

Begeleider(s):

Wouter Lemoine

Academiejaar 2022-2023

Abstract

Code obfuscation is a technique that can be used to complicate the process of analyzing and modifying programs. It transforms a program in a way that maintains its functionality while making it harder to reverse engineer. There are numerous tools and publications for obfuscating C(++) and Java programs, but practically none for Python. This thesis explores the feasibility of obfuscating Python programs. It also looks into possible evaluation methodologies that can quantify the impact of obfuscation techniques for Python. Four obfuscation techniques have been implemented in a proof-of-concept obfuscator for Python. This implementation has been evaluated using a set of chosen metrics that are commonly used to evaluate obfuscation techniques. The results indicate that all of these techniques can be effective for Python, with varying impact on program performance.

Acknowledgements

Writing a thesis is a unique challenge that would not have been possible without the constant support of the people around me.

I would first and foremost like to thank my mentor Wouter Lemoine and my promotor prof. dr. Peter Quax for providing me with this opportunity and helping me every step of the way. The continuous monitoring and guidance provided to me has been invaluable in pushing me and my thesis towards success.

Secondly I would like to thank prof. dr. Peter Quax (again) together with prof. dr. Wim Lamotte and the rest of the research group Networked and Secured Systems. From hosting the UHCTF to organizing engaging classes and projects every semester, all these experiences have indirectly contributed towards the completion of this thesis.

Lastly I would like to thank my friends and close family for their endless support and encouragement.

Contents

Chapter 1

Introduction

1.1 Context

Python[1] is currently one of the most popular programming languages. With a relatively simple syntax and a wide variety of libraries, it can be used as a tool for any kind of software development. By design, the language is intended to be used with an interpreter, a program that directly executes the source code by translating it to CPU instructions on the go. This means that typically an interpreter and the full source code of a program have to be available for a machine to run a program written in Python.

A problem arises when a developer does not want their Python code to be (easily) readable or modifiable by the users of the software. Python is not built around keeping this source code inaccessible or protected in any way. Converting the source code to bytecode is the only native[2] way to run Python code without directly distributing the source code.

Bytecode is an intermediate format that executes in the same way as the original source code. Chapter 3 explores the conversion of regular source code to *Python Bytecode* and the potential benefits of this conversion. However, this conversion can be reversed and it therefore does not allow programmers to keep their source code truly hidden.

Instead of relying on bytecode conversion, this thesis explores the use of *Code Obfuscation* to hide the original source code of Python programs. Code obfuscation (formally defined in Chapter 4) is an umbrella term for techniques that aid developers to protect their software and their source code. This protection happens through modifications of the source code. These modifications makes the program execute in the same way while making it harder to analyze the source code.

Python is very uncommon in academic research surrounding code obfuscation, as most authors focus on languages such as Java and C++. This can partially be explained by the fact that Python is typically used as a utility scripting language and is less popular for designing end-user development. The historical trends of using Java and C++ for research also make it a less obvious choice for new research, since there is less existing reference material tailored to Python specifically.

However, there is active research into obfuscation for languages similar to Python such as JavaScript, another popular interpreted language. This research can be used as a point of reference to analyze the transformations and their effectiveness on Python. Parts of code obfuscation research for other languages can possibly also be applied to Python.

[1] `https://www.python.org/`

[2] Native in this context refers to using the reference implementation of Python, CPython. This is the most used implementation, which is provided by the developers of Python.

1.2 Research Questions

With the previously explained context in mind, this thesis attempts to answer the following research questions:

- Obfuscation techniques for Python
    1. Which modern obfuscation techniques exist that can be applied to Python while achieving their intended purpose?
- Evaluation of relevant techniques
    2. Can typical metrics and evaluation of obfuscation techniques be applied to Python?
    3. Do obfuscation techniques applicable to Python provide provable protection against attacks that intend to analyse or modify the source code?

The first research question takes a broad look into obfuscation techniques and applying them to Python. The specification of "achieving their intended purpose" allows for a deeper discussion on potential differences for Python compared to when the technique is used for a different programming language. For example, it is possible that a certain technique has no positive effect because it relies on static typing, whereas Python is a dynamically typed language.

The second question looks into what type of metrics can be used for Python, which builds on the insights gained while answering the first question. It looks into what methods are typically used to quantify and evaluate obfuscation techniques, and whether this is possible to apply to Python.

The first question will be answered with some sort of list of obfuscation techniques. The second question will provide insights into what evaluation techniques can be used for Python. The third questions combines these to answers, for which the evaluation techniques can be applied to a list of obfuscation techniques.

Besides a theoretical analysis of these techniques and metrics, a proof-of-concept Python obfuscator is created. This implementation contains some of the techniques that are discussed in the context of the first research question. This implementation provides an accurate example of a Python obfuscator.

To answer the second research question, some metrics or other evaluation methodologies that can be applied for Python will be explored. These metrics can be applied to the source code that is obfuscated using the implemented obfuscator. This allows the third research question to be tested in practice by using obfuscated source code that is the result of an actual Python obfuscator.

1.3 Motivation

Python provides an interesting target to research for code obfuscation. It is a dynamically typed, interpreted language. This might bring new challenges when compared to other languages such as Java (which has no dynamic typing) and C++ (which has no dynamic typing and is compiled to a native binary).

It is non-trivial[3] to reconstruct the source code of a natively compiled program, even without obfuscation. For Python, a program that is not obfuscated is likely directly understandable and modifiable when the original source code is available. Even when the program is converted to Python bytecode, the decompilation of this bytecode is possible and only comments and

[3]The compilation and linking process that typically takes place when building a program binary discards information. It is not easy nor guaranteed that the original source code can be correctly reconstructed using only the binary.

irrelevant formatting are lost. This means that Python programs are easy to analyze, which only makes it more important to find techniques that can protect the source code.

As mentioned before, Python is not commonly found within academic literature for code obfuscation, which mostly favours Java and C++, as well as some JavaScript (for browser related malware and similar use-cases). A lot of high level programming constructs are shared between Python and these other languages, but each language still has defining features that are not shared with the other languages. These differences require the existing research to be adapted to Python. This thesis can provide some clarity to which degree this adaptation is possible and useful.

1.4 Thesis Outline

Chapter 2 contains an introduction into software protection and techniques related to obfuscation, including watermarking and tamper-proofing. It provides the necessary background information to properly understand the following chapters without requiring any prior knowledge.

Chapter 3 covers Python bytecode and the potential of using bytecode for obfuscating Python programs. It also covers the decompilation of bytecode (converting bytecode back to regular source code). Python bytecode and the decompilation process are relevant for answering the first research question.

Chapter 4 takes a practical look into existing obfuscation techniques and applies some techniques to Python. An implementation of a few selected techniques is created and discussed. This chapter combined with chapter 3 provides an answer to the first research question.

Chapter 5 aims to evaluate obfuscation techniques. It explores existing metrics for obfuscation transformations and creates a framework to evaluate obfuscation transformations for Python. This framework answers the second research question, as it contains a usable methodology to evaluate Python obfuscation based on existing metrics. This framework is applied to the implementation that Chapter 4 discusses. The results of applying this framework are used to answer the third research question.

Lastly, chapter 6 contains the conclusion in which the research questions are explicitly answered and a person reflection on this thesis is provided. It also suggest some ideas for future research.

Chapter 2

Software Protection

Code obfuscation, the main topic of this thesis, is a type of software protection strategy. This chapter discusses software protection using specific techniques such as watermarking and tamper-proofing. These techniques can be combined with code obfuscation to improve the strength of the protection, which makes them relevant knowledge for researching code obfuscation. Code obfuscation itself is not the focus of this chapter, it is explored in more detail in chapter 4 instead.

The different code execution approaches and the reverse engineering of software are also discussed in this chapter. These elements are closely related to the software protection strategies, since they impact the way in which these strategies should be used. This means that this chapter mostly provides the broader scope in which code obfuscation is situated, with some relevant knowledge that can make it easier to understand the rest of this thesis.

[Collberg and Thomborson, 2002] discuss the three general types of defenses for software protection:

- **Watermarking**: Marking software to make it possible to determine the origin of a specific copy.
- **Tamper-proofing**: Ensuring that the software cannot be ran when modified.
- **Obfuscation**: Turning software unintelligible while keeping its functionality.

Each of these techniques can help to prevent piracy or deter users from illegally modifying their copy of a piece of software. “Software piracy is the copying and/or distribution of software for personal and/or business use without the authorization of the copyright holder”, and it costs the information technology billions of euros every year. [Wang et al., 2017] This provides a clear incentive for software developers to use software protection techniques that can prevent piracy.

Outside of direct financial gain by preventing piracy, other common motivations for protecting software include protecting proprietary algorithms and techniques, as well as strengthening the security of a system by making it more difficult to analyze for an unauthorized party.

Depending on the context, certain techniques may negatively affect the software’s execution. For example, a program that is heavily obfuscated can run slower due to extra instructions being added, or a tamper-proofed piece of software that triggers a false positive (an unmodified copy being flagged as modified) will not execute under conditions where it should. Developers should weigh the benefits and potential drawbacks before applying these techniques.

Figure 2.1: Normal image (left) and watermarked image (right)

2.1 Watermarking

Watermarking [Collberg and Thomborson, 2002] is the act of embedding a (secret) message into this cover message. This allows the original creator of the cover message to prove ownership. A cover message can be any type of content or data that the owner or creator whishes to protect.

Fingerprinting embeds a unique watermark in every distributed version of a cover message. This can be used to trace the origin of every copy of a cover message, which discourages theft or illegal distribution.

Figure 2.1[1] shows an example of a visual watermark added to a digital image. In this example, the original image has been watermark with the logo of the copyright holder. This ensures that the image cannot (easily) be stolen, as the watermark will remain visible and is hard to remove.

In this case the watermark is used to distort the original image in such a way that it cannot be used as is, this allows the original creator to distribute the watermarked image, for example for advertising purposes, without having to distribute the original image. Users can then acquire the appropriate license to get access to the non-watermarked version.

A different approach could be adding a smaller, semi-transparent watermark. This distorts the original less, meaning that the image can be used directly. This can be done to add copyright information for example, when the publisher of this image has the rights to use it but use by another party is protected by copyright.

The same principle can be applied to software. The original software will be modified with a watermark to either mark the original copyright holder, or a unique fingerprint can be added to every copy to be able to trace its origin (by using information such as vendor and unique id). This watermark or fingerprint may be added to the software in such a way that the person buying the software is not even aware of its existence, as it does not have to be physically visible to be present.

Software watermarking has the following notable properties:

- **Resilience**: The amount of effort required to dewatermark (remove the watermark)
- **Data rate**: The amount of data required to store the watermark
- **Cost**: The impact on software performance (usually minimal)

[1] Image source: `https://www.teepublic.com/blog/announcing-watermarking`, posted on september 28, 2018

- **Stealth**: The impact on the statistical properties[2] of the program

There is a trade-off between these properties. For example, the resilience of a watermark can be improved by adding multiple watermarks at several places within the software, but the data rate will increase as well. It is expected that any software watermark can be analyzed and removed with a manual attack given enough time and effort, even if the watermark has a strong resilience. The main intent of watermarking techniques is thus to counteract automatic watermark removal tools and make it non-trivial for manual attacks to be effective.

There are two types of software watermarks: *static* and *dynamic*. These differ in the way the watermark is embedded into the software, but have the same end-goal of embedding a secret message in the software.

Static watermarks are made by directly inserting the watermark in the data of the software. The program does not have to be executed for the watermark to be observed.

The exact implementation of static watermarks varies. A simple approach could be to insert a few bytes of static data that encode copyright information or a fingerprint. This can be inserted in the data section of the software. There are also a lot more complex solution, such a reconstructing the entire control flow graph of a program. This reconstructed version of the program keeps the same execution but a certain pattern can be encoded in this control flow graph.[3]

A watermark is considered *dynamic* when it can only be observed by executing the software. The watermark is generated at runtime and stored in the state of an object (or data structure). This means that instead of inserting a watermark in the data section, replacing the control flow, or any similar modification that alters the software, a piece of code is added that will generate the watermark while the software is running.

Both static and dynamic watermarking are vulnerable to dewatermarking attacks, which have the intention to render the watermark useless. For example, a potential way to locate fingerprinting information is by obtaining different copies of the same software and compare which parts differ. Depending on how the fingerprint is added this will allow the attacker to easily see where the fingerprint is located and how it can be modified to make it unusable.

[Myles and Collberg, 2006] analyses a watermarking technique using Opaque Predicates (which will get explored more in chapter 4). Opaque Predicates are a technique used for code obfuscation, and they can also serve the purpose of a watermark as they do add some extra data to a piece of software.

[Huang et al., 2023] have recently used this technique for obfuscating and watermarking blockchain smart contracts. This makes it clear that watermarking is still relevant and being used for modern technologies. The potential combination of watermarking and obfuscation make it an interesting technique to be aware off.

2.2 Tamper-proofing

Tamper-proofing [Collberg and Thomborson, 2002] adds a mechanism to software that can detect when the software has been modified. This causes the software to no longer execute (or at least not execute as intended). The goal of this technique is to prevent any illegal modifications to a piece of software, as a modified version would be unable to execute.

Tamper-proofing is something that can be found in the physical world too. For example, a lot of medication has induction seals applied to them by their manufacturer. These have to be

[2]The source material does not explain what statistical properties can be impacted. It likely refers to properties such as code complexity, structural patterns, and other high level program constructs.

[3]A full definition of control flow graphs can be found at section 4.1.1.

broken by the user when they want to use the medication for the first time, and indicates to the user that their medication has most likely not been tampered with.

When software is watermarked and tamper-proofed, any modifications to the watermark will automatically result in the software not executing. This layering of techniques means that an attacker has to both defeat the watermarking and tamper-proofing mechanism before they can illegally distribute their copy. However, this does bring the risk that due to a bug or developer mistake a user triggers this mechanism on their legitimate copy.

An example of a simple check to detect tampering is by applying a hash-function to the entire program and check if the resulting hash matches the hash that the developers obtained when releasing the software. This would require the anti-tampering mechanism to be separated from the software itself, and this check has to be protected in a way that makes it difficult to modify or bypass the check. Otherwise an attacker can simply patch in the modified version's hash or modify the check to always return a valid result even if the expected hash does not match the computed version.

In general, the succes of a tamper proofing mechanism relies on the mechanism itself being difficult to analyze and modify. If a simple attack such as patching the hash can immediately be recognized and applied the tamper proofing becomes virtually useless.

More complex mechanisms usually add extra steps, such as checking multiple results during execution, but in essence boil down to the same principle of checking whether a certain conditions is met during execution, which will break when the software is modified incorrectly. Another option is decrypting the software during execution, which will fail when the software is modified as the decryption procedure will generate incorrect results. This, however, only moves the problem from the software itself to the decryption routine, which then has to be protected using anti-reversing techniques (such as obfuscation or using hardware solutions).

These mechanisms can also be used to allow users of software to verify the integrity of their copy by having the developers generate a signature and publish the signature of their unmodified copy using public-private key encryption (meaning the developers use their private key to sign the software). Users that downloaded the software can then generate this signature on their copy using the public key of the developers and compare them to make sure they are using the original software. SigStore [Newman et al., 2022] is an example of a modern solution that allows easy signing of software using this flow.

Lastly, obfuscation itself can also serve as a tamper-proofing mechanism. Depending on the techniques used, it becomes a lot harder to modify a piece of software without breaking it. The software will first have to be obfuscated (which modifies the program without altering the semantic execution) and then the tamper-proofing mechanism has to be added afterwards, as obfuscation modifies software in a way that would most likely get flagged as illegally modified, triggering the anti-tampering mechanism.

2.3 Code Execution

There are different ways in which programs are generated and executed. This section discusses the three ways in which programs are executed, as well as some typical characteristics of each approach. The following three methods can be used to execute a program:

- **Native Compilation**: A compiler generates an executable binary for a specific target processor and operating system.
- **Bytecode Compilation**: A compiler generates bytecode, a type of instructions that has to be interpreted by a bytecode interpreter (typically a virtual machine) that uses these instructions to generate the processor instructions at runtime.

- **Interpretation**: The code can directly be ran as is by an interpreter, a piece of software that reads and executes the source code one line at a time.

These different approaches all have their advantages and disadvantages in different areas. It is important for software protection to understand these core differences to make an informed decision on what type of programming language and approach is optimal.

2.3.1 Native Compilation

With *Native Compilation*, the source code of a program is used to generate an executable binary for a specific processor (and operating system). This binary consists of the required data and instructions to execute the program. No more translation or generation steps have to be done during the execution, as the instructions can be directly executed on the processor.

This process of compiling source code is “lossy”. Source code is typically created in a human readable format and intended to be modifiable. A binary after compilation is no longer easily readable or modifiable. The original source code cannot be retrieved using the binary (although in 2.4 techniques that attempt to do this will be discussed).

The output of this approach is the hardest to analyzed compared to the other discussed approaches. It therefore can be reasoned that this provides some extra protection for the software. It does have certain drawbacks, such as the fact that a binary is compiled for only a specific operating system and processor type (e.g. ARM vs x86).

LLVM (Low Level Virtual Machine) originally designed by [Lattner and Adve, 2004] is an example of a set of compiler(-related) tools that can do native compilation. In practice, a compiler is often split in several components. Splitting the process into different steps allows for better support for multiple programming languages and multiple compilation targets by having an intermediate language, which is a representation of the code in a shared language.

This means that a “front-end” compiler first compiles the source code into this intermediate language, and the “back-end” compiler compiles this into native code for the specified target. The end result is a piece of source code that has been built into a native binary. A lot of these compilation aspects are discussed in the compilers ´´Dragon book” by [Aho et al., 2006].

2.3.2 Bytecode Compilation

Bytecode Compilation is similar to native compilation, but the target is not a native binary for a certain processor and operating system. Instead, it is an intermediate form that is not platform-specific but already is more optimized for execution than the original source code, allowing an interpreter to use this intermediate form to generate the instructions needed to execute the program.

Typically, a so-called Virtual Machine is used to execute bytecode, which translates the bytecode to native instructions and does all the interacting with the processor. It sets up a virtual environment and does anything else that is needed to succesfully execute the bytecode.

The performance of this approach is usually improved by using Just-In-Time (JIT) compilation. JIT compilation optimizes performance by analyzing which parts of the code are executed frequently and caches these parts once they have been translate to native instructions. These so-called “hot paths” can be directly executed without the extra translation overhead the next time they are encountered. [Kulkarni, 2011] have researched and found that correctly tuning these JIT compilers can have a significant impact on the performance of software being executed using bytecode, for example by using aggressive policies on multi-core machines.

Because the source code is only converted into an intermediate format instead of a native binary, it is easier to analyze and modify software that executes using bytecode. The fact that all class and function names are kept in the intermediate format makes it easy to analyze which classes and functions are intended to achieve a certain goal. This can possibly be counteracted by obfuscation techniques, like name obfuscation (which gets explored in chapter 4). It does have the benefit of being platform-independent while still being relatively fast, especially when using JIT compilation.

2.3.3 Interpretation

For *interpretation*, a piece of source code can be executed directly by the interpreter without requiring any compilation or similar process. The interpreter reads the file containing the source code line by line, parses it and then starts executing the parsed file if no errors are triggered.

This thesis focuses on Python, which is typically used as an interpreted language.[4] Some interactions between a computer and an interpreter are quite different than how a native binary would execute. This makes it an interesting research topic to explore within the context of code obfuscation.

Because no translation of the source code takes place ahead of time, all translations have to happen real-time, which will almost always be slower than a native binary (or a VM using a JIT compiler). It is however possible to execute an interpreted program with a JIT compiler just like how a JIT compiler can be used for bytecode. For example, PyPy [5], is a Python implementation that is able to interprete Python code with a JIT Compiler, which on average allows it to be significantly faster than the native C Implementation (CPython).

The source code of the software has to be distributed to the end-user in some form when an interpreted programming language is being used. This user is then able to directly analyze and modify the source code. This might be an undesired situation depending on the goals of the developer. In that case, steps have to be taken prevent the user from doing so. This can be achieved by switching the code execution approach (such as using a native binary), or by obfuscating the source code.

When a piece of software is written in an interpreted language, the user of the software needs to have a valid interpreter present on their system. For example, to execute Python code, the correct Python version has to be installed. There are however solutions for this where the user does not have to install the interpreter, but where the interpreter and source code are packed together into a single binary, such as PyInstaller [6] for Python.

2.3.4 Source-to-Source compilation

With *Source-to-Source Compilation*, it is possible to compile a piece of source code written in a certain programming language to a semantically equivalent piece of source code in a different programming language. This follows the same flow as a normal compiler, but instead of the target being a native executable or bytecode, it is a different (high level) programming language. This is also known as transpiling.

In practice, transpiling is mostly done when migrating legacy systems to modern architectures and programming languages. For example [Schnappinger and Streit, 2021] designed their own custom transpiler to aid with the conversion of legacy code. These techniques can also be used to transpile between modern programming languages, although this is not always trivially possible to full automate.

For example, some languagues have techniques that are not present in others. Java has reflection while C++ does not. Transpiling any code with reflection to C++ is therefore difficult to do and would likely require some manual intervention to obtain the desired results.

[4] This thesis covers Python code executed with the default implementation, CPython. There are environments such as Cython [Behnel et al., 2011] that can compile Python to C code, which can be natively compiled.

[5] `https://www.pypy.org/`

[6] `https://pyinstaller.org/en/stable/`

Source-to-source compilation can be used to turn a piece of code that has to be executed with a certain approach into a different language that has the option to execute differently. For example, a Python program that gets transpiled into C source code changes from an interpreted program into a native binary. This could be used to obfuscate the program using obfuscation techniques specifically for C code.

However, this just moves the problem from protecting the interpreted language to protecting the compiled language. If an attacker knows an effective method of analyzing programs of the compiled language, they would then also be able to analyze the original program while this program could potentially be obfuscated differently in its original language. This thesis does not look further into this approach, and instead focuses on obfuscating interpreted code that is executed by the intended interpreter.

2.4 Reverse Engineering

Reverse engineering is closely related to software protection. If software protection is seen in the context of a software developer defending their code and software, reverse engineering is the process that attacks the software. It therefore makes sense to also incorporate this process in a discussion about software protection.

[Chikofsky and Cross, 1990] define Reverse Engineering as the process of analyzing a subject system to

1. identify the system's components and their interrelationships and
2. create representations of the system in another form or at a higher level of abstraction.

This definition can be applied to both hardware and software, though within the scope of this chapter it will be focused solely on software. Reverse engineering software can be split further into *static* and *dynamic* analysis.

During *static* reverse engineering, the software that is to be analyzed is not executed, but further insight in its functionality is gained by analyzing the software and all the included files that are required for the software to work. This typically focuses on analyzing any executable program that comes with the software, but other elements such as included libraries or assets can also be analyzed.

Dynamic reverse engineering [Schrittwieser and Katzenbeisser, 2011] observes the software during execution, which can give further insights to its behavior when running under certain conditions. Different kinds of interactions can be observed and the behavior over time can be analyzed, such as which parts of the instructions get executed a lot.

2.4.1 Static Analysis

To analyze executable code, disassemblers and decompilers are frequently used. The following definitions will be used for these tools:

- **Disassemblers** translate machine code to Assembly.
- **Decompilers** attempt to reconstruct the original source code of a program. Information is lost during compilation, so the exact source code can (usually) not be recreated.

Listing 1 contains a snippet of C code, showing a function that does some basic arithmetic operations on three variables. Listing 2 show the assembly code that a disassembler (Godbolt [7]) generates based on this function. Even this simple three line function containing only arithmetic operations generates notably more (14) assembly instructions.

[7] `https://godbolt.org/`

```
void calc() {
    int a = 2;
    int b = 4;
    float c = 3 * a + b / 2.5;
}
```

Listing 1: Simple function calc() doing some math

```
.LCPI0_0:
        .quad   4612811918334230528     # double 2.5
calc():                                # @calc()
.Lfunc_begin0:
        push    rbp
        mov     rbp, rsp
        movsd   xmm0, qword ptr [rip + .LCPI0_0] # xmm0 = mem[0],zero
.Ltmp0:
        mov     dword ptr [rbp - 4], 2
        mov     dword ptr [rbp - 8], 4
        imul    eax, dword ptr [rbp - 4], 3
        cvtsi2sd        xmm1, eax
        cvtsi2sd        xmm2, dword ptr [rbp - 8]
        divsd   xmm2, xmm0
        addsd   xmm1, xmm2
        cvtsd2ss        xmm0, xmm1
        movss   dword ptr [rbp - 12], xmm0
        pop     rbp
        ret
```

Listing 2: Disassembly (x86-64) of calc() by GodBolt Compiler Explorer

Although all necessary information to attempt to reconstruct the source code of a program is present when using a dissassembler, a decompiler can make this process easier by attempting to reconstruct parts of the program into source code. Listing 3 contains an implementation of Selection Sort in C in the function *selectionSort()* and another function *decompiledSelectionSort()*, a reconstruction of that same function using Ghidra's decompiler [8].

The original source code was built using Clang v10.0.0-4ubuntu1 with default settings. The decompilation was done using Ghidra 10.2.3 (20230208), having specified the used compiler and using the default analysis options. In practice the used compiler and settings might not be known ahead of time, requiring some experimentation and educated guesses based on common patterns that different compilers have.

When comparing the original source code and decompiled version in Listing 3, we can see a clear similarity in the high level structures. The nested for loops are reconstructed (although the inner loop is replaced with an equivalent while loop) including the if statement with a less than condition. The variable names are lost on compilation, and the function parameters are not automatically recognized. All instructions that rely on data accessed through pointers have been reconstructed in a less intuitive way which adds a lot of cluttered pointer notations.

The decompiled code from Listing 3 was given as a prompt to ChatGPT (version March 2023), asking whether it could recognize the program and reconstruct it as "a clean C program".

[8] `https://ghidra-sre.org/`

The result of this reconstruction is shown in Listing 4, in which the algorithm was recognized as selection sort and ChatGPT provided a readable reconstruction which closely matches the original source code. When provided with just the dissassembly of the program, it could not recognize the algorithm.

Using AI and Large Language Models for software analysis and might be the next big evolution within the field of reverse engineering, but at the time of writing this thesis no in-depth research or results into these techniques have been published. [Liu and Wang, 2020] have tested a selection of modern decompilation tools to analyse the viability and correctness of C decompilers. This research gives an optimistic view on the current state of tools and techniques used for decompilation, even before this type of AI was being considered for reverse engineering. Combining these existing tools with AI is beyond the scope of this thesis, but this idea can be used for potential future research.

2.4.2 Dynamic Analysis

Dynamic analysis qualifies as any type of analysis done while a piece of software is being executed. The following types of tools are commonly used for this purpose:

- **Memory Scanners** allow the memory space of a program to be explored and possibly modified during execution.
- **Packet Sniffers** such as Wireshark [9] can capture all incoming and outgoing network traffic of a program in execution
- **Debuggers** such as x64dbg [10] allows debugging a binary of which the source code is not available.

During static analysis, interesting memory regions, functions and possible network traffic can be discovered and marked. Afterwards, these dynamic analysis tools can be used to observe the interesting sections. These tools are not necessarily made with reverse engineering as their main purpose, as observing memory regions and network traffic can provide useful insights and information for a developer working on their own software.

[9] `https://www.wireshark.org/`
[10] `https://x64dbg.com/`

```c
void selectionSort(int *array, int arrayLength) {
    for (int i = 0; i < arrayLength - 1; i++) {

        int smallestIndex = i;  // index of the smallest element so far

        for (int j = i + 1; j < arrayLength; j++) {
            if (array[j] < array[smallestIndex]) {
                smallestIndex = j;
            }
        }

        int temp = array[i];
        array[i] = array[smallestIndex];
        array[smallestIndex] = temp;
    }
}

void decompiledSelectionSort(void)

{
  undefined4 uVar1;
  int unaff_ESI;
  longlong unaff_RDI;
  int local_20;
  int local_1c;
  int local_18;

  for (local_18 = 0; local_18 < unaff_ESI + -1; local_18 = local_18 + 1) {
    local_1c = local_18;
    local_20 = local_18;
    while (local_20 = local_20 + 1, local_20 < unaff_ESI) {
      if (*(int *)(unaff_RDI + (longlong)local_20 * 4) <
          *(int *)(unaff_RDI + (longlong)local_1c * 4)) {
        local_1c = local_20;
      }
    }
    uVar1 = *(undefined4 *)(unaff_RDI + (longlong)local_18 * 4);
    *(undefined4 *)(unaff_RDI + (longlong)local_18 * 4) =
         *(undefined4 *)(unaff_RDI + (longlong)local_1c * 4);
    *(undefined4 *)(unaff_RDI + (longlong)local_1c * 4) = uVar1;
  }
  return;
}
```

Listing 3: Selection Sort in C, decompiled version by Ghidra

```
void selection_sort(int arr[], int n) {
    int i, j, min_idx, temp;
    for (i = 0; i < n - 1; i++) {
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}
```

Listing 4: Selection Sort ChatGPT reconstruction

Chapter 3

Bytecode and Decompilation

Bytecode is an intermediate format that can be used by interpreters to generate the instructions needed to execute a program. It is compiled from the original source code of a program. The original source code is no longer needed to execute the program when using bytecode. Bytecode is briefly introduced in chapter 2 (section 2.3.2), where it gets compared to natively compiled code and fully interpreted code.

Python can also be converted into *Python Bytecode*, which is touched upon in the introduction of this thesis (1.1). It gets mentioned as a simple method to help hiding the original source code of a Python program. Bytecode carries most of the contents of the original source code while discarding some unneeded information. This provides an interesting look into the interpretation process, and helps to understand the code execution process. It is therefore relevant within the context of this thesis to explore this aspect of Python.

This chapter looks into the general uses of bytecode both for Python and other programming languages such as Java. The specific application and some technical details of Python Bytecode get explored. The process of generating bytecode is not a one-way transformation, so the reverse process (*Decompilation* of bytecode) for Python gets addressed. The process of freezing Python programs gets explored as well.

3.1 Context & Python Bytecode

Java is historically the most popular programming language that heavily relies on bytecode. A java program is compiled into bytecode, a lower level intermediate representation which is no longer stored in a human readable format. It can be executed by the Java Virtual Machine (JVM), which uses a just-in-time compiler to translate bytecode into machine code at runtime. [Stärk et al., 2012] This general approach partly matches with how Python code is interpreted, since Python also uses Bytecode to execute code.

Every Python interpreter has a virtual machine, there even is a Python interpreter which uses the JVM to execute instructions, Jython [Juneau et al., 2010]. The reference implementation, CPython, uses the Python Virtual Machine (PVM) to execute bytecode[1]. The Python compiler generates the bytecode of a program, which is passed to the PVM to execute it. The PVM cannot directly execute code, a bytecode object has to be compiled first.

There is a built-in module, `dis`[2], which can be used to inspect the bytecode objects of a Python program. Listing 5 shows the output of using this module on the same function in version 3.8 and version 3.11. There are some notable differences, for example, addition is listed as `BINARY_ADD` in Python3.8, while it is listed as `BINARY_OP 0 (+)`. This accurately reflects

[1] `https://devguide.python.org/internals/compiler/` describes some of the internals.
[2] `https://docs.python.org/3/library/dis.html`

the continuous change in the bytecode-format between Python versions. This also makes it clear that the bytecode of future Python versions cannot be interpreted by an older interpreter. Interpreting older bytecode is fine, since Python supports backwards compatability.

```
# PYTHON 3.8 dis.dis() OUTPUT
Disassembly of <code object hello_world at 0x7f23d6b6b2f0, file
↪  "showcase.py", line 3>:
  4           0 LOAD_CONST               1 ('Hello, ')
              2 LOAD_FAST                0 (name)
              4 BINARY_ADD
              6 LOAD_CONST               2 ('!')
              8 BINARY_ADD
             10 STORE_FAST               1 (greeting)

  5          12 LOAD_GLOBAL              0 (print)
             14 LOAD_FAST                1 (greeting)
             16 CALL_FUNCTION            1
             18 POP_TOP
             20 LOAD_CONST               0 (None)
             22 RETURN_VALUE
# PYTHON3.11 dis.dis() OUTPUT
Disassembly of <code object hello_world at 0x7fbd9e1c92f0, file
↪  "showcase.py", line 3>:
  3           0 RESUME                   0

  4           2 LOAD_CONST               1 ('Hello, ')
              4 LOAD_FAST                0 (name)
              6 BINARY_OP                0 (+)
             10 LOAD_CONST               2 ('!')
             12 BINARY_OP                0 (+)
             16 STORE_FAST               1 (greeting)

  5          18 LOAD_GLOBAL              1 (NULL + print)
             30 LOAD_FAST                1 (greeting)
             32 PRECALL                  1
             36 CALL                     1
             46 POP_TOP
             48 LOAD_CONST               0 (None)
             50 RETURN_VALUE
```

Listing 5: Difference between dis for the same code, different Python versions)

3.2 Decompilation

Decompilation is the process of converting executable code, such as bytecode or machine code, back into the source code from which the executable code originates. It is used to analyze and potentially modify a program of which the source code is not available, and therefore an important process to consider in the context of code obfuscation. For example, [Liu and Wang, 2020] have found that modern C compilers can produce higher quality code than what generally is believed to be possible. This shows that decompilation is a viable

When a python module is converted to bytecode, it generates a `.pyc`-file in which the raw bytecode is stored. To decompile Python code, the `.pyc`-file needs to be analyzed to reconstruct the original source code. It is possible for a Python decompiler to reconstruct most elements for a regular Python program, except for comments and unneeded formatting. Listing 6 shows how uncompyle6[3] is able to reconstruct class names, function names and variable names, but it is unable to reconstruct the formatting and comments of the program.

```
##### ORIGINAL PROGRAM
# Create a zoo instance
zoo = Zoo()

# Create animal instances
lion = Lion("Simba")
elephant = Elephant("Dumbo")

# Add animals to the zoo
zoo.add_animal(lion)
zoo.add_animal(elephant)

##### DECOMPILED OUTPUT
zoo = Zoo()
lion = Lion('Simba')
elephant = Elephant('Dumbo')
zoo.add_animal(lion)
zoo.add_animal(elephant)
```

Listing 6: Snippet of a program compared to the decompiled output (uncompyle6)

Uncompyle6 and its rework decompyle3[4] are Python bytecode decompilers that can handle bytecode created by Python versions up to version 3.8. A wiki is included on the github page of uncompyle6, on which a lot of technical details related to Python decompilation are explained, including its limitations such as issues with recreating control flow structures and the problems with supporting more recent Python versions.

[Ahad et al., 2023] propose a solution to overcome the limitations of the current Python decompilers. They introduce the concept of FETs, forensically equivalent transformations, which can be used to turn a binary that doesn't decompile into a binary that does decompile. This is mainly approached from the context of analyzing malware, since FETs are not semantic preserving meaning that some of the original code is changed or lost. This is usually not desirable when decompiling code, but for the usecase of analyzing malware it is an acceptable tradeoff. Their research allows for more binaries to be decompiled, and also has a strategy to change Python3.9 binaries into Python3.8 binaries, which makes it possible to decompile them.

There are some techniques that can be used to make it harder to decompile Python bytecode. For example, there is a tool called PjOrion[5] that is designed to obfuscate Python code by converting it to bytecode and then applying extra transformations to this bytecode, which makes it harder to analyze. This approach of transforming bytecode can be combined with regular source code obfuscating, in which the original code is obfuscated first and the bytecode is obfuscated afterwards. This approach will not be explored further since this thesis mainly focuses on source code obfuscation, but it does provide interesting research opportunities for the future.

[3] `https://github.com/rocky/python-uncompyle6`
[4] `https://github.com/rocky/python-decompile3`
[5] It can be found on a Russian forum about world of tanks

3.3 Freezing Python programs

Python programs are typically executed by executing the source code using a Python interpreter. *Freezing* provides an alternative to this approach, it turns a Python program into a regular executable binary. It achieves this effect by bundling a Python interpreter and the bytecode of the program together with all required modules. This executeable can then be distributed and ran by users without requiring them to install Python or the required modules.

A popular tool to freeze Python programs is PyInstaller[6]. It can freeze Python programs across multiple Python versions and operating systems. However, it is not possible to cross-compile for a different operating system. To generate binaries for different operating systems they have to be build separately on the corresponding operating system.

Freezing should not be confused with native compilation, in which the Python code gets compiled to native instructions that are not executed through a Python interpreter. An example of such a compiler is Nuitka[7]. Nuitka compiles Python code to C code (using source-to-source compilation) after which this C code is natively compiled. This generates a native binary that executes in the same way as the original Python program.

3.3.1 Freezing process (PyInstaller)

The freezing process might be slightly different for different tools, but the general approach is mostly the same. PyInstaller is one of the most popular freezing tools, so the way it freezes programs is shown as an example. It takes the following steps starting from an input file:

1. **Analysis and building spec file**: All imports of the input file are recursively analyzed until all the required dependencies are known. This information is used to generate a "spec file", which describes all the information that is required to freeze the program. This information includes all the imports and data files.

2. **Gathering resources and dependencies**: All required files and dependencies are collected and resolved, including the Python interpreter and libraries.

3. **Compilation**: The python code is converted to bytecode bundled with all other resources, including the interpreter, into a binary.

4. **Finalization**: The required adjustments and optimizations of the binary are applied. The python program is now a binary file.

As can be seen in this high level description, the source code of the Python program is compiled into bytecode and then packed into a binary. Although freezing is not designed as an obfuscation technique, it does protect the source code and makes it harder to analyze or modify the program. This means that it can be seen as an obfuscation technique even though it was not its intended purpose. However, this also means that there is no extra effort put into protecting the code. The next section proves this by showing a method to obtain the source code from frozen programs.

3.3.2 Reverse engineering frozen programs

The main goal of freezing is to create a binary from which the Python program can be executed, not to hide the source code. The code is compiled to bytecode and packed into the binary, which means it is possible to extract the bytecode. Section 3.2 discusses the decompilation of Python bytecode, so the source code of the program can still be reconstructed.

[6]`https://pyinstaller.org/`
[7]`https://nuitka.net/`

There are various tools that can extract the bytecode from frozen programs. For example PyInstaller Extractor[8] is a tool specifically made to extract `.pyc`-files containg bytecode from programs that were generated with PyInstaller. Once the bytecode is obtained, it can be decompiled into the original source code using the tools discussed in section 3.2. There are even tools that automate this entire process. For example, pydecipher[9] takes a frozen binary as input and produces the original source code of this binary as output.

Programs compiled with Nuitka (or any other native compilater) do not contain any of the original Python source code or bytecode, since the source code is translated to C code and its bytecode is not used at all. The only bytecode that can be present is the bytecode of the standard library (depending on the type of build mode used). It is therefore not possible to extract the original bytecode or source code of a Python program that has been compiled using Nuitka.

It is still possible to reverse engineer programs that have been built using Nuitka (or a similar tool that compiles Python to C/C++). They can be analysed and decompiled using the techniques discussed in 2.4. Because the original Python code is not used in the actual compilation process, it is not possible to extract or obtain this code. The code can only be reconstructed using the information obtained during the reverse engineering process.

[8] `https://github.com/extremecoders-re/pyinstxtractor`
[9] `https://github.com/mitre/pydecipher`

Chapter 4

Source Code Obfuscation

In chapter 2, obfuscation is defined as a defense against reverse engineering through changing software in a way that renders the software unintelligible while maintaining its normal functionality. This makes the software harder to reverse engineer.

[Collberg and Thomborson, 2002] propose the following formal rules for an effective approach to code obfuscation:

> Given a set of obfuscating transformations $\mathcal{T} = \{\mathcal{T}_1, \cdots, \mathcal{T}_n\}$ and a program $\mathcal{P}$ consisting of source code (classes, methods, statements, etc.) $\{\mathcal{S}_1, \cdots, \mathcal{S}_k\}$, find a new program $P' = \{\cdots, S'_j = \mathcal{T}_i(\mathcal{S}_j), \cdots\}$ such that:
>
> - P' has the same observable behavior as P, i.e., the transformations are *semantics-preserving.*
> - The *obscurity* of P' maximized, i.e., understanding and reverse engineering P' will be strictly more time-consuming than understanding and reverse engineering P.
> - The *resilience* of each transformation $\mathcal{T}_i(\mathcal{S}_j)$ is maximized, i.e., it will either be difficult to construct an automatic tool to undo the transformations or executing such a tool will be extremely time-consuming.
> - The *stealth* of each transformation $\mathcal{T}_i(\mathcal{S}_j)$ is maximized, i.e., the statistical properties of S'_j are similar to those of $\mathcal{S}_j$.
> - The *cost* (the execution time/space penalty incurred by the transformations) of P' is minimized.

(Directly quoted from [Collberg and Thomborson, 2002])

This formal set of rules can be summarized in a simpler, informal definition:
"Obfuscation is the ***transformation*** *of a program into a* ***semantically identic*** *program that is* ***harder to reverse engineer****, while minimizing the costs of increased execution time or extra required space."*

As this thesis focuses on obfuscation techniques for Python, this comes down to transforming the original source code of python programs into a new version that executes in the same way but is less intelligible. This chapter delves deeper into these exact transformations by going over a taxonomy that classifies different techniques. This taxonomy gets explored with a handful of common or useful techniques for Python.

4.1 Context

There are several programming or programming-related concepts which are important for this chapter. This section introduces some definitions for a few of these concepts. The definitions of the following concepts are covered: runtime, compile time, abstract syntax tree, basic blocks and control flow.

A lot of other terminology such as variables, functions, keywords, etc. is also used to explain certain obfuscation techniques. It is assumed that the reader has enough familiarity with programming and these related concepts to understand them within the given context. As such, explicit definitions of these concepts are not given.

This section also contains an example use-case of the use of obfuscation techniques in a real world scenario. As will be discussed, it is in theory "impossible" to actually obfuscate a piece of software that gets distributed to an end-user. The use-case proves that obfuscation can still provide value for a software developer even though it is considered "impossible".

4.1.1 Definitions

Runtime refers to the period of time in which a program is executing. It starts when the program begins executing and it is done when the program has finished executing. A program is considered to be "at runtime" when it is in execution.

Compile Time refers to the period of time in which a program is being compiled. Elements like constants, array sizes (of static arrays), or any type of explicitly static data are processed at compile time and cannot change during runtime.

```
1  #include <stdio.h>
2  #define PI 3.14159  // this value is known at compile time
3
4  int main() {
5      double radius;
6
7      printf("Enter the radius: ");
8      scanf("%lf", &radius);
9
10     double area = PI * radius * radius; // computed at runtime
11
12     printf("Area of the circle: %f\n", area);
13
14     int alpha = 42; // the value of alpha can change during runtime
15     printf("The variable alpha contains the value %d\n", alpha);
16
17     return 0;
18 }
```

Listing 7: Compile Time and Runtime for a C program

Listing 7 shows the difference between runtime and compile time. The constant PI is defined on rule 2. This value is known at compile time because it cannot change. The value that the variable area holds on rule 10 is computed at runtime. The value of the variable alpha on rule 14 can change during runtime.[1]

[1] It is possible that for this specific example the compiler can optimize this code and will treat alpha like a constant because it is used once without being changed. The value of area relies on user input, therefore area always has to be computed at runtime.

Python is an interpreted language, it only has runtime and no compile time. In this case, runtime refers to the period of time in which a program is being executed by the interpreter.

The **Abstract Syntax Tree (AST)** of a program "represents the hierarchical syntactic structure of the source program" [Aho et al., 2006] (Chapter 2). It is a tree structure built by parsing the source code of a program. The nodes represent the different elements of the program. If a node represented an if-statement, the children of this node would be the if-test, the body of the if-statement and the else body.

The AST documentation for Python [2] shows the abstract grammar that describes Python. It also lists the type that is associated with every token. Listing 8 shows a simple Python program, figure 4.1 shows a visualization of this program (which is seen as a Module).

```python
print("AST example")

def sum_of_numbers(number_1, number_2):
    return number_1 + number_2

a = 12
b = 42

sum = sum_of_numbers(a, b)

print(sum)
```

Listing 8: A simple Python example program

A **basic block** [Aho et al., 2006] is a sequence of instructions to which the following properties apply:

1. The control flow can only execute starting from the first instruction of the block.
2. The control flow will leave the block without branching or halting, except possibly for the last instruction

The **control flow** of a program is the order in which instructions are executed. This order is influenced by control-flow statement such as if-statements, loops and function calls. It can be represented by a graph, the control flow graph, in which basic blocks are the nodes and the potential transfer of control flow between the blocks is represented by the edges. Listing 9 contains a piece of code, of which the control flow graph is shown in figure 4.2.

```cpp
i = 1;
s = 0;
while (i <= 100) {
    s += i;
    i++;
}
```

Listing 9: C++ example program by [László and Kiss, 2009] (Figure 1)

4.1.2 Use-case example

Obfuscation does not provide absolute protection. [Barak et al., 2012] even conclude that in theory, actual obfuscation of a program is impossible. It is assumed that given enough time and

[2] `https://docs.python.org/3/library/ast.html`

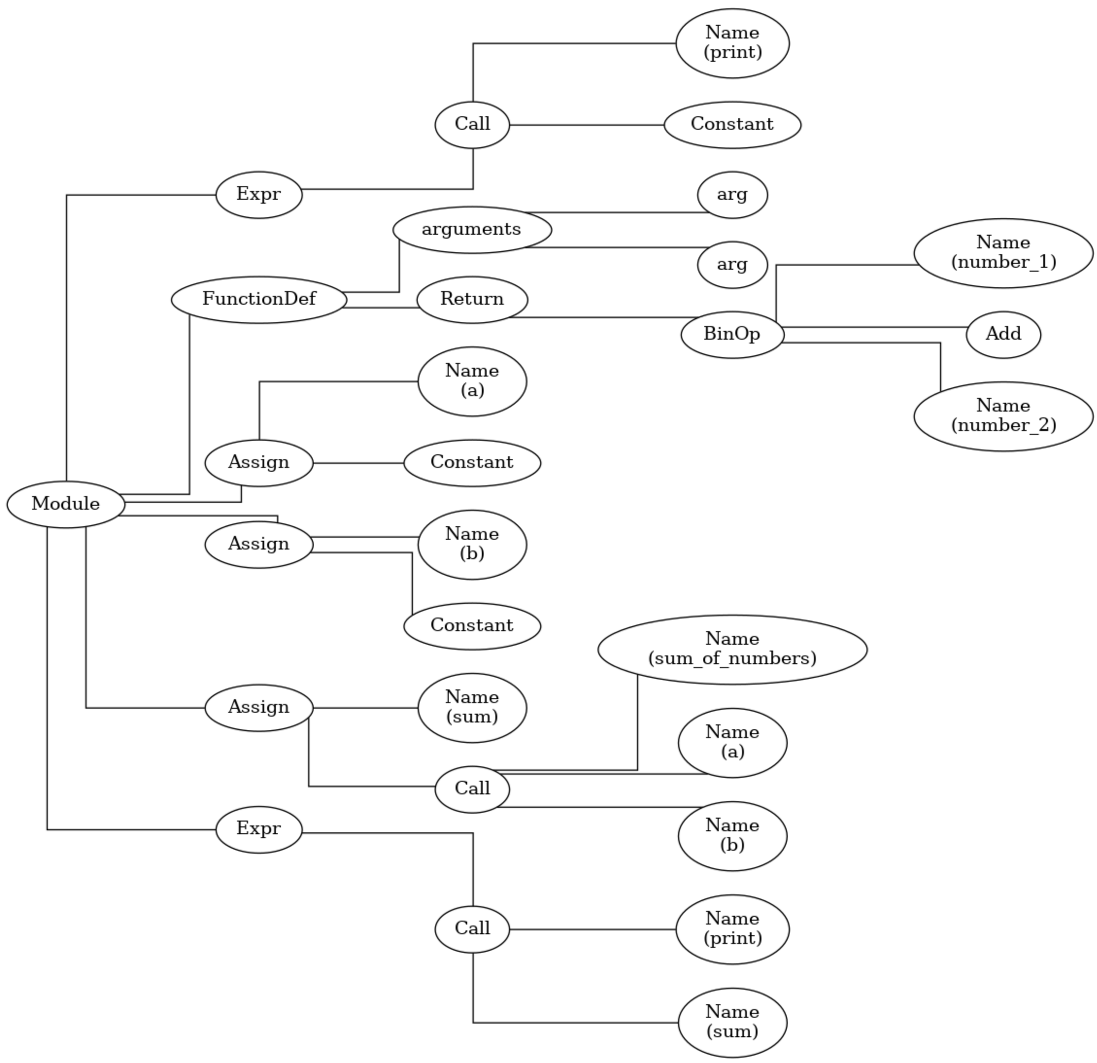

Figure 4.1: Abstract Syntax Tree of listing 8

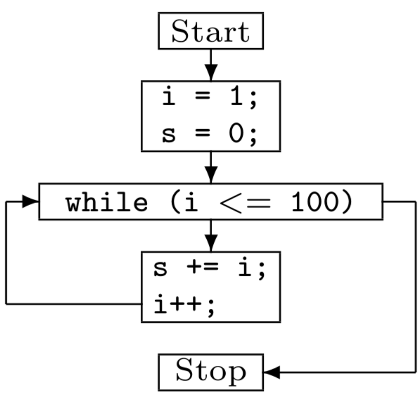

Figure 4.2: CFG example by [László and Kiss, 2009] (Figure 1)

resources any form of obfuscation will be broken or reversed. The main goal is therefore to make the process of analyzing the program or removing the obfuscation as costly as possible.

The main use-case for obfuscation is to complicate Man-At-The-End (MATE) attacks. It is assumed that an attacker with access to a piece of software has unlimited time to analyze the software. If this software isn't obfuscated, then it will likely take a reasonable amount of time and effort for the attacker to reverse engineer the software. When obfuscation is used, the amount of time and effort required increases making it less likely that the attacker will succeed.

There are situations where buying extra time through obfuscation can have a direct impact. For example, when a zero-day vulnerability is discovered in a popular software library, it is possible that attackers can exploit this to take over vulnerable computers running this software. If a piece of software has this vulnerability, but it is obfuscated in a way that makes it very hard to know that this library is being used, the attackers might not be realize that this piece of software is vulnerable. For example, the log4shell vulnerability [Everson et al., 2022] that got published in 2021 could be detected by automatic scanning for this vulnerability. These types of scanners can be circumvented with proper obfuscation, so it is possible that obfuscation could have mitigated the impact of this vulnerability.

4.2 Taxonomy

This section contains a taxonomy that can be used to classify obfuscation techniques[3]. This taxonomy is based on an existing taxonomy that is adjusted to be useful for Python. It is used to compile a list of techniques that can be applied to Python, including a lot of examples of the transformation applied to a Python program when relevant. This is a non-exhaustive list since there exist a lot of obfuscation techniques, but it does include most techniques that are frequently discussed in literature.

An initial taxonomy is taken from [Anckaert et al., 2007], who listed four fundamental program properties that should be used to evaluate obfuscation techniques: *instructions, control flow, data flow* and *data*. This taxonomy is intended to analyze which evaluation metrics can be used by analyzing how a metric reflects one or more properties. However, it can also be used to classify obfuscation transformations, since they also have an impact on these properties.

Transformations will frequently impact multiple properties, but there is usually a certain property that is the focus of a technique, which has some impact on the other properties when the program is transformed. For example, a technique that tries to obfuscate the data of a program is targeted at the data-property, but the transformation will likely impact the instructions and possibly the data flow of the program.

Adapting the program properties to be used for classifying Python obfuscation creates the following taxonomy:

- **Control Flow Obfuscation**: Techniques that try to hide information about the control flow of the program.
- **Data Obfuscation**: Techniques that try to hide what data is being processed.
- **Instruction Obfuscation**: Techniques that are used to hide what instructions are being executed.
- **Interpreter Specific Techniques**: Techniques that rely on the interpreted nature of Python.
- **Other techniques**: Techniques that have some unique quality which makes them unfit for the other categories.

[3] The term "technique" and "transformation" are used interchangeably in this section.

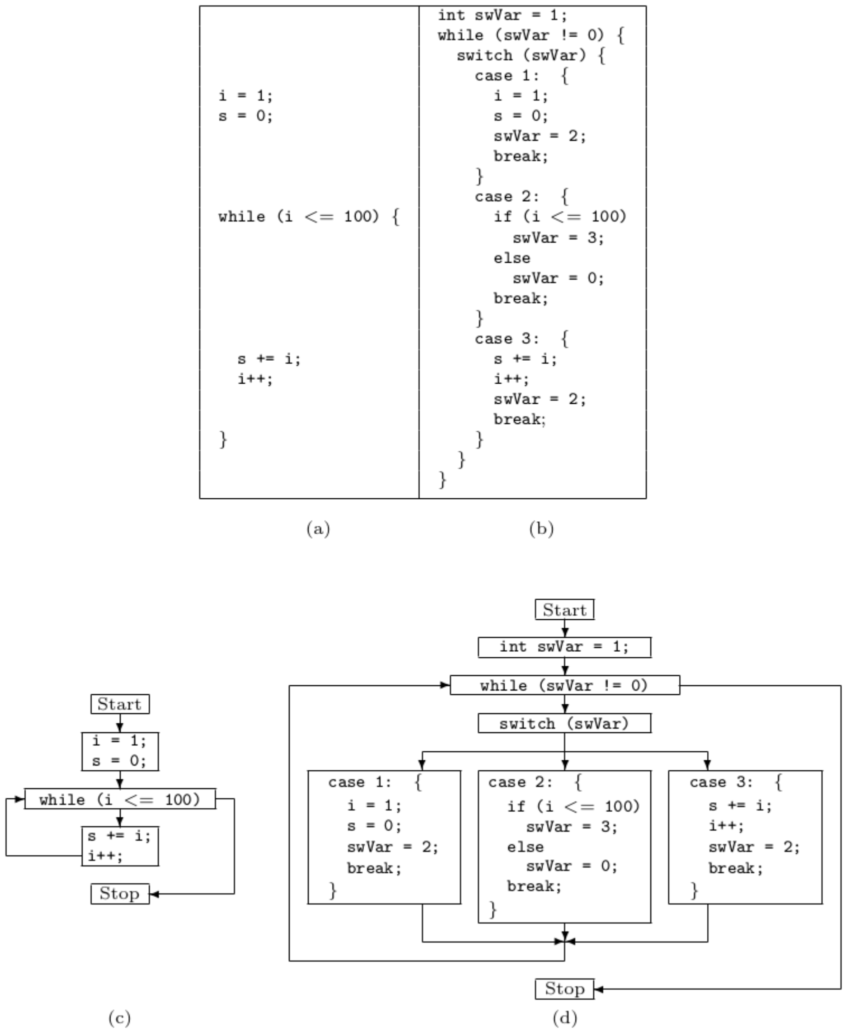

Figure 4.3: Flattening example by [László and Kiss, 2009] (Figure 1)

4.2.1 Control Flow Obfuscation

Control flow obfuscation tries to hide any information about the control flow of a program under normal execution conditions. This can be done by modifying the structures that a programming language normally uses to control the flow of execution.

Control Flow Graph Flattening is the process of hiding all control flow information by removing all edges between the nodes of a control flow graph. A dispatcher node is added which is connected to all nodes and every node connects back to this node. Instead of going through the normal control flow, the flow always goes through the dispatcher, any of the other nodes, and then back to the dispatcher. This means that regular tools to analyze control flow information cannot be used. An example of a program before and after control flow flattening is shown in figure 4.3. [László and Kiss, 2009] [BinShamlan et al., 2019]

Junk Code/Dead Code Generation Dead code is any code that might get executed, but which has no impact on the results of the computation. Unreachable code is any code that is part of a basic block that can never be reached through the normal program control flow. Unreachable code is therefore always considered to be dead code. Listing 10 shows an example containing both dead code and unreachable code. [Bhansali et al., 2022]

```
def foo():
    out = 4 ** 3 + 12    # dead code, not used
    return 42
    print("This will never print") # unreachable code
```

Listing 10: Dead code and Unreachable Code

Opaque Predicates [Liang et al., 2017] are expressions that have to be evaluated at runtime to determine the control flow of a program, but the programmer already knows what this expression will evaluate to while developing the program. A predicate is an expression that evaluates to True of False, and it is considered opaque when the programmer knows the expected value. Opaque predicates can be combined with junk code generation for sections which the programmer knows will always or never be reached at runtime, which allows the control flow graph to be made a lot more complex. [Kang et al., 2021] [Collberg and Thomborson, 2002]

```
import time

# automatic tools can't assume the output of a system library call
# but the programmer knows that this statement will always be true
if time.time() > 1:
    # this block will always execute
    doSomething()
```

Listing 11: Opaque Predicate using system time

4.2.2 Data Obfuscation

Data obfuscation tries to hide information about the data that is used by the program. This can be done by changing the way data is stored or processed, making it harder to analyze how the program interacts with the data.

Encoding/Encrypting Literals Using (uncommon) encoding techniques for static data (strings, booleans, integers). Any (human-readable) string can be replaced with a string using a different encoding, and integers can be encoded using a different base. As booleans only have a binary value, encoding them in a different format is possible but relatively easy readable. They can instead also be "encoded" in an expression that evaluates to the desired value, either by a constant expression for each value, or by a unique expression for every boolean constant.

It is also possible to replace every piece of static data with a call to a decryption function, with the original data encrypted. Listing 12 shows an example of a program containing static data, which is turned into dynamic data by evaluating it as a runtime expression. [Bhansali et al., 2022] [Herrera, 2020]

Split Variables Splitting the value of a variable across multiple variables. This can be done for strings by literally splitting the characters across multiple variables, but also for integers by replacing the integer by an expression using multiple integers that evaluates to the desired value. [Herrera, 2020] [Collberg and Thomborson, 2002]

Array Restructuring Restructuring the elements in an array to obscure (the operations performed) on the array, while keeping all the data accessible. This can be categorised in splitting, merging, folding and flattening. [Collberg and Thomborson, 2002]

```
# BEFORE ENCODING
x = True
if(x):
        print("hi")
else:
        print("hello")
# AFTER ENCODING
x = (()==())
if(x):
        print(''.join(map(chr, [104, 105])))
else:
        print(''.join(map(chr, [104, 101, 108, 108, 111])))
```

Listing 12: Literal Encoding of strings and booleans

4.2.3 Instruction Obfuscation

Instruction obfuscation tries to hide what instructions are executed by the program.

Antidebugging is any type of method that makes a regular debugger crash or unable to obtain information from a process. For example, a program can have a thread that monitors the main thread to detect debugging, after which the thread exists or forcefully crashes the program. [Collberg and Thomborson, 2002]

Antidisassembly attempts to counteract disassemblers, making the output invalid or making the disassembler crash. This can take many forms, for example by generating an assembly-instruction of which it is known that it causes problems for disassemblers. This technique cannot be applied to Python because the code isn't compiled to machine code. [Anckaert et al., 2007]

Virtualization turns the program into a virtual machine that runs the actual program. The original source code of the program is turned into a format (such as a unique bytecode) that can be executed by the virtual machine. This is possible in Python, but it would likely cause a high amount of overhead compared to running the program normally. [Bhansali et al., 2022] [Kang et al., 2021]

4.2.4 Interpreter Specific Techniques

These are techniques that typically can be used by interpreted programming languages but commonly are not (directly) possible using non-interpreted programming languages.

Eval and Exec `eval()` and `exec()` are built-in functions that allow a string of text to be ran by the interpreter as a piece of executeable code. Eval is used for expressions, exec is used for statements. This makes it possible to build strings at runtime and use them to execute code that cannot be analyzed by using static analysis. Listing 13 shows how a function is defined with a string at runtime using exec at line 1, and how the value of y is evaluated to the value 3 on line 3. [Herrera, 2020]

```
1 exec("def sayHelloReturnOne(): print(\"Hello world!\"); return 1")
2 x = sayHelloReturnOne()
3 y = eval("x * 3")
4 print(y)    # output: 3
```

Listing 13: A function defined with exec with the return value used by eval

Keyword Substitution Some programming languages allow constructs (such as a preproccessor-macro in C) that make it possible to substitute language keywords with self-defined names. This can be used to replace some (or all) keywords with a substitution, or even different substitutions for the same keyword. This has no impact on the resulting output after compilation, but for a interpreted language this can be used to make it harder to identify which keywords are being used.

Name Obfuscation/Identifier Stripping replaces any sort of (semantic) identifier (variable or parameter name, function name, class name, ...) with a new identifier that is hard to read and loses semantic value. These new identifiers can be re-used within different scopes to further obfuscate where these identifiers are used. Comments and docstrings can be removed as well as they have no impact on the execution of a program while they might provide insights on how the code works. Listing 14 shows an example before and after this transformation. [Lacomis et al., 2019] [Kang et al., 2021]

```
# BEFORE NAME OBFUSCATION
# class containing the name and age of a person
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # @return the info as a string
    def toString(self):
        out = self.name + " " + str(self.age)
        return out
# AFTER NAME OBFUCCATION
class l:
    def __init__(l, I, lI):
        l.ll = I
        l.lI = lI
    def l(l):
        I = l.ll + " " + str(l.lI)
        return I
```

Listing 14: Name Obfuscation/Identifier Stripping

Redefining Functions/Function Aliasing When a function-definition is interpreted at runtime, it is used to construct a function-object that is tied to the name of the function. This function-object is immutable, but the name that is used to access the function-object can be modified to reference a different function-object. A function-name that is being used to call a function can therefore call different functions at runtime, depending on which function-object is currently mapped to the name. Listing 15 shows an example in Python in which the function with the name `addOne()` is replaced by the function `subtractOne()`. [Bhansali et al., 2022]

4.2.5 Other

Minification removes unneccesary characters and minimizes the total length of the source code of a program. This is primarily done for JavaScript, as the resulting smaller filesize allows faster loading of webpages. A side effect of minification is making programs harder to read, as any formatting gets removed. This is only useful for interpreted code because this only affects the formatting of the code and not the execution of instructions.

MovFuscator[4] A proof-of-concept technique that was showcased in a tool that compiles a C program into only "mov" instructions at machine-code level. This means that there is no regular

[4] `https://github.com/Battelle/movfuscator`

```
def addOne(x): return x+1
def subtractOne(x): return x-1

a = 0
for i in range(2): a = addOne(a)

addOne = subtractOne # redefined function

for i in range(5): a = addOne(a)

print(a) # a = -3 after calling addOne(a) 7 times
```

Listing 15: Redefining a function

control flow, and standard reverse engineering tools cannot be used to analyze the resulting source code. It relies on the compilation to machine code, and is therefore not applicable to Python. It falls beyond the scope of this thesis, but is a unique approach to obfuscating programs that completely counteracts regular reverse engineering practices.

Function Copies Functions can be copied multiple times, for which the copies have the same function-body but a different name. This makes it harder to analyze which functions are being called a lot, since the calls to a function can be split between the different copies. It can be combined with other techniques to ensure that every function has a different body yet it keeps the same functionality. This makes it even harder to keep track of the functions that are being called, because every function needs to be analyzed separately to know whether it is a copy of an existing function. [Bhansali et al., 2022]

4.3 Existing Obfuscators for Python

There are some tools for obfuscating and protecting Python programs. This section briefly goes over some of the more prominent ones, compares their features and discusses the potential for future tools.

PyArmor[5] is a free shareware Python obfuscator. Its main obfuscation techniques are name obfuscation and some native compilation, which works by transpiling parts of the Python code to C which then gets compiled to native instructions. The rest of the tool is mostly a packer, in which a custom shared library is generated to unpack the code at runtime. It mainly uses some tricks to obfuscate and encrypt the code-objects at runtime, which makes it difficult to obtain the bytecode objects for decompilation.

However, there still are tools such as *PyArmor-Unpacker*[6] which can still retrieve the packed bytecode objects, either through dynamic execution in which some code is injected into the running program, or even through a static attack to avoid having to run the program, which can be useful for malware analysis. However, even with these tools existing, it takes some effort and time to unpack and decompile the code, which still makes PyArmor a useful tool.

PyMinifier, as the name implies, is mostly a tool used to minify Python scripts. This is mostly achieved by changing some of the formatting, such as removing unnecessary whitespaces and comments. It also has an option to compress the input program, which then gets decompressed at runtime (by its own code). The most interesting feature is the obfuscation, which is an implementation of Name Obfuscation that also has the option use aliases to hide certain built-in elements. It is not a very complex obfuscation technique but it is still useful. It has also an

[5] `https://github.com/dashingsoft/pyarmor`
[6] `https://github.com/Svenskithesource/PyArmor-Unpacker/`

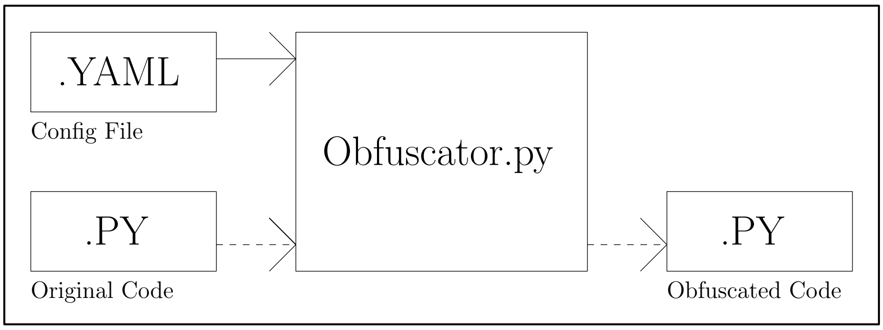

Figure 4.4: Obfuscator usage flow.

option to use non-latin characters for its name ofuscation, which in theory does not change the effect of the obfuscation, but possibly

Outside of these two tools, there are a lot of other smaller tools, such as *PyCloak*[7], *Anubis*[8], and a few others. PyCloak can apply literal encoding and obfuscate functions by generating lengthy lambda expression that evaluate to the correct value at runtime. Anubis applies name obfuscation, can generate junk code and has has an anti-debug technique, which in practice just inserts some static code that can possibly crash debuggers. A lot of other tools do some basic "obfuscation" such as compressing the script, formatting it as a long hex string, and then using `exec()` to call the functions required to decompress and execute themselves. These do add some confusion, but they require so little time and effort to analyze that they are not considered as viable obfuscators.

The main obfuscation technique that is implemented a lot is name obfuscation. This makes sense since Python is an interpreted language, meaning that the different names and identifiers provide a lot of information about the program, while they are not required for the program to correctly execute. Besides that, the only interesting techniques are the literal encoding from PyCloak and two features from Anubis: anti-debugging and junk code generation. These are all legitimate techniques that increase the amount of effort that is required to analyze programs. PyArmor does not use conventional code obfuscation techniques, but its packing technique does add some protection to Python programs so it is still a useful tool.

4.4 Implementation

A set of obfuscation techniques that is applicable to Python has been selected and implemented as a proof-of-concept obfuscator. This implementation is also used in chapter 5 during the evaluation of Python obfuscation techniques. This section covers the features and some technical details of this implementation.

```yaml
INPUTFILE: measure_in/prime.py
OUTPUTFILE: measure_out/prime-functioncall-flattener.py
TESTCODE: False
TRANSFORMATIONS: functioncall, flattener
```

Listing 16: Example YAML configuration file

The obfuscator is written in Python and transforms a file containing Python code into a new file containing an obfuscated version of this code. It is designed to work on a single file. To obfuscate a project containing multiple files it has to be used on each separate file. This will

[7]https://github.com/addi00000/pycloak
[8]https://github.com/0sir1ss/Anubis

work for every transformation except name obfuscation, since it does not keep track of names throughout multiple files.

The program (Obfuscator.py) takes a single argument, a configuration file using YAML as its formatting language. This file contains the required information to run the obfuscator: the input file, the output path, the list of transformations and a flag that can automatically test the obfuscation result. This general usage flow is show in Figure 4.4 with Listing 16 showing an example configuration-file that can be used to run the obfuscator.

The transformations[9] are applied in the order in which they are listed. This can be seen as a pipeline of transformations which is applied to the original program. The order of transformations can make a difference on the final result so it is important to specify the intended order.

The test-flag (`TESTCODE`) is used to run the original program as well as the obfuscated version. The output of both versions is compared to check whether the obfuscation changed the output as a basic check. This is done using a subprocess running both versions. It is used for testing and debugging this proof-of-concept implementation. In practice a feature like this should be properly sandboxed and secured.

4.4.1 Techniques and features

The obfuscator is written in Python and does not require any external modules or tools to execute. It mainly relies on the ast module[10], which is a standard module (when using CPython). This module can parse a file or string containing Python code into a collection of nodes. These nodes represent all the elements of the Python program and can be modified to change the program.

As of Python 3.9 it is possible to use the ast module to turn a collection of nodes into Python code with the function `ast.unparse()`. This allows the obfuscator to parse a program into ast-nodes, transform the program by changing the nodes, and output the resulting program as Python code. One of the implemented techniques (control flow flattening) uses keywords introduced in Python 3.10 and later. Therefore a Python version of 3.10 or later is required when using this technique, with at least version 3.9 being an absolute neccesity to use the obfuscator.

The following techniques are implemented:

- Name Obfuscation/Identifier Strippinfg (*name.py*)
- Literal Encoding (*literals.py*)
- Function Call Obfuscation (*functioncall.py*)
- Control Flow Flattening (*flattener.py*)

Each transformation has its own Python module with a function (`transform_code()`) that takes a string of text representing Python code and returns a string of Python code after the operation has been applied. The name of the Python file containing their code is listed after the name of each technique.

These modules are loaded dynamically using `importlib`. This allows the obfuscator to load a .py file based on the configuration file that was provided. This makes it easier to add or rename transformations without having to hard-code every transformation in the obfuscator. Instead, the obfuscator provides a simple framework to add new transformations without having to change its code.

[9] The YAML-format is used incorrectly for a list of values due to a mistake. A comma separated string is used instead of the normal formatting. This is parsed correctly by the obfuscator so it does not cause any issues.
[10] `https://docs.python.org/3/library/ast.html`

4.4.2 Transformations

The list of implemented transformations is given in the previous section 4.4.1. This section briefly goes over every transformation, giving a high level explanation of how it functions. An example of every technique will be given to show the impact on a Python program.

Name Obfuscation goes over all the identifiers in a program and replaces them with a new name that is comprised of a combination of the letter `I` (uppercase i) and the letter `l` (lowercase L) whenever it is possible to rename that identifier. It also removes all the comments and docstrings of the original script. It keeps a dictionary that maps the original identifiers to the renamed identifiers. Listing 17 shows a snippet of code before and after Name Obfuscation.

```python
# BEFORE NAME OBFUSCATION
# class containing the name and age of a person
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # @return the info as a string
    def toString(self):
        out = self.name + " " + str(self.age)
        return out
# AFTER NAME OBFUCCATION
class l:
    def __init__(l, I, lI):
        l.ll = I
        l.lI = lI
    def l(l):
        I = l.ll + " " + str(l.lI)
        return I
```

Listing 17: Name Obfuscation Example (same code as listing 14)

This transformation traverses the AST of a program, and every identifier is looked up in this dictionary. If it exists as a key in the dictionary, it gets replaced by the value in the dictionary, if not then it is added to the dictionary and a unique name is generated. However, there could be problems when identifiers are renamed that should not be renamed, like built-in functions (such as `print`) or imported functions.

This problem is prevented by automatically including all built-in functions and functions for built-in types in the dictionary, in which these identifiers are mapped to themselves (meaning that they are replaced by their own value when the AST is traversed). Issues with imported modules are prevented by explicitly checking whether an identifier is being used in the context of an imported module.

Literal Encoding[11] replaces all boolean and string literals with an expression that evaluates to the literal at runtime. Listing 18 shows a snippet in which the boolean value `True` and two strings are encoded. Booleans are encoded with a comparison-expression that evaluates to their original value , the strings are encoding with a `map`-expression that evaluates to the original string. The transformation traverses the AST and replaces every node that contains a boolean literal or a string literal with a new node that contains the expression that will evaluate to the literal at runtime.

Function Call Obfuscation is the most simple transformation, it replaces any regular function

[11] Partially based on Pycloak (`https://github.com/addi00000/pycloak/blob/main/pycloak/main.py`) - version december 27, 2022

```
# BEFORE ENCODING
x = True
if(x):
        print("hi")
else:
        print("hello")
# AFTER ENCODING
x = (()==())
if(x):
        print(''.join(map(chr, [104, 105])))
else:
        print(''.join(map(chr, [104, 101, 108, 108, 111])))
```

Listing 18: Literal Encoding Example (Same code as listing 12)

call with a dynamic call that is made at runtime, by using the `eval`-statement. This means that the function-call is converted to a string and passed as an argument to the eval-function. Listing 19 shows a script that calculates prime numbers to which function call obfuscation is applied. It is easy for a human reader to recognize what the original code looks like, since it is literally encoded as the string-argument for the eval function.

```
number_of_primes = 30000

def is_prime(num):
    if num <= 1:
        return False
    for i in eval('range(2, int(num ** 0.5) + 1)'):
        if num % i == 0:
            return False
    return True

def main():
    count = 0
    num = 2
    primes = []
    while count < number_of_primes:
        if eval('is_prime(num)'):
            primes.append(num)
            count += 1
        num += 1
    eval("print('First 50 prime numbers:')")
    for i in eval('range(5)'):
        for j in eval('range(10)'):
            index = i * 10 + j
            eval("print(f'{primes[index]:<4}', end='')")
        eval('print()')
eval('main()')
```

Listing 19: Function Call Obfuscation

Control Flow Flattening is the most complex transformation, and it has never been applied to Python before. It modifies the control flow of every function in a way that the flow of control is managed by a dispatch-structure, in this case a `match-case` structure. Every basic block of the function is accessed through the dispatcher, and returns to the dispatcher after the block has finished execution. Listing 20 shows a function before and after flattening. The "flattened" parted of the obfuscated code can be observed visually because Python uses indentation to indicate control flow changes. The original code has some variation in the level of indentation. The obfuscated code, once the match-statement is reached, only has one extra level of indentation at most between all the handled cases.

```python
# BEFORE FLATTENING
def is_prime(num):
    if num <= 1:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True
# AFTER FLATTENING
def is_prime(num):
    swVar = 0
    while swVar != -1:
        match swVar:
            case 0:
                if num <= 1:
                    swVar = 2
                else:
                    swVar = 1
            case 2:
                return False
                swVar = 1
            case 1:
                var_1ea097 = iter(range(2, int(num ** 0.5) + 1))
                swVar = 4
            case 4:
                try:
                    i = next(var_1ea097)
                    swVar = 5
                except StopIteration:
                    swVar = 3
            case 5:
                if num % i == 0:
                    swVar = 6
                else:
                    swVar = 4
            case 6:
                return False
                swVar = 4
            case 3:
                return True
                swVar = -1
```

Listing 20: Control Flow Flattening Example

This implementation is partially based on a publication by [László and Kiss, 2009] in which a practical implementation for a C++ control flow flattener is discussed. There are quite a lot of fundamental differences between C++ and Python, but the high level control flow structures behave mostly the same. This paper provided a starting point of reference to start creating this transformation, which has useful for solving problems that are specific to control flow flattening.

For example, the way that the Python ast-library traverses the nodes with the default traversal function is not optimal for flattening the control flow, so the traversal algorithm from the paper was taken and applied to Python. The correct marking of basic blocks and control flow structures, which in this implementation are `if`, `for` and `while` statements, relies on proper tracking of certain information. This becomes more complex when nested control flow structures are traversed and the right exit-points need to be tracked. The correct approach for marking blocks and handling their exit-points is copied directly from this paper.

There are a lot of complex and nuanced problems related to control flow flattening, most of them have not been fully solved in this implementation, since it is a proof-of-concept and not a full implementation. For example, functions containing exception handling code are not flattened, because this is a structure with some complex challenges. It is even harder to flatten exception handling in Python, since the C++ solution relies on `GOTO`-statements. There are also a lot of edge-cases (such as the else-statement in loops) that are not handled correctly, but it can handle most conventional functions.

4.4.3 Deobfuscation

Deobfuscation is the process of taking an obfuscated program and making it easier to understand. This is done by applying transformations that partially or fully revert the used obfuscation transformations. A deobfuscator is an automatic tool that can take an obfuscated program as input and produces a version that is easier to understand. Deobfuscation can also (partially) be done by hand as it is possible that writing a deobfuscator is more challenging that manually reverse engineering the interesting components of a program.

Deobfuscation is possible because a obfuscated program is semantically identic to the original program. All the required information to execute the program still has to be included in the program. This implies that it is possible to creation a transformation that takes the obfuscated program as input and creates a less obfuscated/more intelligible version as output. [Barak et al., 2012] also support this notion as they have proven that true obfuscation is "impossible" in theory. This implies that deobfuscation is always possible to a certain extent.

Chapter 5 discusses the concept of resilience by [Collberg et al., 1997] in depth. It is their approach to classify the amount of effort required to write an automatic deobfuscator. A five-point scale (*trivial*, *weak*, *strong*, *full*, *one-way*) is used to classify every transformation. This is based on the scope of the transformation as well as the time that an obfuscator needs to deobfuscate a program.

The implemented techniques are classified as the following: one-way (*name obfuscation*), trivial (*literal encoding*), trivial (*function call obfuscation*) and weak (*control flow flattening*). This implies that writing a deobfuscator for the last three transformations should be relatively easy. The trivial transformations only affect a single basic block and the weak transformation stays within the control flow graph of a function.

The first transformation, *name obfuscation,* is marked as a one-way transformation. It is not possible to use the obfuscated program to reconstruct the original source code. It is one-way because the transformation discards information that is not needed by the interpreter to execute the code (such as comments and identifiers).

[Lacomis et al., 2019] have however shown that it is possible to use machine learning to partially reconstruct semantic identifiers. The original comments and variable names cannot be reconstructed with certainty but identifiers can be created based on the semantic context. This is still considered to be deobfuscation as the new identifiers make the program more intelligible.

Literal encoding is implemented as a transformation that takes static data and transforms it into an expression that gets evaluated to the static data at runtime. A deobfuscator can relatively easily be created for this implementation as it generates a consistent pattern. Once this pattern is encountered, the static data can be recreated and replace the expression.

Function call obfuscation is the most basic transformation in which `eval` is called instead of the function that is supposed to be called. A deobfuscator can be created that parses the program and iterates over every function call. It can then convert any calls to `eval` to a direct call instead.

Due to the generated patterns in the source code this can even be deobufscated without fully parsing the program. By simply going over the source code as a string and replacing `eval(X́')` with `X` a basic deobfuscator can be constructed.[12] On Linux this can be achieved using the sed tool and a regular expression. This results in the following command:

```
sed -E "s/eval\('([^']+)',*\s*\)/\1/g" input.py > output.py
```

Control flow flattening is the most complex transformation. It reconstructs the entire control flow graph of every function which classifies it as having weak resilience. Most control flow structures (if, for, while) are replaced with a dispatch structure that manages the control flow using match statements.

There are still patterns that can be taken to reconstruct the original control flow graph. For example, for loops are replaced by repeatedly calling `next()` on an iterator. Whenever `next` is encountered, it can be deduced that the following matched pattern contains the start of the body of a for-loop. Furthermore, the identifier used for the block containing the next-block is also helpful. This identifier is only used for the initialization of the loop and the end of the loop.

Every basic block has an identifier that the match statement uses to select the correct block. The variable holding this identifier is continuously modified to control which block is executed next. This variable can be tracked to construct a graph representing the dispatch flow between possible blocks. An automatic deobfuscator could create this graph and use it to recreate the original control flow elements.

Specifically for this implementation there is also an issue with the chosen identifiers for blocks. They are chosen by incrementing a counter every time a new block is processed during the obfuscation. This transformation goes through the CFG of the function using depth-first traversal. The value of identifiers can be exploited to ease the process of reconstructing the control flow graph. Randomization of these identifiers would remove this weakness.

A potential deobfuscation approach for every implemented technique has been discussed. This is a lot easier to do for an obfuscator of which the transformations and implementation are fully known. In practice the programmer only has the obfuscated program to work with. This requires the programmer to first analyze the program and figure out a way to revert every transformations. The goal of obfuscation is mainly that this process takes as much time and effort as possible.

4.4.4 Potential Improvements

The previous section discusses some theoretical deobfuscators that can be created to revert the implemented transformations. This section discusses improvements that could make this deobufscation harder. It also discusses general improvements to the transformations and the obfuscator itself.

Chapter 5 also contains a section (5.7.1) that discusses improvements for the obfuscator. It uses the results gained of some practical tests using multiple evaluation metrics. That section can be seen as a complement to this section.

[12] This deobfuscator could be broken by a piece of source code specifically crafted to incorrectly match the pattern. The theoretical deobfuscator described in the previous section would still work correctly.

There are some features and edge-cases missing from the existing transformations since they were created as proof-of-concepts. A simple improvement would therefore be to add any missing features and do extensive testing to handle more edge-cases. There are several techniques mentioned in 4.2 which can be applied to Python. These could also be implemented to add more transformations to the obfuscator.

Name obfuscation achieves its main goal: removing all identifiers, docstrings, comments and similar information. The main problem is that it is designed to only work on a single file. To handle multiple files requires some high level changes to make sure the changing of identifiers does not break imports across files.

The renaming of variables can also still be improved because the current implementation does not look at the scope of variables. For example, a variable called `foo` used in function `bar` gets renamed to `lIl`. If a function `quux` uses the variablename `foo` as well, it will also rename to `lIl` since the identifier already has a mapping assigned to it. This leaks the information that a variable shared its name with a variable on a different location.

The scoping of names can also be improved in the opposite direction: new names can be reused in different scopes. For example, a local variable gets renamed to `lIl` in function A. It is possible to rename a local variable for function B to `lIl` as well. The implemented version generates a unique new name for every variable regardless of scope. Reusing names can provide extra confusion and does not have a disadvantage.

Literal encoding uses a simple way to generate expressions that result in a stable pattern for any piece of static data that has been obfuscated. A more complex expression generator could be implemented that generates a lot of different types of expressions that still evaluate to the static data. This requires the deobufcator to handle a lot more cases. It also becomes a lot harder to recognize when static data is reused.

This can be made a lot more complex by including variables and data that is computed at runtime. When data from variables is involved deobfuscation becomes a lot harder. The deobfuscator needs a lot more information and more complex logic to know whether an expression contains static data or not.

Function call obfuscation can be reversed using a regular expression. It is a transformation that targets basic static analysis of Python programs and is not very strong outside of that niche. Since Python is interpreted it is possible to dynamically generate copies of a function every time it is executed. This would hide function calls a lot better since every call is a unique function.[13]

Control flow flattening has a weakness regarding the way identifiers for blocks are generated. This is discussed in the previous section. It can be removed by randomizing the value of every identifier once the function has been reconstructed.

A major improvement can be made in the way the dispatch-variable is treated. In the implemented version it is directly assigned at the end of blocks. It is possible to turn this static assignment into a dynamic expression that relies on data that is computed at runtime. This would make it more challenging to map out in which order blocks are being traversed since this value determines which block is executed next.

Section 4.2.1 discusses control flow obfuscation. Control flow flattening is a big part of this process, but it can be combined with opaque predicates and dead code generation to make it a lot more difficult to understand the control flow. Adding these techniques would be the best improvement of any of the implemented techniques.

The main issue with the obfuscator itself is the configuration method and the automatic testing functionality. To turn this obfuscator from a proof-of-concept into a full

[13]This would require further research into the potential overhead of creating a lot of function-objects at runtime.

implementation it would be better to remove the configuration file and use command-line parameters instead.

The configuration file was initially used to test and configure more parameters so during development a configuration file was helpful. In the last version only three parameters are required: input file, output path and list of transformations. Using the command-line makes more sense when only three parameters are required.

The automatic testing functionality is not necessary and can be scrapped. It was mainly used throughout development to verify that the transformations did not change the output of an obfuscated program compared to the original version. It only brings added risks due to the feature not being sandboxed or secured while external code is being executed.

The obfuscator is designed to take a single source file and produce a single source file. Since most serious projects consist of multiple files it makes sense to change the obfuscator to handle full projects instead of single files. As mentioned before this would require some changes to name obfuscation.

A lot of the listed improvements are for elements that did not get worked out in-depth because this implementation is mainly intended as a proof-of-concept. Out of the suggested improvements, the ones for control flow flattening and literal encoding are the ones that could turn this implementation from a PoC into a full implementation.

4.5 Large Language Models and Obfuscation

Large language models such as ChatGPT[14] can be used as a tool for programming. It can generate code, suggest improvements, analyze bits of code and do a lot more. Section 2.4 touches on this by showing how it can be used to help with the reverse engineering process. It therefore makes sense to look how it interacts with obfuscation.

All ChatGPT interactions in this section were done using separate sessions per script or interaction. Some full of the transcripts are present in Appendix B, these are referenced when they are discussed. Script containing the following algorithms were used for testing: testing the *collatz* conjecture for a given input, calculating *prime* numbers, calculating a specific number in the *fibonacci* sequence and *quicksort*.

A possible approach to use ChatGPT with obfuscation is to use it as a tool to analyze obfuscated code. This has been tested by copying the obfuscated code of a few scripts together with the instruction `Analyze this Python code.` as a prompt. The full responses for each script can be found at listings 27, 28, 29 and 30.

The first notable observation is that the algorithms for calculating the Collatz conjecture was correctly identified. An algorithm for calculating prime numbers and an implementation of quicksort were not identified at all. The algorithm for calculating the Fibonacci sequence gets mentioned as `...deals with Fibonacci numbers and likely prints some information related to them`. It was identified because the string "fibonacci" is present in encoded form, which ChatGPT managed to decode.

Listing 31 is the response for the same program and prompt except for the string "fibonacci" which is no longer present in any form. This time it is unable to determine which algorithm is being performed. When prompted with extra information `Perhaps it calculates a sequence?` it responds with `...mention of Fibonacci-like operations...` and `...perhaps related to Fibonacci or a similar pattern`.

This means that with the basic prompt, only one out of the four algorithms was recognized. This was the Collatz conjecture, the least complex algorithm out of the four. It has a very

[14] `https://chat.openai.com/` - GPT3 August 3 2023 version was used for experiments

recognizable computation which is `n = 3 * n + 1`. It is possible that even this simple algorithm would not be identified if some obfuscation is applied that targets arithmetic operations.

A second notable observation is the frequent mentioning of which techniques are applied without being able to analyze them. The following snippets are some examples:

- `The code contains various 'eval' statements that construct strings using ASCII values of characters and then evaluate those strings. This dynamic evaluation makes the code harder to read and understand.` (Listing 27
- `The ASCII values of characters are used to create strings that are then evaluated to execute certain code snippets. This technique is used to obfuscate the actual logic of the program.` (Listing 28)
- `Within the 'l(ll)' function, there are multiple cases that evaluate and manipulate 'lI' to control the flow of execution. There is also an 'eval' function used to execute dynamically generated code based on ASCII values of characters.` (Listing 29)

Listing 32 shows a more succesful approach in which the prompt `Make a deobfuscated version of this Python program.` was used. Although the resulting code is not entirely correct, ChatGPT was able to revert the literal encoding and function call obfuscation. A prompt asking to analyze the deobfuscated code resulted in a correct identification of the fibonacci sequence.

A further prompt to fix the deobfuscated code (which was partially broken) was submitted. This generated a clean, working implementation of the algorithm which closely resembles the original code. This means that the control flow flattening and name obfuscation were also deobfuscated.

It is unclear how much of the deobfuscation is possible because the use examples were simple scripts with known algorithms. For example, the last paragraph mentions that the control flow flattening was deobfuscated. It is possible that this worked because the algorithm for generating the Fibonacci sequence was identified. A lot more extensive testing for analyzing obfuscated programs and deobfuscating programs would be required to be able to draw definite conclusions.

Another interaction that has been tested is asking ChatGPT to obfuscate a piece of Python code. Listing 21 shows the final result. The deobfuscation was started by prompting the source code of fibonacci.py with the command `Obfuscate this program`. This applied some basic name obfuscation to the program but the resulting source code contained a small mistake that resulted in an error. By giving this error as a prompt an updated version was generated which properly executed and had the same output as the obfuscated version.

By adding the prompt `Can you also hide the static strings?` the version shown in Listing 21 was generated. There were multiple attempts to obfuscate the output more, with prompts such as `Can you obfuscate this more?` and `Can you also flatten the control flow?`. These all resulted in code that would not execute, even after adding prompts to fix that code.

The next major public iteration of ChatGPT, GPT4, is designed to have code interpreters (including one for Python) integrated into the application that interacts with their model. It also supports adding extra plugins to the model and much more. It is possible that these changes allow for better experimentation between ChatGPT and code obfuscation. It can be concluded that there is a lot of potential for interesting research into the interactions between ChatGPT (or other large language models) and code obfuscation.

```python
import sys as _0x1
_0x1.set_int_max_str_digits(100000)
n = 300000

def _0x2(_0x3):
    if _0x3 <= 2:
        return 1
    _0x4, _0x5 = 1, 1
    for _ in range(3, _0x3 + 1):
        _0x4, _0x5 = _0x5, _0x4 + _0x5
    return _0x5

def _0x6():
    _0x7 = _0x2(n)
    _0x1.stdout.write(chr(84) + chr(104) + chr(101) + " " + str(n) + chr(116)
    ↪   + chr(104) + " " + chr(70) + chr(105) + chr(98) + chr(111) + chr(110)
    ↪   + chr(97) + chr(99) + chr(99) + chr(105) + chr(32) + chr(110) +
    ↪   chr(117) + chr(109) + chr(98) + chr(101) + chr(114) + " " + chr(105)
    ↪   + chr(115) + ": " + str(_0x7) + "\n")

_0x6()
```

Listing 21: fibonnaci.py obfuscated by ChatGPT

Chapter 5

Evaluating obfuscation techniques

The previous chapters explore several obfuscation transformations and techniques. These chapters mainly focus on the technical execution and effects of these transformations. They don't discuss the actual efficacy and cost of a given transformation. This chapter introduces a set of metrics which can be used to evaluate such obfuscation techniques.

These metrics are applied to a self-made implementation of some obfuscation techniques to evaluate their effects. The results of this evaluation are used to answer two of the research questions (2 and 3) posed at the start of this thesis.

5.1 Existing Work

There already exists a lot of literature which can be consulted to decide on an evaluation strategy. Section 5.1.1 looks into some literature on obfuscation and the evaluation of obfuscation techniques. Section 5.1.2 looks into some of the software complexity metrics that are frequently encountered in the literature about evaluating software obfuscation.

5.1.1 Evaluating Obfuscation Techniques

[Ceccato et al., 2015] have researched the effect of code obfuscation on java code. In section 2 of their paper, a whole slew of publications surrounding the evaluation of code obfuscation is listed. This includes both evaluation by metrics surrounding program properties as well as evaluation by experimenting with human actors. This paper (or any other paper that contains a similar list of publications such as [Viticchié et al., 2016]) provides a good entry point to assess the different evaluation methods that are used throughout different approaches and use-cases.

However, a common theme in all these publications is the lack of common standard or approach. A lot of different metrics and approaches are used by different authors. There are some frequent commonalities, so these are explored in more detail to get a grasp on what approaches for evaluating obfuscation transformations are viable.

One of the oldest but most frequently referenced publications is by [Collberg et al., 1997]. It provides an approach that is based on metrics and program properties to analyze and quantify obfuscation transformations. This paper gets explored in more detail in section 5.2, since it is incorporated in the evaluation methodology that is used in section 5.4.3.

A paper by [Anckaert et al., 2007] is also discussed in section 5.3. It is one of the most popular meta-level publications surrounding program obfuscation. It attempts to lay the foundations of a framework that can be used to quantify and evaluate obfuscation transformations. However, the co-author of this paper has stated [Basile et al., 2023] that there is still no widely accepted set of metrics that should be used to evaluate obfuscation techniques. This paper has also been used as a reference point when deciding on the evaluation methodology used in section 5.4.3.

5.1.2 Software Complexity Metrics

A recurring element in the domain of evaluating obfuscation techniques is the use of *software complexity metrics*. These metrics are quantifiable properties of a program that can be used to indicate the complexity of a program. They are typically used by software engineers to assess the complexity of a project. Having proper insights into the complexity helps with the general development process, e.g., they can be used as an indicator that a component has become too complex and requires refactoring.

These metrics can also be used to help with the evaluation of obfuscation techniques. A set of chosen metrics can be extracted from a program before obfuscation and after obfuscation. The differences between these sets of metrics can then be used to prove some property about the obfuscation. A change or lack of change in the metrics can indicate the effectiveness of the obfuscation depending on the metric used.

Two of the most popular metrics are McCabe's cyclomatic complexity and Halstead's length. Both of these metrics are expressed in a number that increases when the program's complexity increases. These metrics can therefore indicate when an obfuscation technique has increased the complexity of a program, which is the goal of program obfuscation.

[McCabe, 1976] defines the cyclomatic complexity of a program, a metric based on the number of paths that can be traversed through the program. This metric uses the control flow graph (defined in chapter 4) of a program or routine in a program.

It is defined as the following:

$$V(G) = E - N + 2P$$

where $V(G)$ is the cyclomatic complexity of a control flow graph, E is the number of edges, N is the number of nodes, and P is the number of connected components (exit nodes). A higher cyclomatic complexity indicates a more complex control flow graph.

[Halstead, 1977] defines a set of metrics based on the operands and operators[1] that are present in a program. η_1 is the unique operator count, η_2 is the unique operand count, N_1 is the total number of operators and N_2 is the total number of operands. Multiple metrics can be calculated using these parameters.

Three commonly used metrics are Halstead's length N, Halstead's calculated length $\hat{N}$ and Halstead's effort E. :

$$N = N1 + N2$$

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

$$E = D \cdot V$$

$$\text{with } V = N \cdot \log_2 \eta \text{ and } D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$$

The length metrics are reflections of the size of a program. The effort metric attempts to express the amount of mental effort required to understand a program. Both the calculated length and effort have a logarithmic relation towards the total number of operators and operands. This expresses how the complexity of a program does not grow linearly as the diversity of operators and operands decreases.

[1] Operators take zero or more operands. E.g., the operator for `a + b` is `+` and the operands are `a` and `b`.

Halstead's effort is defined using Halstead's Difficulty (D) and Halstead's Volume (V). The difficulty quantifies how difficult it is for a developer to understand a program. The volume is a more complex expression of the length of the program. The resulting metric, Halstead's effort, is a broader metric that better expresses how much effort is required to understand and maintain a piece of code since it factors in the volume on top of the difficulty.

These metrics got defined in respectively 1976 and 1977 and they are still relevant as complexity metrics to this day. A lot of research listed in the existing work section (5.1.1) uses these metrics directly or references them as relevant techniques. It is therefore reasonable to recognize these metrics as well-established tools that can provide help with the evaluation of obfuscation techniques.

5.2 Metrics used by Collberg et al.

[Collberg et al., 1997] define a set of metrics that can be used to quantify and evaluate the effectiveness of an obfuscation transformation. These metrics are combined to quantify both effectiveness on human attackers as well as automated attacks. They also take the potential overhead generated by the obfuscation in account. The combination of these different angles allows a flexible and clear evaluation.

The three metrics used are *potency, resilience* and *cost*. *Potency* entails the difficulty for a human reader to understand a program. A higher potency means that a human reader has a harder time understanding the program. *Resilience* expresses a program's resistance to automated attacks that attempt to undo the obfuscation. The *cost* quantifies the overhead incurred by applying the obfuscation.

These metrics are combined into the overall measure of quality of an obfuscation transformation. This results in a set of three parameters (potency, resilience, cost) which describe the quality of a transformation. This section introduces these metrics as well as the proper way of using them.

5.2.1 Potency

Potency expresses the effect of a transformation on a human reader. A transformation is potent if it confuses a human reader. A higher potency implies that more time is needed to analyze and understand the program. Software complexity metrics are used to quantify potency. A piece of software with a higher complexity is more difficult to understand.

The following metrics (μ_1 to μ_7) are chosen to evaluate potency:

$\boldsymbol{\mu_1}$ **Halstead's Length** is defined in 5.1.2. It counts the number operators and operands in a program.

$\boldsymbol{\mu_2}$ **Cyclomatic Complexity** is defined in 5.1.2. It uses the control flow graph of a program or routine to calculate a number that expresses the complexity of this graph.

$\boldsymbol{\mu_3}$ **Nesting Complexity** is a metric created by [Harrison and Magel, 1981] to address the flaws of the previous metrics (μ_1 and μ_2). These metrics don't take the added complexity of nested blocks into account.

For example, a program that has no nesting at all and is simply a sequence of operations can have the same Halstead's Length as a program with multiple levels of nesting. It can also have the same cyclomatic complexity as a program with less nesting, even though the nesting makes the program potentially more complex. This measure is therefore used to express this level of nesting as an added measure.

$\boldsymbol{\mu_4}$ **Data Flow Complexity** [Oviedo, 1980][2] expresses the number of times variables are used

[2] This paper cannot be found online. This is a third hand description.

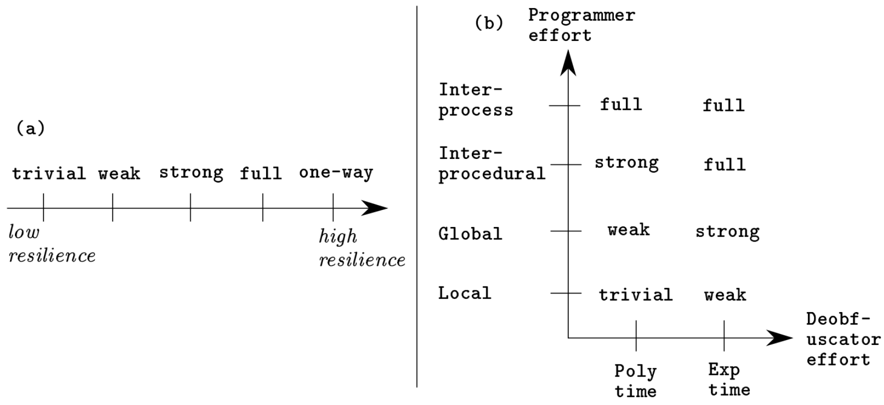

Figure 5.1: Resilience Graphs by [Collberg et al., 1997] (Figure 7)

in multiple basic blocks. It increases when variables are referenced in more blocks.

μ_5 **Fan-in/out Complexity** [Henry and Kafura, 1981] indicates the number of parameters of a component, as well as the number of data structures read or updated by this component.

μ_6 **Data Structure Complexity** [Munson and Kohshgoftaar, 1993] increases, as the name implies, when a data structure becomes more complex. For example, when an array gets more dimension or a class gets more (or more complex) member variables.

μ_7 **OO Metrics** [Chidamber and Kemerer, 1994] are a full set of metrics on their own. They express a lot of OO properties such as the number of methods, the depth of the inheritance tree, coupling, and more.

The goal of an obfuscating transformation is to increase the complexity of some of these metrics. For example, adding predicates and nodes to a program can make the control flow graph more complex. This will be reflected by an increase in μ_2 and μ_3. Collberg classifies the potency of transformations on a three-point scale (*low*, *medium*, *high*).

A formula is given (in definition 2 of the paper) which expresses whether a transformation is considered potent or not. There is however no exact mapping or explanation given for how the scale is used and if/how it relates to this formula:

$T_{\text{pot}}(P) = \frac{E(P')}{E(P)} - 1$

E(P) expresses the complexity of a program P (using one of the 7 metrics). T is a transformation which turns program P into target P'. The transformation T is potent if $T_{\text{pot}} > 0$.

5.2.2 Resilience

An increased value for any complexity metric cannot be directly correlated with an increase in practical obfuscation efficacy. A transformation can artificially inflate these metrics without adding any real complexity to the program. This is why Collberg defined the concept of *resilience*.

Resilience expresses how well a transformation can be “attacked”[3] by an automatic deobfuscator. This is split into programmer effort and deobufscator effort. Programmer effort describes how much time is needed for a programmer to write a deobfuscator that reduces the potency of transformation. Deobfuscator effort describes how much execution time and space is required by the deobfuscator to reduce the potency.

[3]In this case an attack would mean a transformation that takes the obfuscated program and creates a new version in which the obfuscation has a lower potency.

Collberg classifies the resilience of transformations on a five-point scale (*trivial*, *weak*, *strong*, *full*, *one-way*) as shown in Figure 5.1 (a). Furthermore, the scope of the transformation is used in combination with the time an automatic obfuscator would need to deobfuscate the transformation as shown in Figure 5.1 (b).

The scope of a transformation is considered to be one of the following:

- *Local* if it affects a single basic block
- *Global* if it affects the entire control flow graph
- *Inter-procedural* if it affects data-flow between procedures
- *Inter-process* if it affects interactions between independently executing threads

A transformation that has one-way resilience is considered the strongest. It applies to transformations for which the original program cannot be reconstructed using the obfuscated version. These are typically transformations that remove information that does not impact the program execution but does impact the human analyzing the program (because it removes elements such as formatting, comments, identifiers, etc.).

5.2.3 Cost & Applying metrics

Cost is the last component used to describe the quality of a transformation. The extra execution time and space required to execute the obfuscated program is used to determine this metric. It is expressed in a four-point scale: *free*, *cheap*, *costly* and *dear*. It is determined if the obfuscated program uses more resources in the following orders of magnitude: $O(1)$, $O(n)$, $O(n^p)$ with $p > 1$ and $O(2^n)$ more resources.

As mentioned in the introduction of this section, the three properties potency, resilience and cost are combined into a set of three values. These values can be determined for any given obfuscation transformation, and can then be used by developers to determine whether the quality of the transformation lines up with their goals and restrictions (in terms of cost). Developers can then decide whether or not they want to apply that transformation to their software.

[Canavese et al., 2017] have experimented with using neural networks to determine the potency of a transformation. In the classical approach, the program gets obfuscated and the chosen metrics are compared with the version of the program before obfuscation. The change of these metrics is then used to determine the potency.

This approach of using neural networks is able to estimate the potency of transformation before actually applying the transformation. This predicted potency can then be used to decide on which transformations should be applied to this application. It struggles with predictions for multiple transformations due to error propagation.

The neural network approach is not entirely usable in practice due to this struggle. However, it does open the door for better approaches to choosing transformations built on the original framework by Collberg. This shows that this methodology is still considered relevant and improving for the future.

5.3 Evaluation Framework proposed by Anckaert et al.

[Anckaert et al., 2007] have the goal of working towards a standardized evaluation suite that allows for a quantified comparison between obfuscation transformations. As mentioned before, there is currently still no general accepted standard to evaluate and compare obfuscation transformations. This means that their approach has not become a new standard, but their contribution still has a noticeable impact on obfuscation research. A lot of literature references this framework and uses evaluation metrics and methodologies that are partly based on it.

5.3.1 Metrics & Methodology

Anckaert et al. start from the idea that every program has the following fundamental properties: *instructions, control flow, data flow* and *data*. To understand a program, all four of these properties need to be analyzed. For example, if all the instructions of a program are known, it is still not known in which order they will be executed; that information can be found in the control flow. Even if both these properties are known and understood, it is still possible that a lack of understanding of the data flow or a lack of knowledge on the data itself makes it hard to understand the program. For example, it may be uncertain why data is being moved, or what the actual data entails (e.g. because it is hashed).

The suggested methodology for evaluating obfuscation techniques is to look at software complexity metrics that impact these program properties. If a software complexity metric accurately reflects changes in one or more of these fundamental properties, it can be used to evaluate obfuscation transformations. Within their paper they also applied this approach to some actual programs, for which the following software complexity metrics were used: *instruction count, cyclomatic number* and *knot count* (which measures the number of crossing arrows in a control flow graph) .

In their experiment to compute these metrics for a set of programs[4], a dynamic trace of each program was used. This is an approach in which the program is executed, during which dynamic analysis is used to observe which parts of the program were actually executed. A reconstruction of the program is then be created with only the executed parts, and the metrics are applied to this reconstruction.

The usage of the dynamic traces allows the evaluation to focus on the parts of the program that were executed. It will result in equal or lower observed complexity when compared to static analysis, since any unused parts of the program are omitted. If an attacker would try to reverse engineer a program, they would likely also use dynamic analysis to improve their analysis, so this new approach is more realistic for practical evaluations.

5.3.2 Comparison with Collberg

Anckaert focuses on four fundamental program properties to decide on what metrics can be used to evaluate obfuscation techniques. Collberg reasons that any type of metric used in the field of software complexity analysis can be used to evaluate obfuscation techniques, since a higher complexity implies better obfuscation. This second approach seems less thought out, but in practice the list of actual metrics will have quite some overlap.

A big difference between these methodologies is that Anckaert uses a dynamic trace of a program to analyse the obfuscation, rather than using static analysis like Collberg. The use of dynamic analysis allows for a more realistic evaluation of a program, since reverse engineers attempting to analyze a program will likely use this method too. It is not necessarily bad to use static analysis like Collberg does, but dynamic analysis can give a more realistic evaluation of obfuscation techniques.

Anckaert has a practical experiment for which instruction count, cyclomatic number and knot count are chosen as metrics. Collberg has a list of seven metrics that can be considered when evaluating obfuscation transformations. This list has some overlap with Anckaert's metrics: the cyclomatic complexity is used by both authors, and the instruction count is somewhat related to Halstead's metrics. The other metrics suggested by Collberg also impact one or more of the fundamental program properties, for example, nesting complexity expresses a metric that

[4] Their tested programs were taking from the SPECint2000 suite. This is a set of C programs that is used to benchmark the integer processing power of CPUs.

is related to the control flow of a program. This shows that these approaches are similar when used to choose metrics.

5.4 Experiments

A set of metrics is chosen based on the publications discussed in the previous two sections. These metrics are used in a practical experiment using a Python obfuscator and some Python programs. The results of this experiment are discussed in the next section of this chapter to answer the second and third research question posed in the introduction (1.2).

5.4.1 Chosen Metrics

The following metrics are chosen to analyse:

- Halstead's Calculated Length
- Halstead's Effort
- McCabe's Complexity (Cyclomatic Complexity)
- Execution time

The first three metrics are software complexity metrics which are used by Collberg, of which McCabe's complexity is also directly used by Anckaert. The metric "instruction count" by Anckaert is not the same metric as Halstead's metrics but it is also influenced by the program size. The last metric, execution time, is used by both Collberg and Anckaert. It therefore makes sense to include these four metrics that have been used by both approaches.

Halstead's calculated length is used over the "raw" length. This calculated length is a better estimator of the complexity of a program as it grows logarithmically with the size of the program rather than linearly. This logarithmic growth expresses the perceived complexity better as explained in 5.1.2.

The software complexity metrics on their own are not very comprehensible. They are best used to compare different programs to see the relative difference in the metrics. In the case for evaluating obfuscation, the change of metrics between the original program and the obfuscated version can be used to observe the change of complexity.

All these metrics can be determined for Python programs in the same way as they would be obtained for any other programming language. Some of the practical aspects are different per programming language, but these are all metrics that are related to high level operations and structures.

For example, in C there is a `GOTO` statement which will jump the flow of execution from the current location to a specified label. This is a method in which the control flow graph of a C program can be changed, which in turn will affect McCabe's complexity. This statement does not exist in Python, but Python does have a `with` statement which allows for easier exception handling. This is a control flow statement that is not present in C and also affects McCabe's complexity.

5.4.2 Implementation

The chosen metrics are tested in practice with an implementation of a proof-of-concept obfuscator. This section provides a high level description over this implementation. This should provide enough context to understand the experiment and interpret the results. A more detailed description of this implementation is discussed in 4.4.

A full description of the implemented transformations including examples can be found at 4.4.2. This includes examples of the results of every technique separately and all of them layered together. The following transformations are implemented:

- Name Obfuscation (hides all information that does not impact execution e.g. names, comments, docstrings, ...)
- Literal Encoding (turns constant booleans and strings into an expression that evaluates to the value at runtime)
- Function Call Obfuscation (hides functions by calling them through eval instead of using direct calls)
- Control Flow Flattening (restructures each function with a dispatch-structure that manages the control flow instead of using the control flow keywords `if, for, while`)

The obfuscator transforms a file containing Python code into a file with an obfuscated version of this code. It does not allow the obfuscation of multiple files at a time, but this can be achieved by separately obfuscating every file. The transformations can be applied together or alone in an order which is also provided as input for the obfuscator.

5.4.3 Methodology

The chosen software complexity metrics are be obtained through static analysis of the Python source code of a program. The execution time can be measured by executing the program. All of these metrics are measured using existing tools that are popular within the Python community.

Radon[5] is used to compute the software complexity metrics. It is described by the developers as the following: "Radon is a Python tool that computes various metrics from the source code." These various metrics include all of Halstead's metrics and McCabe's complexity. Radon can therefore be used to measure the three chosen software complexity metrics. It also computes some extra metrics based on line counts and similar properties. These are discarded, together with the rest of Halstead's metrics that aren't used.

Measuring the execution time is done using the built-in timeit[6] module. This module eases the process of taking a timestamp before and after the execution of a piece of Python code. As with any piece of software, the execution time depends on the circumstances in which the program is run. The operating system, background processes, the device's specifications, and other factors might impact the performance. This can possibly invalidate any measurements, so a measuring strategy is required.

The impact of influencing elements is limited by using a small automation framework that runs all the tests together. Unnecessary processes and connections are closed to minimize random variations and every script is executed five times in total. The average of these five executions is used to compare the execution speed of the scripts.[7] This timing method is not perfectly accurate but it does provide good insights on differences in execution time, even for small differences.

Multiple versions of each program are tested to measure results: the original version and multiple obfuscated versions. This includes every transformation applied separately as well as combinations of transformations. The change of the metrics between the original version and the obfuscated versions is then analyzed to determine the impact of the transformations.

Python also has some built-in modules for more complex analyses and profiling. This includes trace, profile and cProfile. These modules allow for a much more in-depth analysis of the execution of a program, including which lines get executed at what times, the time spent on certain functions, and more. This comes at the cost of some overhead when using these tools, which makes them impractical for accurate time measurements within this context.

[5] `https://pypi.org/project/radon/`

[6] `https://docs.python.org/3/library/timeit.html`

[7] The worst case standard deviation for the performed tests was 0.64%. This deviation went in a range of 2% to 5% when tests were performed without limiting external factors.

5.5 Discussion on Singular Transformations

Three small scripts have been created as minimal examples to analyse the metrics. These are a script calculating a high fibonacci number (fibonacci.py) , a script calculating prime numbers (prime.py) and a script that sorts a lot of lists using quicksort (sort.py). Listing 26 in appendix B contains a full example of sort.py with all obfuscation transformations applied. Data on these scripts has been gathered using the methodology described in the previous section.

As is mentioned in the previous section, there are likely some minor random fluctuations in the data that tracks the execution times of these scripts. The main goal of this discussion is to observe and explain the general order of magnitude of the changes in execution time, so changes of 3% or less are considered negligible throughout the discussion of this data.

This section discusses the data gathered with experiments that focus on having a singular transformation applied at a time. This makes it possible to isolate and analyze the effects of every transformation on the used metrics. The data on experiments with multiple transformations gets discussed in the next section 5.6.

Filename	Original	Name	Name (%)	Literal	Literal (%)
fibonacci.py	1.169	1.181	101%	1.181	101%
prime.py	0.925	0.923	100%	0.944	102%
sort.py	0.66	0.649	98%	0.648	98%

Table 5.1: Execution time (seconds) and the relative change for name obfuscation and literal encoding

Filename	Original	Call	Call (%)	Flatten	Flatten (%)
fibonacci.py	1.169	1.174	100%	1.301	111%
prime.py	0.925	9.386	1015%	3.746	405%
sort.py	0.66	6.812	1032%	0.675	102%

Table 5.2: Execution time (seconds) and the relative change for function call obfuscation and control flow flattening

Tables 5.1 and 5.2 contain the execution times of the three scripts when a single transformation is applied. Both tables also contain a column "Original", which contains the execution time for the scripts without any obfuscation applied. There is a column for every transformation with the absolute time that the obfuscated version took to execute, and a column that compares this value with the execution time of the script without obfuscation.

Table 5.3 shows an overview of the measured complexity metrics when a single transformation is used. It contains the metrics for the original scripts and all the created variants with a singular obfuscation transformations applied. It also has three columns showing the change (Δ) of each metric compared to the original script. Note that the "length" is Halstead's calculated length as explained in the previous section that goes over the metrics and methodology.

Table 5.4 lists the quality of each transformation using the framework proposed by [Collberg et al., 1997] can be applied to these transformations. Name Obfuscation was already evaluated by this publication, the other entries are based upon the gathered data.

5.5.1 Name Obfuscation

Name Obfuscation removes all names of variables, classes, and similar. It also removes all comments and docstrings. Removing this information only impacts the person reading the code. This transformation does not change the structure or execution of the program. It is

Transformation	**Length**	**Δ Length**	**Effort**	**Δ Effort**	**McCabe**	**Δ McCabe**
			fibonacci.py			
Original	40	X	73	X	4	X
Name	40	0%	73	0%	4	0%
Literal	40	0%	73	0%	4	0%
Call	10	-75%	16	-78%	4	0%
Flatten	72	80%	346	374%	14	250%
			prime.py			
Original	78	X	624	X	9	X
Name	78	0%	624	0%	9	0%
Literal	112	44%	812	30%	9	0%
Call	64	-18%	376	-40%	9	0%
Flatten	109	40%	1403	125%	30	233%
			sort.py			
Original	91	X	555	X	13	X
Name	91	0%	555	0%	13	0%
Literal	91	0%	555	0%	13	0%
Call	49	-46%	353	-36%	13	0%
Flatten	140	54%	1482	167%	27	108%

Table 5.3: Complexity metrics for singular transformations

Transformation	**Potency**	**Resilience**	**Cost**
Name Obfuscation	High	One-Way	Free
Encoding Literals	?	Trivial	Cheap
Function Call Obfuscation	Low	Trivial	Dear
Control Flow Flattening	?	Weak	Costly

Table 5.4: Transformations applied to the quality framework by Collberg

thus expected that the software complexity metrics do not change and that that the execution time is not impacted.

This expectation matches up with the practical results. There is no change in any of the tables containing complexity metrics whenever only name obfuscation is applied. Table 5.1 shows that the difference in execution times with or without name obfuscation only differs around 10 milliseconds. There is even a 2% speedup for sort.py, which will be considered negligible. The expectation that the execution time is impacted minimally is therefore also correct.

Collberg's taxonomy lists name obfuscation as having high potency[8] with a free cost and one-way resilience. As discussed in section 4.4.3 there has been research by [Lacomis et al., 2019] to reconstruct identifiers using neural networks. This is the only approach that can potentially help to deobfuscate this transformation. Name obfuscation is a potent transformation that should be applied to any language that (partially) retains information that this transformation can remove since it has no negative impact on the execution times.

5.5.2 Literal Encoding

Literal Encoding has no impact on the complexity metrics for fibonacci.py and sort.py. It has an effect on these scripts but not on the metrics because of the way the encoding is applied. It

[8]It is listed as 2 separate transformations, of which one has high potency (remove comments) and the other has medium potency (scramble identifiers).

does however increase the length and effort metrics for prime.py by respectively 44% and 30%. This makes sense because Halstead's metrics depend on the number of operators and operands. This number increases due to this transformation when boolean literals are encoded, which only happens in prime.py.

For example, the original string `'Sorted list '` gets changed to the expression `''.join(map(chr, [83, 111, 114, 116, 101, 100, 32, 108, 105, 115, 116, 32]))`. According to Halstead's metrics the complexity of the program is identical whether the strings are encoded or not. Only the boolean encoding counts as an increase in complexity because it increases the number of operands. For example `True` is encoded to the expression `(()==())` which gets counted as one operator and two operands.

McCabe's complexity is not impacted at all by this transformation. This can be explained by the fact that a literal being replaced by an expression does not change the CFG. There is also no noticeable impact on the execution time. This can partially be explained due to the fact that there are not many literals being used in these scripts, although this is common for most programs.[9]

The script prime.py is the only script in which an encoded literal is very frequently encountered during normal execution in the function `is_prime(num)`. Even with this literal being encountered a lot the execution times for this script only increased by 2%, which is considered negligible.

The resilience of literal encoding is considered trivial, as is displayed in table 5.4. This matches with the potential deobfuscator that gets discussed in section 4.4.3. It is hard to conclude anything about the actual potency of this transformation because the complexity metrics are not affected. Intuition indicates that the hiding of static data, especially strings, makes a program a lot harder to understand for a human. This requires experimenting with a group of human readers to test this hypothesis.

5.5.3 Function Call Obfuscation

Function Call Obfuscation is the transformation with the most notable results. Halstead's length and effort went down in value for all three scripts (in table 5.2). This would imply that the program has become less complex after applying the transformation, which seems counterproductive for an obfuscating transformation. The obfuscated versions of prime.py and sort.py took 10 times the normal execution time for the obfuscated variants to finish executing. Meanwhile ibonacci.py maintained the same execution time.

The decrease of Halstead's metrics can be explained by looking at what the transformation actually does: it replaces a normal function call with a dynamic call using `eval` and a string. If the original call contained an expression, then this expression would now be replaced by a single string. This can potentially remove operators and operands from the program, which in turn lowers the values for Halstead's metrics.

The lack of slowdown for fibonacci.py can be explained by its implementation: most of the execution is spent in a loop that does not call any functions. The loop-iterator itself is initialized with `eval`, from the code `for _ in eval('range(3, n + 1)')`. This will only execute the eval-statement once, which is negligible in terms of execution time (since it only adds a single constant expression).

The execution time being 10 times longer for sort.py and prime.py is an extreme result. It is expected that using `eval` over a direct function call incurs some overhead, but this experiment proves that this overhead can be significant. There are a lot of function calls executed during the execution of these two scripts. It can be concluded that programs with a high volume of function calls will experience an extreme slowdown when using function call obfuscation.

[9] It is unusual for a program to rely on a lot of literals being processed. Having a lot of literals is usually considered to be a code smell. [Taibi et al., 2017]

This results in a quality of low potency, trivial resilience and dear cost according to Collberg's evaluation method. This would imply that this is a very weak transformation with a high cost, which is most likely undesirable. However, it can be used to invalidate static analysis.

```
vulture sort_fc.py
sort_fc.py:2: unused variable 'num_lists' (60% confidence)
sort_fc.py:4: unused function 'quicksort' (60% confidence)
sort_fc.py:8: unused variable 'left' (60% confidence)
sort_fc.py:10: unused variable 'right' (60% confidence)
sort_fc.py:13: unused function 'generate_random_list' (60% confidence)
sort_fc.py:13: unused variable 'length' (100% confidence)
sort_fc.py:16: unused function 'main' (60% confidence)
sort_fc.py:17: unused variable 'list_length' (60% confidence)
sort_fc.py:20: unused variable 'random_list' (60% confidence)
sort_fc.py:23: unused variable 'i' (60% confidence)
```

Listing 22: Vulture output for sort.py with function call obfuscation

This transformation has also been tested with Vulture[10]. Vulture is a static Python analyzer that is used to find unused code. Listing 22 shows the output for the script sort.py that has function call obfuscation (and no other transformations) applied. There are no unused variables and functions in this script, yet but seven variables and all three functions are still marked as unused with 60% confidence. This is the same confidence that variables and functions have when they are actually unused. This means that this static analysis tool generated 10 false positives that are indistinguishable from true positives.

```
python -m trace -l sort_functioncall.py
...
filename: sort_functioncall.py, modulename: sort_functioncall, funcname:
↪  generate_random_list
filename: sort_functioncall.py, modulename: sort_functioncall, funcname: main
filename: sort_functioncall.py, modulename: sort_functioncall, funcname:
↪  quicksort
```

Listing 23: End of the trace output showing called functions

Although Vulture is a tool that is used to determine unused code, it is symbolic for most static analysis tools. Any code called using `eval` statements only gets executed at runtime, which means that automated static analysis is likely to be incorrect. Any form of dynamic analysis is still able to recognize what functions are called, for example when using the built-in module `trace` to analyze the execution of a program. Listing 23 shows that this module still correctly outputs the three defined functions as used, as expected.[11]

This transformation has been shown to have the potential to create a significant increase in execution time. It lowers the values for Halstead's metrics which in theory means that the program is less complex. Its main quality is the fact that it is able to circumvent normal static analysis that gathers information about which functions are called. This makes it a very situational transformation that should only be used when the overhead has been measured and is within the acceptable range for the developer using it.

[10] `https://github.com/jendrikseipp/vulture`

[11] Only the final lines out the trace output are shown. The omitted output mostly contains built-in function calls, which are irrelevant to the discussion at hand.

5.5.4 Control Flow Flattening

Control Flow Flattening is the only transformation that has an impact on McCabe's complexity, which consistently gets increased. It also increases Halstead's Length and Effort. The execution time increases in varying degrees for the three scripts.

As explained in section 5.1.2, McCabe's cyclomatic complexity expresses the complexity of a control flow graph. This transformation will always generate more basic blocks (nodes) and more movement between basic blocks (edges) than the original version. It approximately doubles for sort.py and more than triples for fibonacci.py and prime.py (in table 5.3. This means that the complexity of the control flow graph has increased.

The `match` and `case` statements used within this transformation have been introduced in Python3.10. Existing tools used to visualize the control flow graph don't support these statements yet, so a visual comparison has not been generated.

Every handled case connects back to the match-statement, and can also have an if-statement present in the case-block. This is what causes the cyclomatic complexity to increase, since every case is counted as a node and an edge, combined with the if-statement in the cases counting as extra nodes and edges.

This transformation also impacts Halstead's metrics, which consistently increase in value after flattening has been applied. Halstead's effort always at least doubles compared to the original version, with fibonacci.py even having almost five times the original value for effort measured (table 5.3). All of the original statements are included in the transformed version, with a lot of added statements that are required to properly manage the control flow. It therefore makes sense that this transformation always increases the complexity metrics.

The changes in execution times are inconsistent. The 2% change for sort.py is negligible. A 10% increase in execution time takes place for fibonacci.py, which is significant enough to observe as an impactful increase. The flattened version of prime.py took four times as long to execute as the unobfuscated version. This is a slowndown which is an order of magnitude bigger than the 10% slowdown of fibonacci.py.

This difference in slowdown is likely explained by the difference in the way instructions are distributed between the programs. For prime.py, a lot of time is spent moving between control flow statements and function calls with a lot of simple arithmetic. This means that any overhead to the control flow structures is going to be amplified. For sort.py this is not the case, because list comprehensions are used which avoid certain control flow changes.

For example, `left = [x for x in arr if x < pivot]` cis a list comprehension that is part of sort.py. This will not get flattened since a list comprehension is not treated as a control flow structure even though it contains an iterator using a structure resembling a for-loop. This results in most of the execution time being determined by these list comprehensions, and they are not impacted by the flattening of the control flow.

It is also possible that the `match` statement that is used as a dispatch-structure for the control flow flattening is optimized after it is traversed a certain number of times.[12] Fibonacci.py stays within the same match statement while prime.py continually switched between functions with different match statements. This could explain why fibonacci.py only has a 10% slowdown even though a lot of instructions are executed through this dispatch structure.

There also some extra overhead when there are more cases that need to be handled in the match-statement. There are a lot more cases in prime.py so this partially explains some of the experienced slowdown. A lot of extra testing would be required to determine the exact impact of the number of cases.

[12] Hot path optimization [Aycock, 2003] is a common optimization technique for interpreters. In this case, this would mean changing the lookup order of the match-cases based on the most frequently encountered cases.

When applying Collberg's quality evaluation (table 5.4) this transformation gets classified as having a weak resilience since the scope of the transformation stays within the control flow graph of every function. Although the potency formula is not known, it is likely that the potency of this transformation would be classified as "high" since multiple complexity metrics have a significant increase in value after this transformation.

This transformation has provable potency and a varying cost, which was cheap for two out of the three tested scripts. As [BinShamlan et al., 2019] have shown, this type of transformation does have an observable effect on human readers. It is also discussed in broader scope in section 4.2.1 since it is a relatively popular obfuscation technique. The results show that this technique has a provable effect when applied to Python, which had not been done before.

5.6 Discussion on Combined Transformations

A realistic obfuscator does not apply a singular transformation. Instead, a set of transformations is layered, which increases the difficulty for an attacker to analyze or modify a program. The order in which transformations are layered can impact the final result.

5.6.1 Order of transformations

For this implementation, the following order has to be used for optimal obfuscation: *control flow flattening*, *name obfuscation*, *function call obfuscation* and *literal encoding* as the last transformation. This is partially due to certain interactions generating invalid code and partially due to certain obfuscation transformations being strengthened by others. Whenever the combination "All" is discussed, it uses this order.

The control flow flattening is applied first. It fully restructures all functions and generates new variables in the process. This means that new names are generated that could be obfuscated by name obfuscation, so this transformation should happen before name obfuscation as an optimization. It also adds some extra function calls, so it is also better to use it before function call obfuscation.

When function call obfuscation is applied, all function names are encapsulated in a string in an `eval` statement. This means that any normal renaming that takes place after this transformation does not correctly apply to these strings, which would result in errors at runtime. It is therefore required to do any renaming (such as name obfuscation) before this transformation is applied.

Literal encoding transforms all static strings and booleans. Because function call obfuscation generates new strings, it is better for this transformation to be applied after function call obfuscation. This turns an obfuscated function call from `eval('main()')` to `eval(''.join(map(chr, [109, 97, 105, 110, 40, 41])))`, which is harder to understand for a human attacker.

5.6.2 Execution Times

Table 5.5 shows the execution times for the original scripts compared to multiple combinations of obfuscation transformations. Table 5.6 contains the relative change per combination compared to the original script.

Filename	Original	All	Call + Literal	Flatten + Call	Flatten + Literal
fibonacci.py	1.169	4.424	1.178	4.031	1.310
prime.py	0.925	154.934	10.483	145.528	3.771
sort.py	0.660	7.411	7.587	6.921	0.678

Table 5.5: Execution times of combined transformations (seconds)

Filename	All	Call + Literal	Flatten + Call	Flatten + Literal
fibonacci.py	378%	101%	345%	112%
prime.py	16758%	1134%	15741%	408%
sort.py	1123%	1150%	1049%	103%

Table 5.6: Execution times of combined transformations (relative to original)

The execution times (in table 5.6) are increased significantly for the three scripts when all transformations are applied. Fibonacci.py takes almost four times as long to execute, sort.py takes more than 10 times longer to execute, and prime.py takes over 160 times longer to execute. This means that the execution time for prime.py (in table 5.5) changed from a little under a second into over 150 seconds. This is clearly undesirable and the different combinations of transformations should be used to determine which combination causes the execution time slowdown to be amplified.

Name obfuscation on its own did not have any impact on any metrics or execution time as is discussed in section 5.5.1. That is why it was not used in any of the combinations except when all transformations are layered together.

Prime.py has the worst slowdown by far when all transformations are combined. Looking at the other combinations makes it clear the the combination of control flow flattening and function call obfuscation is the cause of this extreme slowdown, as the slowdown of this combination is similar to the slowdown of all transformations combined. For fibonacci.py this combination is also the slowest, where sort.py has a slightly slower execution when all transformations or call and literal obfuscation are used.

For prime.py the combination of control flow flattening and function call obfuscation is still an extra order of magnitude slower than for the other two scripts. These transformations already caused some notable slowdown for prime.py separately, and combining them amplified this effect. Flattening adds some extra function calls which causes extra overhead from the function call obfuscation. This means that the slowdown of flattening and function call obfuscation is multiplied with the latter being increased compared to when it is used as the only transformation.

The combinations with literal encoding (so both flatten and function call obfuscation) stay within the same order of magnitude as the original transformation without literal encoding in terms of execution times. When it is combined with control flow flattening, the execution time is practically identical to the execution time of only using control flow flattening.

The combination of literal encoding with function call obfuscation does have some increase in execution time. This makes sense because function call obfuscation generates extra literals, which get encoded afterwards. This causes some extra overhead at runtime. For example, prime.py went from 1015% to 1134% and sort.py from 1032% to 1150%. This is approximately a 10% extra slowndown when comparing the combined version to only function call obfuscation (in tables 5.2 and 5.6).

In table 5.6, it can be noticed that sort.py executes faster when all transformations are applied than when only function call obfuscation and literal encoding are applied. This makes no sense at a first glance. Even though flattening has a minimal impact on the execution time of sort.py, it should still be faster to not do all transformations, unless some unexpected interaction has taken place.

In this example, the unexpected interaction likely comes from the name obfuscation (which was assumed to have no impact for combination testing). When name obfuscation is applied, most variables names become a lot shorter than their original version. It has already been established that literal encoding causes some extra overhead.

It can reasonably be assumed that short literals can be decoded as fast or faster than longer literals. Because name obfuscation shortens variable and function names, it also shortens the literals that are created by name obfuscation. The main cause for slowdown for sort.py is by far the function call obfuscation (table 5.2), so this increase in speed is enough to make the "All" combination be faster than the combination of only function call obfuscation and literal encoding.

This unexpected result was caused by the incorrect assumption that name obfuscation has no impact when combining transformations. The impact of name obfuscation can be considered negligible for normal situations. The specific situation where a variable-name is encoded into a string that has to be decoded at runtime is probably one of the few rare edge cases where it can have an impact.

5.6.3 Complexity Metrics

Transformation	Length	Δ Length	Effort	Δ Effort	McCabe	Δ McCabe
			fibonacci.py			
Original	40	X	73	X	4	X
All	41	3%	183	151%	14	250%
Call + Literal	10	-75%	16	-78%	4	0%
Flatten + Call	41	3%	183	151%	14	250%
Flatten + Literal	72	80%	346	374%	14	250%
			prime.py			
Original	78	X	624	X	9	X
All	128	64%	1215	95%	30	233%
Call + Literal	96	23%	532	-15%	9	0%
Flatten + Call	93	19%	997	60%	30	233%
Flatten + Literal	145	56%	1650	65%	30	0%
			sort.py			
Original	91	X	555	X	13	X
All	99	9%	1141	106%	27	108%
Call + Literal	49	-46%	353	-36%	13	0%
Flatten + Call	99	9%	1141	106%	27	108%
Flatten + Literal	140	54%	1482	167%	27	108%

Table 5.7: Complexity metrics for combined transformations

The results for the metrics in table 5.7 are within the expected values for all the combinations. McCabe's complexity only gets impacted by control flow flattening so this value is either 0 or the same value as when only flattening is applied. Halstead's metrics are combinations of the metrics with singular transformations applied. Some interactions between transformations do slightly change this result, but there are no remarkable results or outliers.

As discussed in section 5.5.3, function call obfuscation decreases the value of Halstead's metrics. This is also noticeable when looking at Halstead's metrics for combined transformations. For example, the combination of all transformations does not have the highest length or effort for fibonacci.py and sort.py. This means that the decrease in the values for function call obfuscation outweighs the increase in the values of the other transformations combined.

Halstead's length is is the highest for all transformations for prime.py, but Halstead's effort is still a bit lower than its value for control flow flattening. This implies that the total number of operator and operands has increased, but their diversity is less than the diversity of the operators and operands for control flow flattening.

```
vulture sort_combo.py
sort_combo.py:2: unused variable 'll' (60% confidence)
sort_combo.py:4: unused function 'l' (60% confidence)
sort_combo.py:15: unreachable code after 'return' (100% confidence)
sort_combo.py:18: unused variable 'llI' (60% confidence)
sort_combo.py:20: unused variable 'lII' (60% confidence)
sort_combo.py:22: unreachable code after 'return' (100% confidence)
sort_combo.py:24: unused function 'I' (60% confidence)
sort_combo.py:24: unused variable 'II' (100% confidence)
sort_combo.py:29: unused variable 'llII' (60% confidence)
sort_combo.py:30: unreachable code after 'return' (100% confidence)
sort_combo.py:32: unused function 'main' (60% confidence)
sort_combo.py:37: unused variable 'Ill' (60% confidence)
sort_combo.py:41: unused variable 'IIl' (60% confidence)
sort_combo.py:45: unused variable 'llII' (60% confidence)
sort_combo.py:50: unused variable 'III' (60% confidence)
sort_combo.py:55: unused variable 'lllI' (60% confidence)
sort_combo.py:59: unused variable 'lIll' (60% confidence)
```

Listing 24: Vulture output for sort.py with all transformations

Listing 24 shows the output from Vulture for sort.py when all the transformations are combined. It has three extra results compared to listing 22. These extra results indicate unused code after a return. The extra results are correct, but the other 10 results are still false positives. This means that this tool generates a mix of true positives and false positives when all transformations are combined. The changed names due to name obfuscation also make it harder to read which variables are shown in the output

As mentioned before, all of these results are based on metrics and data gathered from the output of the obfuscator. For example, Collberg's potency tries to predict how difficult it is for a human reader to understand a program after obfuscation by using complexity metrics. This can be a solid indicator of its potency, but there are no guarantees on how well this translates to actual human readers. Practical experiments involving human testers would be required to get more conclusive results for real world applications.

Some variability has been observed between the results for certain transformations or combinations applied to different scripts, especially for the execution times. There is also some variability present in the other metrics, but it is less noticeable than the changes in execution times. It can therefore be conclude that content of the input program will significantly influence the impact of these transformations on the tested metrics.

Some of the presented results, especially the ones surrounding software complexity metrics, are a good indicator for what the effect of a transformation will be. The impact on execution time for name obfuscation and literal obfuscation, even in combinations, is minimal and they can likely be used without much overhead. However, the control flow flattening and especially the function call obfuscation can have very mixed results in terms of execution times. Developers will therefore still have to test whether the loss in performance and the changes in the metrics are within their intended goals.

5.7 Conclusion

This chapter focused on the evaluation of obfuscation transformations, both from a theoretical and a practical point of view. The knowledge and results obtained in this chapter are used in this section to suggest some improvements to the implementation that was made for this thesis. It also provides some nuance and insights to two of the research questions posed at the start of the thesis. This will help to answer these questions in the conclusion of this thesis.

5.7.1 Implementation Improvements

Chapter 4 contains a discussion about potential improvements to the implemented obfuscator in section 4.4.4. That discussion mainly focuses on ways to make it more difficult to deobfuscate a program after obfuscation, as well as some general improvements from a developer's point of view. The suggested improvements in this section are based on the results that are discussed within this chapter, mainly focused on metric-based improvements. *Name Obfuscation* will not be discussed since it practically had no impact on any of the metrics during the experiments.

Literal encoding on its own does not have much impact on the execution time of a program, nor on any of the other metrics. This is expected for McCabe's complexity since the control flow graph of the program does not change, but Halstead's metrics are also only impacted when the program has boolean literals that get encoded. This transformation could be improved by making the expression that encodes to the literal more complex, for example by adding in some basic arithmetic. This would increase the complexity, which would also be reflected in Halstead's metrics. However, that improvement could cause some extra overhead making the program slower, so it should be made optional.

Function call obfuscation is the transformation that has the worst impact on the execution times of programs. It scales really poorly for some programs, so it needs to be modified to cause less overhead. The main reasons for all the overhead are that calls to `eval` are costly, and that this transformation applies to every function call. A simple but effective improvement would be to make the obfuscation more targeted: either the user should specify which calls should be hidden, or there should be some efficient algorithm to determine which calls can be obfuscated while maintaining certain performance goals.

Another approach could be to get rid of all the `eval`-statements by using different techniques to hide function calls. This could for example be done by splitting the function into different parts, having multiple copies of each function, or using a variable that points to the next function that needs to be called.[13] Parts of this could still rely on the dynamic nature of `eval`, such as assigning variables pointing to functions at runtime. It is even possible to define functions at runtime using `exec`, but do the actual function calls directly. This would reduce most of the overhead while still having a dynamic aspect in the process of calling functions.

Control flow flattening has a varying impact on the execution times of programs. It has a clear impact on all complexity metrics, that all increase in value when this transformation is applied. Just like for function call obfuscation, a simple improvement would be to not flatten every control flow graph in a program but instead be more selective about which functions are flattened. This could be done by requiring the user to choose which functions should be flattened, or by having an algorithm that determines which functions are best to flatten based on the goals of the developer. For example, this algorithm could suggest which functions should not be flattened based on the loss of performance.

A more minor improvement to control flow flattening would be to optimize the dispatching-structure. It is implemented with using a `match case`-structure, which goes through a list of cases to select the next basic block to execute. It is possible that using a different dispatch-method is more efficient, or that a better ordering of the encountered cases increases the performance. However, as long as some dispatch method is used to manually manage the control flow of the program there will be overhead compared to letting the interpreter do this for you.

[13] In C, this would be a function pointer. A similar construct can be used in Python, in which a variable points to a function, which can be called using that variable.

5.7.2 Research Answers

This chapter covered multiple elements related to the evaluation of code obfuscation, both in general and for Python specifically. The practical results and discussion of these results offer new insights into the feasibility of code obfuscation for Python. These insights can be used to answer the second and third question posed in the research questions (1.2):

1. Which modern obfuscation techniques exist that can be applied to Python while achieving their intended purpose?
2. Can typical metrics and evaluation of obfuscation techniques be applied to Python?
3. Do obfuscation techniques applicable to Python provide provable protection against attacks that intend to analyse or modify the source code?

The second question uses the term “typical metrics and evaluation” under the assumption that there is some sort of standard methodology to evaluate obfuscation techniques. This assumption is not entirely correct since there is no absolute standard, as is mentioned in section 5.1.1. However, there are some popular publications by [Collberg et al., 1997] and by [Anckaert et al., 2007] which have clearly defined approaches to evaluate obfuscation transformations.

These publications are discussed in section 5.2 and section 5.3, and are combined in section 5.4.3 in which a practical evaluation methodology for Python is covered. This practical methodology, which has also been tested in the practical experiments in this chapter, has proven that these metrics are indeed applicable to Python, which answers the second research question.

The third research question builds on the second question, and it’s mostly answered by the results discussed in section 5.5 and section 5.6. The metrics that are tracked do get impacted by the implemented transformations in a way that implies improved complexity, which is seen as provable protection against attacks that try to analyze or modify the code. The actual quality of this protection is however varied based on what technique or techniques are used.

Name obfuscation was already evaluated by [Collberg et al., 1997], who had concluded that it has a high potency and one-way resilience. It has been applied to Python in the created implementation, and has been shown to work for Python. *Literal Encoding* has some minor impact on the metrics, which means that the “provable” condition of the research question is satisfied on a binary scale. The impact on the metrics could be bigger, but it is also possible that this transformation is still potent in practice against a human attacker. A lack of impact on the metrics does not instantly disqualify a transformation from providing protection, so tests with human attackers could be conducted to explore this technique further.

Function call obfuscation, as discussed, has a “negative” impact on the metrics, because it made the complexity of the program less complex according to the metrics. It was shown with the output from Vulture that there is still a benefit to using this technique, but this benefit was not proven through the metrics that are used in research question two. Out of the four implemented techniques, this is the only one that does not provide provable protection when using the chosen metrics.

Lastly, *control flow flattening* has a noticeable impact on all complexity metrics, which clearly indicates that it provides provable protection of Python programs. This also matches with the research by [BinShamlan et al., 2019], that researched control flow obfuscation techniques (including flattening) on human readers and have found that it achieves its goal by making programs harder to analyze. Overall it can be concluded that the answer to both of the research questions is a “yes”, but there is a lot of nuance behind those answers as is explained in this section.

Chapter 6

Conclusion & Reflection

6.1 Findings & Future Work

There are three research questions that were posed in the introduction of this thesis:

1. Which modern obfuscation techniques exist that can be applied to Python while achieving their intended purpose?
2. Can typical metrics and evaluation of obfuscation techniques be applied to Python?
3. Do obfuscation techniques applicable to Python provide provable protection against attacks that intend to analyse or modify the source code?

The answer to the first research question is the list of techniques in section 4.2. This section contains a taxonomy that classifies an extensive list of obfuscation transformations, which includes multiple code examples of obfuscation techniques applied to Python to demonstrate their effect. There are also four techniques that are implemented in a proof-of-concepts obfuscator discussed in section 4.4.2 as a practical showcase.

This is not an exhaustive list, there are likely a lot more techniques that can be applied to Python, and a lot more new techniques could be created too. However, the provided list consists of applicable techniques that either occur frequently throughout various publications or techniques that are particularly relevant for Python. This on itself is a useful contribution, as such a list did not exist before in English literature.

Research question 2 can be answered with a "yes" because popular methodologies that researchers use to evaluate other programming languages (such as the methodologies discussed in sections 5.2 and 5.3) are based on metrics that can be applied to Python, which makes it possible to use a similar evaluation methodology for Python. Research question 3 can be answered with a "yes" because the implemented transformations have been tested with such an evaluation strategy, and some of the transformations had a quantifiable impact on the relevant metrics. Chapter 5 contains all the content related to answering these two questions, with section 5.7.2 discussing the answer to these two questions in more detail.

This thesis presented a proof-of-concept obfuscator, with which some practical experiments have been performed. Some of the results, as mentioned in section 5.5, indicate the need for more practical research to draw more conclusion about the impact of certain transformation as well as the impact of the original program. Various improvements have been suggested (in sections 4.4.4 and 5.7.1) that could improve this implementation or be taken into account when creating a new implementation. This implementation (or an improved implementation) should also be tested with human testers (as done by [BinShamlan et al., 2019]) to compare the quantifiable metrics with the impact on real programmers.

A different avenue that can be explored for future work is the extending of the existing research surrounding code obfuscation. For example, the combination of AI and obfuscation could be researched a lot more, since there are interesting results on innovative studies. This includes the research by [Lacomis et al., 2019] on reconstructing identifiers and the research by [Canavese et al., 2017] that uses AI to predict the potency of obfuscation transformations. Some experimentation with ChatGPT and the proof-of-concept implementation (in section 4.5) also show that large language models can possibly play an important role for code obfuscation in the future.

6.2 Personal Reflection

The process of doing research, creating and testing an implementation, and writing a thesis was a new kind of challenge on a scale that I have not experienced before. I think that the overall end result mostly matches up with my initial expectations for this process. The two main results, the implementation and this text, are an accurate reflection of the work I have done throughout the last year.

The general process of creating this thesis brought some serious struggles with it along the way. Accurately estimating the time it takes to research and write text is something that I keep having issues with. I understand that it will always take a lot more time than expected, especially when writing in English (as a non-native speaker), but this process has proven to me that accurate estimations are a skill that I need to work on a lot more.

I got lot of good advice from my promotor during the overall process and lot of specific feedback from my mentor for the writing specifically. I have taken in and applied this feedback as much as possible. Some suggestions or ideas had to be scrapped or changed due to time constraints, and there is some feedback that I did not manage to properly process. However, I did always try to incorporate all feedback and make sure that I kept past feedback in mind.

The actual content of my thesis contains a lot of things that would get different priorities or approaches if I had to do this all over. For example, opaque predicates were briefly mentioned in chapter 4, but in reality a whole chapter could be dedicated to this technique, which warrants more than a brief mention and a single example. I also had some wrong assumptions about various things, but these were either corrected during the process or explicitly mentioned in the thesis text.

There are of course also a lot of general things that could be improved: more testing, a more complete implementation, incorporating opaque predicates more, testing with human testers, and more. These things are stated when applicable, and some improvements are suggested both in the general text as well as in this conclusion.

Lastly, I think that I can safely state that code obfuscation for Python (or interpreted languages in general) is an interesting research field with a lot more potential. However, it is likely better for there to be more general research into obfuscation with more human testing as well as more research to combine AI and obfuscation. This thesis has proven that obfuscating Python fundamentally is similar to obfuscating other programming languages and that most of the differences can be overcome. More research into the other languages and techniques can therefore also contribute to obfuscating Python.

Bibliography

[Ahad et al., 2023] Ahad, A., Jung, C., Askar, A., Kim, D., Kim, T., and Kwon, Y. (2023). Pyfet: Forensically equivalent transformation for python binary decompilation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3296–3313.

[Aho et al., 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Pearson Education, Boston, MA, 2nd edition.

[Anckaert et al., 2007] Anckaert, B., Madou, M., De Sutter, B., De Bus, B., De Bosschere, K., and Preneel, B. (2007). Program obfuscation: A quantitative approach. In *Proceedings of the 2007 ACM Workshop on Quality of Protection*, QoP '07, page 15–20, New York, NY, USA. Association for Computing Machinery.

[Aycock, 2003] Aycock, J. (2003). A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113.

[Barak et al., 2012] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2012). On the (im)possibility of obfuscating programs. *J. ACM*, 59(2).

[Basile et al., 2023] Basile, C., De Sutter, B., Canavese, D., Regano, L., and Coppens, B. (2023). Design, implementation, and automation of a risk management approach for man-at-the-end software protection. *Computers Security*, 132:103321.

[Behnel et al., 2011] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39.

[Bhansali et al., 2022] Bhansali, S., Aris, A., Acar, A., Oz, H., and Uluagac, A. S. (2022). A first look at code obfuscation for webassembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '22, page 140–145, New York, NY, USA. Association for Computing Machinery.

[BinShamlan et al., 2019] BinShamlan, M. H., Bamatraf, M. A., and Zain, A. A. (2019). The impact of control flow obfuscation technique on software protection against human attacks. In *2019 First International Conference of Intelligent Computing and Engineering (ICOICE)*, pages 1–5.

[Canavese et al., 2017] Canavese, D., Regano, L., Basile, C., and Viticchié, A. (2017). Estimating software obfuscation potency with artificial neural networks. In *Security and Trust Management: 13th International Workshop, STM 2017, Oslo, Norway, September 14–15, 2017, Proceedings 13*, pages 193–202. Springer.

[Ceccato et al., 2015] Ceccato, M., Capiluppi, A., Falcarin, P., and Boldyreff, C. (2015). A large study on the effect of code obfuscation on the quality of java code. *Empirical Software Engineering*, 20(6):1486–1524.

[Chidamber and Kemerer, 1994] Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Chikofsky and Cross, 1990] Chikofsky, E. and Cross, J. (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17.

[Collberg and Thomborson, 2002] Collberg, C. and Thomborson, C. (2002). Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746.

[Collberg et al., 1997] Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial*.

[Everson et al., 2022] Everson, D., Cheng, L., and Zhang, Z. (2022). Log4shell: Redefining the web attack surface. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2022*.

[Halstead, 1977] Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

[Harrison and Magel, 1981] Harrison, W. A. and Magel, K. I. (1981). A complexity measure based on nesting level. *ACM Sigplan Notices*, 16(3):63–74.

[Henry and Kafura, 1981] Henry, S. and Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518.

[Herrera, 2020] Herrera, A. (2020). Optimizing away javascript obfuscation. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 215–220.

[Huang et al., 2023] Huang, T., Huang, J., Pang, Y., and Yan, H. (2023). Smart contract watermarking based on code obfuscation. *Information Sciences*, 628:439–448.

[Juneau et al., 2010] Juneau, J., Baker, J., Wierzbicki, F., Muoz, L. S., Ng, V., Ng, A., and Baker, D. L. (2010). *The definitive guide to Jython: Python for the Java platform*. Apress.

[Kang et al., 2021] Kang, S., Lee, S., Kim, Y., Mok, S.-K., and Cho, E.-S. (2021). Obfus: An obfuscation tool for software copyright and vulnerability protection. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, CODASPY '21, page 309–311, New York, NY, USA. Association for Computing Machinery.

[Kulkarni, 2011] Kulkarni, P. A. (2011). Jit compilation policy for modern machines. *SIGPLAN Not.*, 46(10):773–788.

[Lacomis et al., 2019] Lacomis, J., Yin, P., Schwartz, E., Allamanis, M., Le Goues, C., Neubig, G., and Vasilescu, B. (2019). Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639.

[László and Kiss, 2009] László, T. and Kiss, Á. (2009). Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86.

[Liang et al., 2017] Liang, Z., Li, W., Guo, J., Qi, D., and Zeng, J. (2017). A parameterized flattening control flow based obfuscation algorithm with opaque predicate for reduplicate obfuscation. In *2017 International Conference on Progress in Informatics and Computing (PIC)*, pages 372–378.

[Liu and Wang, 2020] Liu, Z. and Wang, S. (2020). How far we have come: Testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT*

International Symposium on Software Testing and Analysis, ISSTA 2020, page 475–487, New York, NY, USA. Association for Computing Machinery.

[McCabe, 1976] McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

[Munson and Kohshgoftaar, 1993] Munson, J. C. and Kohshgoftaar, T. M. (1993). Measurement of data structure complexity. *Journal of Systems and Software*, 20(3):217–225. Oregon Metric Workshop on Software Metrics, 1992.

[Myles and Collberg, 2006] Myles, G. and Collberg, C. (2006). Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171.

[Newman et al., 2022] Newman, Z., Meyers, J. S., and Torres-Arias, S. (2022). Sigstore: Software signing for everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2353–2367, New York, NY, USA. Association for Computing Machinery.

[Oviedo, 1980] Oviedo, E. I. (1980). Control flow, data flow, and program complexity. In *Proceedings of IEEE COMPSAC*, pages 146–152.

[Schnappinger and Streit, 2021] Schnappinger, M. and Streit, J. (2021). Efficient platform migration of a mainframe legacy system using custom transpilation. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 545–554.

[Schrittwieser and Katzenbeisser, 2011] Schrittwieser, S. and Katzenbeisser, S. (2011). Code obfuscation against static and dynamic reverse engineering. In Filler, T., Pevný, T., Craver, S., and Ker, A., editors, *Information Hiding*, pages 270–284, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Stärk et al., 2012] Stärk, R. F., Schmid, J., and Börger, E. (2012). *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media.

[Taibi et al., 2017] Taibi, D., Janes, A., and Lenarduzzi, V. (2017). How developers perceive smells in source code: A replicated study. *Information and Software Technology*, 92:223–235.

[Viticchié et al., 2016] Viticchié, A., Regano, L., Torchiano, M., Basile, C., Ceccato, M., Tonella, P., and Tiella, R. (2016). Assessment of source code obfuscation techniques. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 11–20.

[Wang et al., 2017] Wang, J., Axtell, R. L., and Loerch, A. (2017). Utilizing the positive impacts of software piracy in monopoly industries. In *Proceedings of the Agent-Directed Simulation Symposium*, ADS '17, San Diego, CA, USA. Society for Computer Simulation International.

Appendix A

Dutch Summary

Code obfuscatie is een techniek die gebruikt wordt om het moeilijker te maken om programma's te analyzeren en aan te passen. Het neemt een programma en past het aan op een manier waarbij het programma op dezelfde manier blijft werken, maar het reverse engineeren veel moeilijker wordt. Er is veel onderzoek naar obfuscatie van C(++) en Java, deze talen hebben ook meer praktische hulpmiddelen voor obfuscatie. Python heeft dit allemaal niet. In deze thesis wordt onderzocht of obfuscatie voor Python een mogelijkheid is. Hierbij wordt er ook gekeken naar methodes om technieken voor Python evalueren. Een eigen implementatie van een obfuscator met vier technieken is geschreven als proof-of-concept. Deze werd gebruikt om een evaluatiemethode te testen. De resultaten van de experimenten tonen aan dat obfuscatie op Python kan worden toegepast, maar dat er een uiteenlopend effect kan zijn op de prestaties van het programma.

A.1 Introductie

Hoofdstuk 1 introduceert het onderwerp, stelt de onderzoeksvragen en geeft een overzicht van de tekst.

De volgende 3 onderzoeksvragen worden onderzocht in deze tekst:

1. Welke moderne obfuscatie technieken bestaat er die op Python kunnen worden toegepast zodat ze hun doel bereiken?
2. Is het mogelijk om typische evaluatie methodes voor obfuscatie technieken toe te passen op Python?
3. Zorgen obfuscatie technieken voor Python aantoonbare beveiliging tegen aanvallen die de code willen analyzeren of aanpassen?

A.2 Software bescherming

Hoofdstuk 2 introduceert 3 technieken die gebruikt worden om software te beschermen [Collberg and Thomborson, 2002]:

- Watermarking: Het toevoegen van watermerken waarmee de ontwikkelaar van de software kan aantonen dat deze de software heeft gemaakt. Het kan ook gebruikt worden om te "fingerprinten", dit voegt unieke informatie toe aan elke kopie van een stuk software om te achterhalen wie deze kopie oorspronkelijk heeft gekocht.
- Tamper-proofing: Dit zorgt ervoor dat software niet aangepast kan worden, omdat bij een aanpassing de software niet meer (correct) gaat uitvoeren).

- Obfuscation: Dit is het aanpassen van software om hem onduidelijker te maken terwijl de functionaliteit bewaard wordt.

Het concept *reverse engineering* wordt ook uitgelegd en verkend: dit is het analyzeren van een system om te achterhalen uit welke componenten een systeem bestaat, en deze informatie gebruiken om een abstractie representatie van het systeem te maken (op een hoger niveau).

Reverse engineeren wordt gebruikt om software te analyzeren en heeft twee grote varianten: *statische* analyze en *dynamische* analyze. Bij statische analyze wordt de software geanalyzeerd zonder dat hij wordt uitgevoerd, bijvoorbeeld door alle bestanden te analyzeren. Bij dynamische analyze wordt de software uitgevoerd en wordt het gedrag van de software geanalyzeerd, bijvoorbeeld door te kijken welke paketten deze over het internet verstuurt.

A.3 Bytecode en decompilatie

Hoofdstuk 3 geeft een kleine inleiding over Python bytecode. Bytecode is een tussenvorm van programma code, die typisch tussen gewone broncode en machine code ligt. In het geval van Python is dit een binair formaat dat opgeslagen wordt in `.pyc`-bestanden die kunnen uitgevoerd worden in de plaats van het originele bronbestand.

Deze bytecode is niet ontworpen om terug omgezet te worden naar broncode, maar er bestaand *decompilers*, programma's die dit proces toch (deels) kunnen omkeren om de originele pythoncode na te bouwen. Deze programma's hebben wel enkele beperking en problemen, bijvoorbeeld met recentere versie van Python (zoals versie 3.9). [Ahad et al., 2023] heeft hiervoor een oplossing ontworpen die het decompileren van Python bytecode verbeterd.

Het proces van Python *freezing* wordt ook uitgelegd. Hierbij wordt een bestand met python broncode verwerkt naar een uitvoerbaar bestand, terwijl Python normaal gezien met een interpreter wordt uitgevoerd. Freezing omzeild dit door de interpeter en alle nodige bestanden samen te verpakken in een uitvoerbaar bestand. Hierdoor is het mogelijk voor eindgebruikers om een Python programma uit te voeren zonder dat ze zich zorgen moeten maken over de juiste Python versie of bibliotheken.

A.4 Broncode Obfuscatie

Hoofdstuk 4 bespreekt alles wat te maken heeft met de theorie rond de obfuscatie van broncode. Enkele definities worden uitgelegd aan het begin van dit hoofdstuk, dit bevat onder meer de definities voor control flow, abstract syntax tree en runtime. De *control flow* van een programma omvat de volgorde waarop instructies worden uitgevoerd. Elementen zoals if-statement en lussen zijn typische structuren die de control flow beïnvloeden. De *abstract syntax tree* van een programma is een boomvoorstelling die alle elementen van het programma voorstelt als knopen in een boom-structuur. Deze structuur wordt opgebouwd en aangepast bij het toepassen van obfuscatietechnieken. *Runtime* is de tijdsperiode waarin een programma wordt uitgevoerd, in het geval van Python is dit wanneer de interpreter een stuk broncode aan het uitvoeren is.

Een taxonomie gebaseerd op het werk van [Anckaert et al., 2007] wordt gesuggereerd:

- Control Flow Ofuscatie: Technieken die de control flow moeilijker analyseerbaar maken.
- Data Obfuscatie: Technieken die de data die in het programma verwerkt wordt verbergen.
- Instructie Obfuscatie: Technieken die de instructies die worden uitgevoerd verbergen.
- Interpreter Technieken: Technieken die specifiek voor interpreters gemaakt zijn.

- Andere: Technieken met een bepaalde eigenschap die ervoor zorgt dat ze niet in een andere categorie passen.

Een hele lijst van technieken wordt besproken, inclusief voorbeelden van technieken wanneer die gemaakt kunnen worden. Listing 25 is een voorbeeld van de techniek "Naam Obfuscatie", hierbij worden alle comments, docstrings, en namen uit een programma verwijderd. Dit zorgt ervoor dat het programma dezelfde uitvoeren behoudt maar moeilijker leesbaar is voor een programmeur.

```
# BEFORE NAME OBFUSCATION
# class containing the name and age of a person
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # @return the info as a string
    def toString(self):
        out = self.name + " " + str(self.age)
        return out
# AFTER NAME OBFUCCATION
class l:
    def __init__(l, I, lI):
        l.ll = I
        l.lI = lI
    def l(l):
        I = l.ll + " " + str(l.lI)
        return I
```

Listing 25: Naam Obfuscatie (Voor en Na)

Naam Obfuscatie is ook een van de 4 technieken die in de eigen implementatie zitten verwerkt. De andere zijn het encoderen van literals, het verbergen van functie-aanroepen, en het afvlakken van de control flow graph. Deze implementatie wordt in detail besproken in sectie 4.4.

Op het einde van dit hoofdstuk is er ook een reeks experimenten met ChatGPT die besproken worden. ChatGPT is een groot taalmodel, een vorm van artificiële intelligentie. De experimenten proberen met behulp van ChatGPT de code die mijn implementatie genereert te analyzeren. Dit geeft wat interessante resultaten, zoals onder meer een redelijk accurate reconstructie van een getest programma. Dit geeft duidelijk aan dat er veel mogelijkheden zijn met dit soort interacties, en dat er meer experimentatie moet gebeuren.

A.5 Evaluatie van obfuscatie technieken

Hoofdstuk 5 onderzoekt de verschillende evaluatiemethoden die er bestaan om obfuscatie technieken te evaluaren. Dit gebeurd op basis van het werkd door [Collberg et al., 1997] en [Anckaert et al., 2007]. Er wordt gekeken naar verschillende metrieken die de complexiteit van software uitdrukken, het idee hierachter is dat deze metrieken kunnen aangeven dat een stuk software na obfuscatie complexer is geworden. Dit impliceert dat de obfuscatie het gewenste effect heeft gehad.

De volgende metrieken worden onderzocht:

- Halstead's berekende lengte: Deze metriek drukt op basis van de verschillende operatoren en operaties uit hoe lang een programma is.

- Halstead's moeite: Dit lijkt op de vorige metriek, maar deze is nu omgevormd om aan te geven hoe moeilijk het is om een stuk software te begrijpen.

- McCabe's complexiteit: Deze metriek drukt uit hoe complex de control flow graphs van het programma zijn.

- Uitvoeringstijd: Deze metriek meet hoe lang het duurt om een programma uit te voeren.

Deze metrieken worden gebruikt in een methode om Python obfuscatie technieken te testen. Hiervoor zijn twee typische Python tools gebruikt: Radon voor de complexiteitsmetrieken en timeit voor de uitvoeringstijd. Om de metrieken te gebruiken worden er telkens twee versies van een programma gebruikt: de originele versie en de versie met obfuscatie toegepast. De wijziging van de metrieken tussen deze versies geeft aan wat de impact van de obfuscatie is.

Filename	**Original**	**Name**	**Name (%)**	**Literal**	**Literal (%)**
fibonacci.py	1.169	1.181	101%	1.181	101%
prime.py	0.925	0.923	100%	0.944	102%
sort.py	0.66	0.649	98%	0.648	98%

Table A.1: Uitvoeringstijd (in seconden) en het relatieve verschil voor naam obfuscatie en het encoderen van literals

Transformation	**Length**	**Δ Length**	**Effort**	**Δ Effort**	**McCabe**	**Δ McCabe**
			fibonacci.py			
Original	40	X	73	X	4	X
Name	40	0%	73	0%	4	0%
Literal	40	0%	73	0%	4	0%
Call	10	-75%	16	-78%	4	0%
Flatten	72	80%	346	374%	14	250%

Table A.2: Complexiteitsmetrieken voor enkelvoudige transformaties

Tabel A.1 is een voorbeeldtabel van enkele resultaten rond de uitvoeringstijd die besproken worden. Hierin valt bijvoorbeeld te zien dat Naam Obfuscatie en Literals Encoderen praktisch geen impact heeft op de uitvoeringstijd. Tabel A.2 bevat een voorbeeld van resultaten rond de complexiteitsmetrieken. Ook hier hebben naam obfuscatie en literals encoderen geen impact op de metrieken. Functie-aanroep obfuscatie zorgt ervoor dat de complexiteitsmetrieken omlaag gaan, wat eigenlijk niet het gewesnte effect is, en control flow afvlakken zorgt er duidelijk voor dat alle metrieken omhoog gaan.

De volledige resultaten en discussie zijn te vinden in sectie 5.5 en sectie 5.6. Dit bevat ook een discussie over het gebruik van meerdere transformaties tegelijk. Afhankelijk van het invoerprogramma en de transformaties kan dit extreme resultaten geven, zo is het bijvoorbeeld voorgevallen dat een specifiek resultaat aangaf dat de obfuscatie ervoor zorgde dat het programma 160 keer langer nodig had om uit te voeren, een extreem resultaat.

De conclusie van dit hoofdstuk is dus dat voor het merendeel van de transformaties en combinaties er verschillende omstandigheden zijn die bijdragen aan de impact van obfuscatie. Drie van de vier technieken bieden aantoonbare veiligheid, en dat zijn ook de drie technieken waarbij het minste vertraging veroorzaakt werd.

A.6 Conclusie

De eerste onderzoeksvraag is beantwoord met de lijst van technieken in hoofdstuk 4, de tweede en derde onderzoeksvraag kunnen met een “ja” beantwoord worden. De tweede onderzoeksvraag is zowel in de theorie als in de praktijk beantwoord met een ja. De derde onderzoeksvraag is beantwoord met de resultaten van een praktisch experiment, waarbij drie van de vier obfuscatie technieken aantoonbare beveiliging kunnen toevoegen aan programma’s.

Appendix B

Snippets

This appendix contains a code fragment and some ChatGPT transcripts that were too long to include in the main text.

```
import random
ll = 300

def l(lI):
    Il = 0
    while Il != -1:
        match Il:
            case 0:
                if eval(''.join(map(chr, [108, 101, 110, 40, 108, 73, 41])))
                ↪  <= 1:
                    Il = 2
                else:
                    Il = 1
            case 2:
                return lI
                Il = 1
            case 1:
                lll = lI[eval(''.join(map(chr, [108, 101, 110, 40, 108, 73,
                ↪  41]))) // 2]
                llI = [llIl for llIl in lI if llIl < lll]
                lIl = [llIl for llIl in lI if llIl == lll]
                lII = [llIl for llIl in lI if llIl > lll]
                return eval(''.join(map(chr, [108, 40, 108, 108, 73, 41]))) +
                ↪  lIl + eval(''.join(map(chr, [108, 40, 108, 73, 73, 41])))
                Il = -1

def I(II):
    Il = 0
    while Il != -1:
        match Il:
            case 0:
                return [random.randint(1, 1000) for llII in
                ↪  eval(''.join(map(chr, [114, 97, 110, 103, 101, 40, 73,
                ↪  73, 41])))]
                Il = -1
```

```
def main():
    Il = 0
    while Il != -1:
        match Il:
            case 0:
                Ill = 1000
                IlI = []
                Il = 1
            case 1:
                IIl = eval(''.join(map(chr, [105, 116, 101, 114, 40, 114, 97,
                ↪ 110, 103, 101, 40, 108, 108, 41, 41])))
                Il = 3
            case 3:
                try:
                    lIII = eval(''.join(map(chr, [110, 101, 120, 116, 40, 73,
                    ↪ 73, 108, 41])))
                    Il = 4
                except StopIteration:
                    Il = 2
            case 4:
                III = eval(''.join(map(chr, [73, 40, 73, 108, 108, 41])))
                llll = eval(''.join(map(chr, [108, 40, 73, 73, 73, 41])))
                IlI.append(llll)
                Il = 3
            case 2:
                llII = eval(''.join(map(chr, [105, 116, 101, 114, 40, 114,
                ↪ 97, 110, 103, 101, 40, 109, 105, 110, 40, 50, 44, 32,
                ↪ 108, 108, 41, 41, 41])))
                Il = 6
            case 6:
                try:
                    lIll = eval(''.join(map(chr, [110, 101, 120, 116, 40,
                    ↪ 108, 108, 108, 73, 41])))
                    Il = 7
                except StopIteration:
                    Il = -1
            case 7:
                eval(''.join(map(chr, [112, 114, 105, 110, 116, 40, 39, 83,
                ↪ 111, 114, 116, 101, 100, 32, 108, 105, 115, 116, 32, 39,
                ↪ 32, 43, 32, 115, 116, 114, 40, 108, 73, 108, 108, 32, 43,
                ↪ 32, 49, 41, 32, 43, 32, 39, 58, 32, 39, 32, 43, 32, 115,
                ↪ 116, 114, 40, 73, 108, 73, 91, 108, 73, 108, 108, 93, 41,
                ↪ 41])))
                Il = 6
eval(''.join(map(chr, [109, 97, 105, 110, 40, 41])))
```

Listing 26: sort.py with all transformations applied

```
The provided Python code appears to be obfuscated and intentionally made
↪ difficult to understand. It uses the `match` statement and various
↪ operations to confuse the reader. While I can help you understand the
↪ code's structure and some of its functionality, keep in mind that this
↪ code might have been intentionally made complex to obfuscate its true
↪ purpose. Let's break down what's happening in the code:
```

```
1. The code starts with a series of function definitions: `l`, `I`, and
↪  `main`.

2. The function `l` takes a parameter `lI` and performs some operations on
↪  it. However, due to the obfuscation, the exact purpose of these
↪  operations is not immediately clear. It seems to involve manipulation of
↪  lists, conditionals, and evaluation of characters. The code returns a
↪  value based on these operations.

3. The function `I` takes a parameter `II` and returns a list of random
↪  integers between 1 and 1000.

4. The `main` function contains the main logic of the program. It initializes
↪  variables, enters a loop with a `match` statement, and performs different
↪  actions based on the value of the `Il` variable.

5. In the `main` function, there's a block of code that initializes `Ill` to
↪  1000 and creates an empty list `IlI`. Then, it enters a loop where it
↪  performs various operations based on the value of `Il`.

6. The code contains various `eval` statements that construct strings using
↪  ASCII values of characters and then evaluate those strings. This dynamic
↪  evaluation makes the code harder to read and understand.

7. The code prints a message indicating "Sorted list: " followed by some
↪  calculated value, but the exact calculation is hard to decipher due to
↪  the obfuscation.

Overall, the code seems to involve list manipulation, random number
↪  generation, and complex string manipulation. However, without a clear
↪  understanding of the purpose or context of the code, it's challenging to
↪  provide a definitive analysis of its functionality. If you have more
↪  information about the context or the purpose of this code, I might be
↪  able to provide further insights.
```

Listing 27: ChatGPT response for obfuscated sort.py

```
The provided code is a heavily obfuscated piece of Python code. It seems to
↪  be intentionally written in an obscure way to make it difficult to
↪  understand its purpose and functionality. I'll do my best to break down
↪  the code and explain its structure.

However, keep in mind that the obfuscation used here makes it challenging to
↪  provide a clear understanding of the exact purpose of the code. It's
↪  possible that the code contains malicious or unwanted behavior, so
↪  caution is advised when dealing with such code.

Here's a breakdown of the code:

The code starts with some import statements and a function call at the end
↪  (eval(''.join(map(chr, [109, 97, 105, 110, 40, 41])))). The characters
↪  [109, 97, 105, 110, 40, 41] correspond to the ASCII values of the
↪  characters "main()". This implies that the main() function will be
↪  executed at the end.
```

```
The sys.set_int_max_str_digits(100000) line attempts to set the maximum
↪  number of digits allowed when converting an integer to a string. However,
↪  this function does not exist in the Python sys module, so it will likely
↪  result in an error.

The code defines a function l(I) which takes an argument I. This function
↪  seems to be related to Fibonacci numbers based on the logic it contains.
↪  The function l(I) appears to generate Fibonacci numbers up to a certain
↪  index I.

The main() function seems to be responsible for printing out information
↪  about the Fibonacci sequence.

The code heavily uses the match statement, which is a feature introduced in
↪  Python 3.10 for pattern matching. It's used to determine the flow of the
↪  program based on the patterns defined in the case clauses.

The ASCII values of characters are used to create strings that are then
↪  evaluated to execute certain code snippets. This technique is used to
↪  obfuscate the actual logic of the program.

The try block with except StopIteration seems to be used to catch an
↪  exception, but its purpose is not clear due to the obfuscation.

Overall, the code appears to be a convoluted and intentionally obfuscated
↪  piece of Python code that deals with Fibonacci numbers and likely prints
↪  some information related to them. However, without further context or
↪  clearer code, it's difficult to provide a more detailed analysis of its
↪  behavior or purpose. If you have a clearer version of the code or more
↪  information about its intended functionality, I'd be happy to help
↪  further.
```

Listing 28: ChatGPT response for obfuscated fibonacci.py

```
The provided Python code appears to be obfuscated and convoluted, making it
↪  difficult to understand its purpose and functionality at a glance. The
↪  code consists of several nested loops, match-case constructs, and
↪  conditional statements, making it intentionally hard to decipher. Without
↪  knowing the context or specific intentions of the code, I can provide a
↪  rough analysis of its structure and behavior:

1. The code starts by defining a global variable `I` with the value `30000`.

2. The code defines a function `l(ll)` with a while loop. It uses the
↪  match-case construct to switch between different cases based on the value
↪  of the local variable `lI`. The function seems to return either an empty
↪  tuple `()` or an empty list `[]` in some cases.

3. Within the `l(ll)` function, there are multiple cases that evaluate and
↪  manipulate `lI` to control the flow of execution. There is also an `eval`
↪  function used to execute dynamically generated code based on ASCII values
↪  of characters. The purpose of this function is unclear due to
↪  obfuscation.
```

```
4. The `main()` function is defined next. Like the previous function, it uses
↪  a while loop and match-case construct. It seems to be involved in
↪  generating a list of numbers.

5. The `main()` function iterates over a sequence of cases, appending numbers
↪  to a list `II` and updating the `count` variable until `count` reaches
↪  the value of the global variable `I` (30000).

6. The code involves more obfuscated `eval` statements and manipulations of
↪  variables, making it challenging to determine the exact purpose of the
↪  code.

7. The last few lines of the code call the `main()` function and execute some
↪  more obfuscated code using `eval`.

In summary, the code appears to be intentionally obscured and obfuscated,
↪  making it difficult to determine its true purpose and functionality.
↪  Without more context or clear intent, it's hard to provide a precise
↪  analysis of what the code is meant to accomplish. If you have any
↪  specific questions or context about this code, I'd be happy to help
↪  further.
```

Listing 29: ChatGPT response for obfuscated prime.py

````
The given Python code defines a function `l(lI)` and performs some operations
↪  using it. Let's break down the code step by step:

1. **Function Definition:**
   ```python
   def l(lI):
       Il = 0
       while Il != -1:
           match Il:
               case 0:
                   II = [lI]
                   Il = 1
               case 1:
                   if lI != 1:
                       Il = 3
                   else:
                       Il = 2
               case 3:
                   if lI % 2 == 0:
                       Il = 5
                   else:
                       Il = 6
               case 5:
                   lI //= 2
                   Il = 4
               case 6:
                   lI = 3 * lI + 1
                   Il = 4
               case 4:
                   II.append(lI)
                   Il = 1
````
````

```
            case 2:
                return II
                Il = -1
   ```

   This function takes an integer `lI` as an argument and performs a sequence
   ↪ of operations on it based on a series of cases. It seems to be a
   ↪ custom implementation of the Collatz conjecture, where the sequence is
   ↪ generated based on the rules:
   - If `lI` is even, divide it by 2.
   - If `lI` is odd, multiply it by 3 and add 1.
   The function continues this process until `lI` becomes 1.

2. **Variable Definitions:**
   ```python
   I = eval(''.join(map(chr, [105, 110, 116, 40, 105, 110, 112, 117, 116, 40,
   ↪  39, 73, 110, 58, 39, 41, 41])))
   ll = eval(''.join(map(chr, [108, 40, 73, 41])))
   ```

   These lines create two variables:
   - `I`: This variable appears to be storing a string containing the
   ↪ characters "int(int('In:',))". It's using the `eval` function to
   ↪ convert a series of ASCII values to characters and then interpreting
   ↪ the resulting string. However, the purpose of this string isn't clear
   ↪ without more context.
   - `ll`: This variable seems to store the result of calling the function
   ↪ `int('I')`. This will try to convert the string "I" to an integer, but
   ↪ since "I" is not a valid representation of an integer, this will raise
   ↪ a `ValueError` when evaluated.

3. **Code Execution:**
   ```python
   eval(''.join(map(chr, [112, 114, 105, 110, 116, 40, 39, 79, 117, 116, 58,
   ↪  39, 44, 32, 108, 108, 41])))
   ```

   This line uses `eval` to execute code generated from a sequence of ASCII
   ↪ values. The code appears to be: `print('Out: ', ll)`. Since `ll` would
   ↪ raise an error when trying to convert the string "I" to an integer,
   ↪ this line would result in an error when executed.

In summary, the code defines a function that implements the Collatz
↪ conjecture, but the purpose and usage of the other variables (`I` and
↪ `ll`) are unclear without more context. Additionally, the code will
↪ likely encounter errors when executed due to the incorrect conversion of
↪ characters to integers and the attempt to convert the string "I" to an
↪ integer.
```

**Listing 30:** ChatGPT response for obfuscated collatz.py
```

```
The provided Python code is quite obfuscated, making it challenging to
↪  understand its purpose at first glance. It seems to define a few
↪  functions (`l` and `main`) and involves some complex control flow using
↪  the `match` statement (which isn't a standard Python construct). The code
↪  also uses a lot of character mappings to build strings and perform
↪  operations.

Let's break down the code step by step:

1. `sys.set_int_max_str_digits(100000)`:
   This line seems to be setting a maximum number of digits for
    ↪  integer-to-string conversions. However, `set_int_max_str_digits` is
    ↪  not a standard function in Python's `sys` module, so it's likely that
    ↪  this code might not run in a regular Python environment.

2. `I = 300000`:
   This sets the variable `I` to 300000.

3. `def l(I): ...`:
   The function `l` is defined here. It seems to be a recursive function with
    ↪  a complex control flow using the `match` statement. However, the usage
    ↪  of `match` with cases like `case 0` and `case 2` is unconventional and
    ↪  not part of standard Python syntax.

4. `def main(): ...`:
   The `main` function is defined here. It also uses the non-standard `match`
    ↪  statement and seems to involve some string manipulations, but the
    ↪  exact purpose is unclear due to the obfuscation.

5. `eval(''.join(map(chr, [...])))`:
   Throughout the code, there are multiple instances of `eval` being used to
    ↪  execute dynamically generated strings. The strings are constructed by
    ↪  mapping integers to characters using the `map(chr, ...)` function and
    ↪  then joining them with `join`.

6. `eval(''.join(map(chr, [109, 97, 105, 110, 40, 41])))`:
   This line appears to execute the `main()` function by creating a string
    ↪  containing the characters for "main()" and evaluating it using `eval`.

In summary, this code is heavily obfuscated and uses unconventional
↪  constructs like the `match` statement and non-standard functions like
↪  `sys.set_int_max_str_digits`. It's difficult to ascertain the exact
↪  purpose of the code without a clearer understanding of the intentions
↪  behind the obfuscation. It's also worth noting that the code's reliance
↪  on non-standard constructs and dynamically generated strings makes it
↪  potentially unreliable and non-portable across different Python
↪  environments.
-------
What algorithm could this be? (user prompt)
-------
```

```
The provided code is heavily obfuscated, making it extremely difficult to
↪ determine the underlying algorithm with certainty. However, given the
↪ available context, it's still challenging to identify a specific
↪ algorithm due to the unconventional usage of constructs like the `match`
↪ statement, dynamically generated strings, and non-standard function
↪ calls.

Based on the limited visibility into the code's functionality, it's not
↪ possible to definitively identify the algorithm being implemented. If you
↪ could provide more context about the intended purpose of this code or any
↪ additional information, I might be able to offer more insights.
↪ Additionally, if you could clarify whether this code is part of a puzzle,
↪ challenge, or specific project, it might help in understanding its
↪ purpose.
-------
Perhaps it calculates a sequence? (user prompt)
-------
It's possible that the code is attempting to calculate a sequence, given the
↪ presence of a function `l(I)` and some recursion in the code. However,
↪ due to the heavy obfuscation and unconventional constructs used, it's
↪ still challenging to determine the exact nature of the sequence or the
↪ algorithm being implemented.

The code seems to involve iterative control flow using the `match` statement
↪ and contains recursive calls within the `l` function. There's also a
↪ mention of Fibonacci-like operations where values are added or returned
↪ based on previous values.

If you're looking for a sequence-generating algorithm, this code could
↪ potentially be an attempt to implement some variation of a sequence,
↪ perhaps related to Fibonacci or a similar pattern. However, without
↪ clearer understanding or more context, it's difficult to provide a
↪ precise analysis of the sequence or the algorithm being used.
```

Listing 31: ChatGPT response for fibonacci.py without the string Fibonacci present

```
import sys
sys.set_int_max_str_digits(100000)
I = 300000

def l(I):
    ll = 0
    while ll != -1:
        match ll:
            case 0:
                if I <= 2:
                    ll = 2
                else:
                    ll = 1
            case 2:
                return 1
                ll = 1
            case 1:
                lll, Il = (1, 1)
                ll = 3
```

```
            case 3:
                lI = eval(''.join(map(chr, [105, 116, 101, 114, 40, 114, 97,
                ↪  110, 103, 101, 40, 51, 44, 32, 73, 32, 43, 32, 49, 41,
                ↪  41])))
                ll = 5
            case 5:
                try:
                    llI = eval(''.join(map(chr, [110, 101, 120, 116, 40, 108,
                    ↪  73, 41])))
                    ll = 6
                except StopIteration:
                    ll = 4
            case 6:
                lll, Il = (Il, lll + Il)
                ll = 5
            case 4:
                return Il
                ll = -1

def main():
    ll = 0
    while ll != -1:
        match ll:
            case 0:
                II = eval(''.join(map(chr, [108, 40, 73, 41])))
                eval(''.join(map(chr, [112, 114, 105, 110, 116, 40, 39, 84,
                ↪  104, 101, 32, 39, 32, 43, 32, 115, 116, 114, 40, 73, 41,
                ↪  32, 43, 32, 39, 116, 104, 32, 110, 117, 109, 98, 101,
                ↪  114, 32, 105, 115, 58, 32, 39, 32, 43, 32, 115, 116, 114,
                ↪  40, 73, 73, 41, 41])))
                ll = -1
eval(''.join(map(chr, [109, 97, 105, 110, 40, 41])))

Make a deobfuscated version of this Python program. (user prompt)
-------
Certainly! It seems like the given code is intentionally obfuscated, making
↪  it hard to read and understand. Here's a deobfuscated version of the code
↪  with comments explaining each section:

```python
# Set the maximum number of digits when converting integers to strings
import sys
sys.set_int_max_str_digits(100000)

# Initial value
I = 300000

# Function l(I)
def l(I):
    ll = 0
    while ll != -1:
        if ll == 0:
            if I <= 2:
                ll = 2
            else:
```
```

````
                ll = 1
        elif ll == 2:
            return 1
            ll = 1  # This line is unreachable
        elif ll == 1:
            lll, Il = (1, 1)
            ll = 3
        elif ll == 3:
            lI = iter(range(3, I + 1))
            ll = 5
        elif ll == 5:
            try:
                llI = next(lI)
                ll = 6
            except StopIteration:
                ll = 4
        elif ll == 6:
            lll, Il = (Il, lll + Il)
            ll = 5
        elif ll == 4:
            return Il
            ll = -1

# Main function
def main():
    ll = 0
    while ll != -1:
        if ll == 0:
            II = int(input())  # Convert user input to an integer
            print("The " + str(II) + "th number is: " + str(l(II)))
            ll = -1

main()  # Call the main function
```

Please note that the original code contained various obfuscation techniques
↪  such as using ASCII values to represent characters and using `eval` to
↪  execute code from strings. The deobfuscated version provided here
↪  replaces these obfuscated sections with their actual meanings, making the
↪  code more readable and understandable.
-------
Analyze this deobfuscated code. (user prompt)
-------
Certainly, let's analyze the deobfuscated code step by step:

```python
import sys
sys.set_int_max_str_digits(100000)

I = 300000

def l(I):
    ll = 0
    while ll != -1:
        if ll == 0:
````

```
            if I <= 2:
                ll = 2
            else:
                ll = 1
        elif ll == 2:
            return 1
            ll = 1  # This line is unreachable
        elif ll == 1:
            lll, Il = (1, 1)
            ll = 3
        elif ll == 3:
            lI = iter(range(3, I + 1))
            ll = 5
        elif ll == 5:
            try:
                llI = next(lI)
                ll = 6
            except StopIteration:
                ll = 4
        elif ll == 6:
            lll, Il = (Il, lll + Il)
            ll = 5
        elif ll == 4:
            return Il
            ll = -1

def main():
    ll = 0
    while ll != -1:
        if ll == 0:
            II = int(input())
            print("The " + str(II) + "th number is: " + str(l(II)))
            ll = -1

main()
```

```
1. The code imports the `sys` module and sets the maximum number of digits
↪  when converting integers to strings. This might be useful when dealing
↪  with large numbers.

2. The variable `I` is assigned the value `300000`.

3. The function `l(I)` is defined. This function seems to be calculating the
↪  nth number in a sequence and returns it. It uses a series of cases with
↪  an integer `ll` as a control variable for the different steps of
↪  calculation. This function employs a Fibonacci-like sequence to calculate
↪  the nth number.

4. The `main()` function is defined. It uses a loop controlled by `ll` to
↪  repeatedly take user input, call the `l()` function to calculate the
↪  corresponding number in the sequence, and print the result.

5. The `main()` function is executed at the end of the code, initiating the
↪  user input and output process.
```

````
In summary, the code seems to be calculating the nth number in a sequence
↪  using the `l()` function. The sequence calculation follows a pattern
↪  similar to the Fibonacci sequence. The `main()` function handles user
↪  input and output for displaying the nth number in the sequence. The code
↪  is designed in a convoluted way, likely as an exercise in obfuscation
↪  rather than practical programming.
-------
Fix this code (user prompt)
-------
Certainly, I can help you fix the code. However, since the original code is
↪  quite convoluted, I'll provide a simplified version of the code that
↪  achieves the same functionality without unnecessary complexity and
↪  obfuscation:

```python
def calculate_nth_number(n):
    if n <= 2:
        return 1

    prev_1 = 1
    prev_2 = 1

    for i in range(3, n + 1):
        current = prev_1 + prev_2
        prev_1, prev_2 = prev_2, current

    return prev_2

def main():
    try:
        n = int(input("Enter a positive integer n: "))
        if n <= 0:
            print("Please enter a positive integer.")
        else:
            nth_number = calculate_nth_number(n)
            print(f"The {n}th number is: {nth_number}")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")

if __name__ == "__main__":
    main()
```

In this corrected version, I've simplified the logic and removed unnecessary
↪  obfuscation techniques. The code defines a function
↪  `calculate_nth_number(n)` that calculates the nth number in a sequence.
↪  The `main()` function handles user input, error handling, and output. The
↪  `if __name__ == "__main__":` block ensures that the `main()` function is
↪  executed when the script is run directly and not when it's imported as a
↪  module.
````

Listing 32: ChatGPT deobfuscated fibonacci.py