Multithreading for Obfuscating Code Against Dynamic Analysis and Reverse Engineering

by

Tanveer Shahriar Arnob
22101157

Jafor Mohammad
23341109

Md. Shariar Islam Shuvo
21201105

Mohammad Mehrab Islam Fahim
21101190

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
January 2025

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing the degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Student's Full Name & Signature:

Tanveer Shahriar Arnob
22101157

Jafor Mohammad
23341109

Md. Shariar Islam Shuvo
21201105

Mohammad Mehrab Islam Fahim
21101190

Approval

The thesis/project titled "Multithreading for Obfuscating Code Against Dynamic Analysis and Reverse Engineering" submitted by

1. Tanveer Shahriar Arnob (22101157)
2. Jafor Mohammad (23341109)
3. Md. Shariar Islam Shuvo (21201105)
4. Mohammad Mehrab Islam Fahim (21101190)

Of Spring, 2025 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on June 20, 2025

Examining Committee:

Supervisor:
(Member)

Dr. Muhammad Nur Yanhaona
Associate Professor
Department of Computer Science and Engineering
Brac University

Thesis Coordinator
(Member)

Dr. Md. Golam Rabiul Alam
Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi, PhD
Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

Abstract

Code obfuscation is applied to change the structure of software code that makes it hard to understand; in fact, it reduces the readability of code. Obfuscation techniques give protection to proprietary code from reverse engineering, tampering, and unauthorised usage. General code obfuscation techniques make code analyses difficult. Meanwhile, multithreading allows multiple threads to run concurrently within a single process, where each thread helds a separate execution flow, allowing various parts of code to execute simultaneously and independently. Code obfuscation techniques have been used in industrial applications for years, but the scope of multithreading in this area has still not been properly explored. The general code obfuscation techniques change the code structures as well as change the control flow format in a single thread. Multithreading obfuscation has the potential to utilise the complexities of multithreaded execution by adding layers of obscurity to the content and runtime behaviour of code. By applying M threads in a program code with N functions, it may provide N^M combinations, making the control flow harder to analyse. Applying this approach enables the randomness of code execution, and it can give protection against dynamic analysis. Finally, a synchronisation mechanism can increase enough complexity to hinder the reverse engineering. These approaches should make the reconstruction of the original sequential logic much harder. In this research, we aim to design an obfuscation tool for C++ programs that applies multithreading obfuscation techniques to large-scale source code and this would significantly enhance the software security by increasing complexity and unpredictability. And the next step of our research is to focus on the improvement of our obfuscator.

Keywords: Code Obfuscation; Multithreading; Software Security; POSIX Threads; Control Flow Obfuscation; Dynamic Analysis; Synchronization Mechanisms; LLVM; C Programs; Thread Scheduling

Contents

Acknowledgement

Firstly, all praise to the Great Allah for whom our thesis has been completed without any major interruption.
Secondly, to our supervisor, Dr. Muhammad Nur Yanhaona, sir, for his kind support and advice in our work. He helped us whenever we needed help.
Thirdly, Dr. Muhammad Nur Yanhaona, Dr. MD Sadek Ferdous Sir, and the whole judging panel of Information Security and Distributed Systems of Spring 2025 for accepting our paper and giving all the reviews that helped us a lot in our later works.
And finally, to our parents, without their throughout support, it may not be possible. With their kind support and prayers, we are now on the verge of our graduation.

Chapter 1

Introduction

In today's digital environment, the security of code is a critical concern, with various attacks targetting vulnerabilities in software. Static analysis, reverse engineering, and side-channel attacks are techniques that adversaries use to exploit the weaknesses of the industrial code. Static analysis identifies the critical routines related to cryptography and cryptographic keys, whereas reverse engineering allows attackers to see the encryption algorithm used in the source code. Meanwhile, side-channel attacks extract sensitive information, such as encryption keys, by monitoring the behavior of a system. While these attack combinations are serious threats to software which has not been secured, they also bypass protection mechanisms. At the time, in 2010, when hacker George 'Geohot' Hotz reverse engineered the PlayStation 3 firmware [23], enabling the use of illegal software on the console and allowing piracy of games The types of incidents that underscore the need for mitigations such as code obfuscation against reverse engineering (and other attacking actions) are the Stuxnet worm [11] which compromised Siemens' SCADA systems in Iran, and Jon Lech Johansen's reverse engineering of the Content Scramble System (CSS) for DVDs in 1999 [17].

Making software code hard to read through obfuscation is an old and common method to stop others from misusing and changing it. The use of ProGuard [31], ConfuserEx [41], Babel Minify [38] and JScrambler [13] for JavaScript serves to alter the code structure and makes analysis hard. At present, the modern threats we face cannot be dealt with using these methods. Ghidra [40], CFR [16], Radare2 [26] and ILSpy [35] allow users to reverse many changes created by basic obfuscators. Consequently, it is not enough to rely on the same old techniques to keep our code safe, so we need to look for new and adaptive methods.

Despite obfuscation advancements, the use of multithreading for enhancing software security is underexplored. Existing obfuscation methods are mostly limited to single threaded applications and can be easily defeated by an increasing number of reverse engineering tools. The study of how multithreading's natural complexity and concurrent execution paths can be applied to generate more robust obfuscation strategies is necessary.

1.1 Research Statement

The focus of this research is to explore the limitations of the existing obfuscation tools as well as the general code obfuscation approaches and to explore the area of multithreading, which we aim to integrate with the existing obfuscation techniques. This research aims to enhance both academic and industrial practices for protecting proprietary source code by making the call graph more complex and unpredictable. The findings could help against the risk of reverse engineering and protect critical software systems across various industries. This approach can offer distinct advantages compared to other protection methods emerging in the market.

1.2 Research Objectives

The primary goal of this research is to incorporate a multi-threading based approach to the obfuscation of existing code obfuscation tools for C programs and to improve the software security. The techniques developed will randomize thread assignment and synchronization, complicating the control flow and obstructing dynamic analysis and reverse engineering. In particular, the study (1) designs a multithreaded obfuscation approach that makes control flow and execution path more complicated, (2) evaluates the obfuscation approach against dynamic analysis and reverse engineering tools, and (3) optimizes the approach to minimize performance overhead with maximum security resilience. Considering that most modern multithreaded applications have largely refuted traditional single threaded obfuscation approaches, this work aims to bridge this gap focusing on how to apply such methods to the multithreaded applications of the modern platforms, thereby contributing to the evolution of software protection for cybersecurity.

Depending on the specifics of the study, the study will assume higher priority towards better security, but it may come at a performance price, which must be kept low for practical usability. Furthermore, the research may not cover all the ways in which the attack could be performed, for example, a hardware-based side-channel attack, but focuses on making the code resilient against reverse engineering and dynamic analysis. Applying code obfuscation with the idea of multithreading can increase the complexity of software many times over, meaning that the software is more resistant against reverse engineering and dynamic analysis.

In the beginning of this thesis, there is a back-ground study, a thorough literature review, then the design and implementation of our multithreading based obfuscation tool. It further provides a thorough evaluation of the tool's effectiveness and performance impact, and discusses results and future study directions.

Chapter 2

Background

Protecting software from attackers who would exploit vulnerabilities largely depends on obfuscation. These intruders generally use unusual methods, such as side-channel attacks and analyzing software without changing it, to acquire information, remove secrets, and sometimes compromise software's entire structure. Usually, their process is to spot regions of a program that can be attacked [28]. By throwing in extra complexities, obscuring techniques ensure that hackers will find it difficult to determine the code's logic, even though the software can still work. It greatly reduces the ability to carry out static analysis or reverse engineering [28]. Also, complicated methods such as obfuscation using multithreading make it harder for anyone trying to monitor and analyze the software's behavior by making it more randomized and less predictable during execution.

2.1 Static Analysis

Static analysis is a method of examining software Static analysis is a technique of assessment of the source code without actually running the program. It can be performed on source code as well as on executable files, which helps developers to find out kind of vulnerabilities, coding mistakes and compliance to coding standards. By examining the aspects of structure, basic control flow, and data flow of the code, Static Analysis tools can generate useful information on a number of problems that may reduce the level of security as well as affect the functionality of the system. There are various approaches to apply static analysis. Collbarge et al. describe several techniques, including control flow analysis, data flow analysis, data dependence analysis, alias analysis, slicing, and abstract interpretation, which encircle the methods used in static analysis [28].

2.1.1 Control Flow Analysis

The study of control flow analysis is the grouping of the order of statement, instruction or function call within a program. This technique helps and can identify unreachable code, infinite loops, or incorrect branching logic. They can lead to security vulnerabilities or logical errors. Mapping control flow is the ability to trace step by step execution flow. Thereby, developers can pinpoint areas, where maybe optimization or debugging is needed

to strengthen the application reliability. It is often represented with a Control Flow Graph (CFG) [28]:

- Basic blocks are sequences of instructions that run straight through without any branches, except at the end.
- Edges between these blocks indicate potential control transfers (like branches, loops, or function calls) that can occur at runtime.

The CFG provides a conservative model of all possible execution paths, serving as the foundation for further static analyses.

2.1.2 Data Flow Analysis

The data flow analysis studies how the data travels in a program along different code paths by tracking variables and their values. This analysis helps pinpoint the problem spot like uninitialized variables or data leaks, to ensure that there is secure handling and correct data usage throughout the application. That allows developers, in the process of dealing with how data flows, to introduce necessary safeguards and make their software more robust. In fact, Data flow analysis is to reveal how variables and data are propagated around the control flow of a program [28]. It answers key questions [28] like:

- "Where does the value of variable x come from?"
- "Will variable x be used after a particular point?"
- "Is variable x constant at a particular point?"

This technique analyzes the flow of values across the control flow graph and guarantees that when transformations (such as optimizations or obfuscations) are applied to program code, these transformations do not disturb the intended usage of the data.

2.1.3 Data Dependence Analysis

The goal of data dependence analysis is to check how instructions or statements in a program exchange data. It aims to spot which data elements each instruction deals with and what these transactions do to the manner and course of the program running. We try to find when one instruction relies on data created or altered by a different instruction. Data dependency exists when both instructions try to access the same memory spot and at least one of them modifies it. Dependencies like these play an important role in recognizing possible risks during simultaneous execution or changes.

This analysis is used by reordering transformations such as those in algorithms WMASB and OBFCF so that moving instructions within or between basic blocks does not violate dependencies or alter behavior of the entire program [28]. Data dependence analysis can be used to attack programs in order to reverse engineer, or even compromise programs, by understanding how data flows through program execution. They can see where sensitive data (e.g., encryption keys, credentials) is generated, used or stored by analyzing data dependencies. This enables them to target extraction or manipulation. Yet, attackers can understand how security checks rely on specific data. For example, they could break the dependency chain such that they can alter or bypass checks that prevent unauthorized access by fooling their system into giving the incorrect output for a certain input. Additionally, these attackers can gain understanding of the relationships between its variables and code segments, so they too can become aware of potentially exploitable vulnerabilities, like buffer overflows, uninitialized variables or overflows. In reality, you can discover hidden relationships between code and data, and then use data dependence analysis to extract sensitive information or break the system's logic.

2.1.4 Alias Analysis

The function to know is Alias Analysis, that will tell you if two or more variables, mainly pointers, are referring to the same memory location at a specific point in your program. In pointer heavy languages, it is important that incorrect transformations over aliases can break the memory accesses. Alias analysis is classified [28] into:

- **Flow-insensitive alias analysis:** Ignores control flow and provides a broad but less precise view of possible aliases.
- **Flow-sensitive alias analysis:** Takes control flow into account for more accurate but computationally expensive results.

By seeing the way various parts of the code share memory, it is possible to have an adversary understand the program's logic using alias analysis. It can be used against attack targets and serve as malicious payloads that will fit right in with the application.

2.1.5 Slicing

It is helpful to slice so that it isolates portions of the program based on references to computation or a variable. For instance, a backward slice can be used to describe all instructions that determine a variable value at a certain point, and given some of these examples, these dependencies and parts of code that cannot be safely transformed (or removed) [28]. This lets you tackle attackers isolating relevant code blobs which leaves it possible to execute for

particular feature, function or vulnerability. It lets them see how they can make the data through the program and find the weakness, so they can use information to exploit it. In the slicing case, an adversary can discover which variables are exploited to compromise security critical parts of the code. As an example, some variables that they can mess around so as to have unauthorized access to your system by enabling or disabling the authorization or authentication controls.

2.1.6 Abstract interpretation

Another technique is abstract interpretation, wherein (expressions and) operations of a program are interpreted in a simplified, abstract [28]. It's not working with a bunch of concrete values (numbers), but categories (e.g, negative, zero, positive). This is fine in the sense that it abstracts this so that we can do a more efficient analysis while knowing it's safe (empowered program behavior semantics, transformed semantics). When tracing how data values propagate through a program, attackers use abstract interpretation allowing it to do data flow analysis. They understand how the data holds work, and by implicit that understanding of how they hold the data and handle it - which some of that data may be sensitive (like passwords or tokens) - they can find them and understand how they work and how they handle that data. Things can be made to run, and things that could become known and known what instructions would be invoked to something if it were run. Something that may decide upon some possible states for the variables at some given points of the program. Knowing what types of ranges variables can take allows us to detect the kind of things we might forge (to cause it to behave incorrectly).

2.2 Dynamic Analysis

Dynamic analysis is applied to analyze the software execution, at runtime allowing for real time observation of software behavior, system interaction or runtime issues. Unlike static analysis, it not only discovers vulnerabilities and performance issues that may have fallen beneath the radar of static methods, but instead, provides insights that are deeper in terms of the runtime behaviors of such difficult cases. There are four dynamic analysis techniques explained below: There are four dynamic analysis techniques explained below:

2.2.1 Debugging

Debuggers can execute memory overrides, that allow users to step through code, set breakpoints and make modification to the code. Reverse Debugging (REBB) enables attackers

to backtrack through program's execution and that allows the adversaries to locate important bugs in the source code. Additionally, relative debugging allows the attackers to get te two version of the program compared. It facilitates the identification of specific changes introduced by security mechanism such as obfuscation or temper resistance [28].

2.2.2 Profiling

Attackers understand how often different parts of a program execute and thus, can help profile. It allows you to track down (known) functions that are executed often (crypto routines or license checks) and the others that are rarely used. Attacks that perform frequency spectrum analysis [28] can gain insight to relationships between parts of functions, or to functions themselves that are key to circumventing security mechanisms.

2.2.3 Tracing

Tracing refers to the process through which one can see what instructions are executed so an attacker can analyze the program's behavior without running each line of code. It is helpful for revealing obfuscation that separates the program's logic into its sections. The goal of reconstructing the execution path is to uncover possible patterns and things of interest to the attackers. Managing a large amount of trace data is made possible with compression tools such as SEQUITUR [28]. Attackers can go over and analyze detail-rich areas of the program again to look for flaws.

2.2.4 Emulation

Providing ultimate control over everything related to the software environment, emulators allow the attackers to simulate the hardware platform and make it appear as if the same hardware were being executed. In particular this can be frequently used to circumvent tamper resistant mechanisms as well as to analyze the protected code in real time without tampering with security checks. By finding "hotspots" of certain kinds of machine instructions, such as XOR in encryption algorithms, emulation may disclose hidden or split cryptographic routines [28].

2.3 Side Channel Attack

An attack on a cryptographic device which does not result in a compromise of an algorithm, but rather exploits the physical characteristics of the cryptographic device itself. We aim to get sensitive information from the system such as cryptographic keys using indirect means:

measuring its current, electromagnetic emissions, time or sound. Traditional security mechanisms are bypassed by focusing on hardware implementations of these attacks, which is why they are a serious threat for embedded systems, as they can circumvent most security mechanisms employed. Persial et al. (2011) [21] explained various types of side channel attacks:

Persial et al. (2011) [21] explained various types of side channel attacks:

- **Timing Attacks:** Cryptography attacks that rely on timing variations in the time to perform cryptographic operations can allow the deduction of secret keys information. Attackers are able to gain clues about the data being processed by carefully watching how long some computations take. There are several types:
  - **Simple Power Analysis (SPA):** This method analyzes power traces to directly interpret cryptographic operations.
  - **Differential Power Analysis (DPA):** More sophisticated, DPA involves comparing multiple power traces statistically to find patterns related to the secret keys.
  - **Higher-Order Differential Power Analysis (HODPA):** Combines timing and power analysis, along with advanced cryptanalysis, to break more secure implementations.
  - **Correlation Power Analysis:** Attacks using a power consumption model to relate power consumption and secret data.
- **Electromagnetic Analysis Attacks:** The devices always emit electromagnetic signals while in operation. Similarly, cryptographic secrets can be liberated by deciphering these signals and analyzing them respectively as power consumption. In some examples, Simple and Differential Electromagnetic Analysis (SEMA and DEMA) is performed.
- **Fault Induction Attacks:** Here, the attacker purposely brings in faults into the system to hope for the output to be wrong. The way by which a cryptographic process does. It can be carried out by environmental disruption, for example, abnormal voltage, radiation, or clock manipulation.
- **Optical Side Channel Attacks:** Use visible characteristic of devices to extract information e.g. Visible characteristic such as the light on of the LEDs or monitors. But less common, these can be applied in certain environments.

- **Traffic Analysis Attacks:** These attacks are forms of analysis based on the patterns of communication (traffic flow) between devices to discover. the topology of the system or the importance of our nodes.
- **Acoustic and Thermal Imaging Attacks:**
  - **Acoustic Attacks:** Think about the acoustic signals that come out of things: Acoustic signals from keyboards, computing components.
  - **Thermal Imaging Attacks:** By imaging devices in a heat signature and capturing generated heat signatures to see internal operations in operation.

Side channel attacks are broad and complex as each of these attacks exploits different physical manifestations of cryptographic operations.

2.4 Code Obfuscation

Code obfuscation is a process to make code harder to analyze or reverse engineer. It makes static analysis hard by breaking the code structure and making dynamic analysis hard by the behavior at runtime. Other defenses include obfuscation to protect from algorithms (and sensitive data) being reverse engineered, and from side channel attacks. There are some code obfuscation methods [21] explained below:

- **Control Flow Obfuscation:** Suspends normal flow of a certain program and changes how the program is run or executed. They are as follows: adding opaque predicates, control flow flattening, and dead code insertion.
- **Data Obfuscation:** Centered on concealing the information inherent in a programme, for example, using encryption, or representing the variables by names which do not convey any information.
- **Layout Obfuscation:** Reorganizes physical structure of code such as rearranging the sequence in the instructions, changing variable name or adding unwanted intermediate code segments.
- **Dynamic Obfuscation:** Can only change itself during runtime of a program, and can also change the structure or flow of the program.
- **String Encryption:** Temperately stores string literals and decrypts them at runtime; this ensures that attending personnel cannot view information in raw form.

- **Code Encryption:** Can work with entire blocks of code, with the prior encrypted and decrypted at runtime for enhanced security.
- **Obfuscation of Function Calls:** When functions are defined, it replaces how they are being called making it hard to comprehend the relations between parts of code.
- **Dummy Code Insertion:** Includes extra code so that the code becomes large without the betterment of its performance.

2.5 Limitations and Challenges of Existing Researches

As reverse engineering tools themselves become more sophisticated, the protection of proprietary software becomes more critical in the realm of cybersecurity: using effective code obfuscation. Obfuscation techniques found today, although good, have not always advanced sufficiently in complexity to prevent analysis by advanced actors.

Most relevant obfuscation tools work on single threaded code and are therefore easier to reverse engineer. In addition, little work has been undertaken to investigate how different multithreading patterns affect the effectiveness of obfuscation against modern tools, in particular large scale software systems. Improve on these limitations and these are very important for software security; they would allow much better protection against reverse engineering and unauthorized analysis – particularly in performance critical applications where security matters a lot.

This allows attackers to predict less program flow when multithreading with code obfuscation takes place, increasing complexity. The dynamic analysis is further complicated by obscuring thread interactions and race conditions caused by parallelism. In a way, it contributes more layers of fuzziness into the process if you want to reverse engineer multithreaded code, because to decompile it you need to understand the specific scheduling and execution patterns of those threads.

2.6 Multithreading

Multithreading introduces parallelism by allowing multiple threads to run concurrently, which complicates a program's control flow graph. It enables different functions or segments of code to execute simultaneously, diversifying the paths a program can follow. This increases the code complexity, making it difficult for the intruder to understand and anticipate a program behavior, and hence it provides reward security.

Pthreads is a well accepted standard for implementing multithreading and therefore found wide acceptance. They are important as lightweight thread management with much less

overhead than processes. It is sometimes called lightweight processes (LWP) as each thread shares its resources for context switching faster and more efficiently than switching between a different process. This efficiency is of special importance in parallel application.

POSIX threads are utilized in our work and are critical as they allow for thread based obfuscation. We can however introduce randomness and unpredictability in the program execution by spreading functions across threads. The unpredictable nature of this makes control flow harder to control (and attack) for attackers. Combined, these models allow for varying levels of parallelism with scheduling efficiency.

Scheduler activations further optimize thread management by allowing user level threads to respond to kernel events thereby involving the kernel less in scheduling decisions. It helps to minimize the overhead attached to kernel scheduling on multiprocessor environments. The system improves responsiveness and scalability (%) by managing threads as lightweight processes that share resources

These lightweight processes enhance the security and performance of our obfuscation tool by reducing context-switching overhead while maintaining efficiency. The multithreading approach, combined with the ability to randomly schedule threads, makes the control flow less predictable, further complicating the program's analysis for attackers.

Some Pthreads use different models by which to manage thread connection between user threads and kernel threads: Many to One model where a single user thread can be mapped to a single kernel thread, and One to One model which connects each user thread with its own kernel thread.

Chapter 3

Literature Review

At the data structure level, Collberg et al (1998) first introduced the basis of obfuscation with an article titled, "Breaking Abstractions and Unstructured Data Structures." [9] Where previous work was largely done to identify and establish control flow manipulation, this work stressed on the need to manage data structure, classes, and arrays as nothing but an attempt at reverse engineering. The authors employed changes in dichotomy of control abstraction and changes in class inheritance abstraction that distorts the software structural understanding. These methods were supported due to software complexity metrics such as Chidamber and Munson metrics, which made it possible to build certain quantitative measurements that defined the extent of the effect of the transformations to the code complexity [9].

Consequently, based on the principles of obfuscation, Bhatkar et al. 2003 put forward a second level of inflating the complexity of a program through address obfuscation which introduces certain element of randomness in the memory layout to deter under memory profile attacks such as stack smashes and head overflows [5]. This concept masks base addresses and inserts some form of filler between two variables and this would make it extremely difficult for the attacker to exploit known memory flaws. This is in contrast with StackGuard and ASLR, address obfuscation provides a more general coverage with a relatively small performance penalty and mostly provides a solution against a diverse memory error exploits attack. This systematic defense adds irregular patterns to the memory layout, which significantly increases the level of difficulty for attackers to successfully mount any form of memory exploitation.

Furthermore, Schrittwieser and Katzenbeisser (2011) have offered development of the code diversification techniques that are to complicate the reverse engineering efforts both at the static and dynamic level [34]. Their approach completely separates code into small gadgets and dynamically masked control flow thus making it very difficult to reverse engineer particularly in a dynamic framework. Their approach was shown to have good elasticity in relation to the modern IDA Pro based automated deobfuscation tools. Although not perfect, their work provides a realistic improvement to control flow obfuscation, especially where the attackers have the full rights to monitor the running software [34].

Similarly, Chen and Tang (2013) proposed CONFUSE : code obfuscation tool of LLVM compiler [8]. The first approach implemented in CONFUSE is at the compiler level using

LLVM intermediate representation (IR) and some methods such as control flow flattening, instruction substitution, and dead code insertion. These techniques make it hard for any programmer to reverse-engineer the program's structure and thwart off the two types of analysis. This is true, for instance, in control flow flattening, which obfuscates usual program organization by having no apparent clear-cut hierarchy that is simple for S tools to follow. Opaque predicates further complicate the issue, as they always return true or false, and mislead the dynamic analysis, introducing statements that are always true or false [8]. CONFUSE is quite successful in the scheme of offering performance versus security by eschewing code protection for much of the program but obfuscating only the necessary areas; moreover, the compiler-level integration represents a notable improvement in terms of fluency of obfuscation implementation.

Building on this, Ma et al. (2014) proposed a new method in the control flow of the use of neural networks in obfuscation. Their work was intended to improve defenses against concolic testing, a dynamic analysis method often used in software verification and malware analysis. The method of training neural networks to emulate conditional system behavior effectively thwarts reverse engineering tools, with the least effect on program performance, all to improve software protection. These findings provided evidence that their method helped prevent other tools such as KLEE and TEMU from detecting concealed conditional logic, making an additional addition to the obfuscation dimensionality [24].

In addition, Hataba, Elkhouly, and El-Mahdy (2015) proposed on-demand transformation of conditional branches to effectively avoid such ROP techniques [18]. Their dynamic method of obfuscation incorporates variability at the time of execution not at the time of coding unlike other conventional approaches to obfuscation. This approach greatly hinders the ability of a reverse engineer to work and also weakens the ability of a static analysis. Additionally, dynamic binary instrumentation is enabling this obfuscation to be applied to existing systems, which do not require source level access. As applied, this technique is very effective but does pose the possibility for timing side-channel attacks that the authors propose to counter via random timing delays [18].

Besides these enhancements, Viticchié et al. (2016) also applied a comparative study of numerous source code obfuscation strategies which they divided into control flow, data and layout. To compare the results of different approaches, new criteria, including resistance, concealment, and performance influence, were included in the study. It is noteworthy that the techniques of opaque predicates and control flow flattening were described in view of their application to static analysis tools and at the same time, they entail spiking levels of performance penalties. Data obfuscation methods like renaming of the variables were found to provide only a little resistance to dynamic analysis. Overall, Viticchié et al. (2016) rightly underlined that it requires more than one technique to achieve a successful obfuscation that

provides safety while also trying to prevent harms to performance and the possibility of code maintenance [39]. Their overall scheme is useful to grasp the tensions in security and functionality when applying obfuscation in practical scenarios.

Most recently, Schloegel et al. (2022) presented *Loki* in their paper *"Hardening Code Obfuscation against Automated Attacks"* [33], since automated attacks make it harder to obfuscate code. Authors believe that although virtual machine (VM)-based techniques are used now, they easily fall prey to compiler optimization for automated simplification. In reply, *Loki* uses Mixed Boolean-Arithmetic (MBA) expressions, which make it more difficult to analyze and simplify symbolic expressions. This way, the effectiveness of automatic de-obfuscation is reduced by 19% in program synthesis-based attacks. Besides, the use of both syntax and semantics in *Loki* means that attacking attempts to simplify the code become much more challenging. Therefore, it gives additional protection to important computer systems.

Moving forward towards the future, Bartusek, Brakerski, and Vaikuntanathan (2024) describe the new direction of quantum state obfuscation. They suggest applying quantum-style changes to all the classical crypto techniques, for instance oracles [4]. Kai and their colleagues showed that their research of distillation was difficult because of the difficulty of hiding fundamental quantum entangled states. While they showed impractical projections for completely hiding specific quantum states, they introduced paths for a combination of classical and quantum obfuscation. While the field remains new, the understanding of quantum state obfuscation is likely to play a role for future work in protecting quantum algorithms and data as quantum computers become more widely used (Bartusek et al. 2024) [4].

In a related yet broader study, Balakrishnan and Schulze (2005) and their listing of obfuscation approaches grouped them into control flow, computation, and data abstraction transformation techniques. Their work proposes four measures of obfuscation: potency, resilience, stealth, and cost. They emphasize that 'increasing obscurity of code degrades efficiency of performance' [2]. Moreover, they highlight that even the most complex forms of obfuscation were discussed with reference to the facts stating that any tricks, even the best one, can be unraveled if one has plenty of time and more resources (Balakrishnan & Schulze, 2005) [2].

Expanding from these fundamentals, Barak et al (2012) appreciate the theoretical disadvantages of program obfuscation, especially the impossibility of proactively attaining "ideal' obfuscation. They propose the idea of virtual black-box obfuscation [3], which should protect any information about program functioning while preserving the functions themselves. However, the authors show that for many programs obfuscation to such a level is impossible; thus, denying the capability of obfuscation to prevent reverse engineering in its totality

(Barak et al., 2012) [3]. However, the paper accepts partial obfuscation where adding more features that make reverse engineering difficult is also beneficial to its security. Therefore, there is agreement with Barak et al.'s conclusion (2012) that while obfuscation can be useful in conjunction with such protections, it is ultimately unattainable.

In a more special case, Faruki et al. (2016) exclusively discuss the types of Android code obfuscation that comprise the control flow, data, and layout types [15]. Their evaluation surveys the techniques such as ProGuard and DexGuard assessing them by the degree to which it compels reverse engineering and the performance cost which it applied. This approach resembles Jin et al., 2024, acknowledging that security/efficiency equilibrium is essential [19]. Moreover, Faruki et al. note that anti-malware authors employ such forms of obfuscation to conceal the actual virus code, which makes detection a real challenge (Faruki et al., 2016) [15].

Moving to usability concern, Xu et al. (2017) investigate the decision-making process of trading off between security and usability in program obfuscation [42]. They analyze various levels of obfuscation according to their effects on usability criteria in terms of maintainability performance and user perceived quality. They emphasize and support their argument with the fact that obfuscation penalties should be kept to bare minimum for optimal security and usability and that combining different techniques may enhance the outcome (Xu et al., 2017) [42].

At the same time, Martinelli et al. (2018) pinpoint on the decisive value of the formal verification and, in particular, the method of model checking for revealing and combating with the obfuscation methods [25]. About their evaluation framework, they make use of formal verification in order to identify instances of malicious activity within obfuscated code, pointing out the weakness of both static and dynamic analysis for handling a large number of obfuscation techniques. Therefore, this work adds to the evaluation discourse by demonstrating how model checking can achieve what other techniques do not – uncover latent risks (Martinelli et al., 2018) [25].

In a similar vein, this paper proposes the DoSE tool [37] in order to assess obfuscation techniques, which are aimed at passing simple semantic analysis by defining semantically equivalent regions in binary code. His evaluation is based on three aspects, namely stealth, resilience, and overhead all in an effort to eliminate as much complication as possible only to include it if there is guaranteed performance improvement. Furthermore, Shirazi follows on from the research work attempted in this paper by suggesting, and providing a substantial number of machine learning heuristics [37], that can identify exactly what Shirazi himself describes as opaque predicates; and this is one of the primary forms of obfuscation, that have brought the understanding of how to automate deobfuscating forms a long way along the road (Shirazi, 2019) [37].

According to Jin et al. (2024), a multi-dimensional assessment framework is introduced that covers more than just the standard so there can be a systematic comparison between different obfuscation approaches [19]. They showcase this framework by looking at the various methods code obfuscators use on assembly (ASM) code, underlining resilience, the ability to stand firm during deobfuscation, and stealth, which refers to how easy it is for the code to go undetected during reverse engineering [19]. According to the framework, sometimes increasing obfuscation complexity can reduce security, and thus, the performance of the system should not be greatly affected, and the code should not get too hard to maintain [19].

The method of semantic based code obfuscation [10] was presented by Preda and Giacobazzi (2005) based on abstract interpretation. Unlike simple syntactic transformations, their approach hides program properties at a rather semantic level, proposing a theoretical backdrop to use for classification of the proposed methods concerning complexity and randomness. This semantic viewpoint allows assessing and analyzing numerous forms of obfuscation, as well as developing models for obfuscation, by providing important insights into the program obfuscation (Preda & Giacobazzi, 2005) [10].

In 2017, Kanani et al. put forward *MAZE-OF-CODE* as a method to obfuscate Java code at the Java Virtual Machine's (JVM) intermediate representation level [22]. The process makes it harder for anyone to read the code by changing names and the control structure, thus making it more difficult for static analysis tools. The success of the transformation is checked by considering potency, resilience, the amount needed, and stealth. They help assess the results of obfuscation and highlight where changes can be made (Kanani et al., 2017) [22].

Secondly, Yi et al. (2020) focused on embedded software, and developed a security model for dynamically creating signals that is based on specific control flow and data obfuscation mechanisms suitable for resource-limited contexts [43]. Their lightweight control flow obfuscation introduces optimum security and performance by only scrambling on selective areas of the code. Also, their data obfuscation technique makes use of an encoding process that protects runtime data, making it more secure against memory analysis attacks [43]. In order to support this approach, they raise the idea of applying obfuscation at different stages of development, proving that their methods reduce the overhead while increasing security (Yi et al., 2020).

Even though several researchers have suggested using thread control for obfuscation [29], Omar et al. (2013) managed to do so most directly and effectively. An intruder will find it harder to reverse engineer their code because they break it into small blocks and distribute them across many threads. With this combination, it complicates things for those who might harm the program, since both aspects of the system are harder to analyze. Here, it is shown

that applying multithreading can assist in getting both software safety and obfuscation at the same time [29].

Standardized Pthreads, as defined by POSIX 1003.1c provide an OS independent view of a thread runtime environment and programming interface for shared memory models [27]. Pthreads are described as beneficial in such a sense by Mueller (1993) that is, this kind of threads are light-weighted and so fit to decrease the overhead in thread management especially in programs where high speed parallelism is expected [27]. Compared with the kernel managed threads, Pthreads also has its advantages of less context switching time and overhead; therefore, it also makes sense to be integrated into our obfuscation tool to enhance the performance in addition to achieving the goals for security (Mueller, 1993) [27].

Threads are described as lighter weight than processes because they compete for limited resources such as memory space and entail a low context switching overhead. As pointed out by Mueller (1993) [27], a characteristic as such makes Pthreads effective for systems that have frequent thread creation and destruction as is the case with our obfuscation tool. We can control the threads with the help of Pthreads so that desired response time is achieved along with integration of parallel processing without falling prey to performance degradations related with heavy kernel processes (Mueller, 1993) [27].

POSIX threads need to be merged with the kernel in order to increase system performance through threading models like many threads to one, and one to one. In our case, there is "scheduler activations", a mechanism of interaction between the user level threads and the kernel mentioned by Anderson et al. (1991) [1]. Many-to-one model is highly efficient as many tens pre-empt a single kernel ten but it restricts parallelism by mapping many-user threads to a single kernel thread. On the other hand, the one-to-one model creates a separate kernel thread for each user thread in each of the site's thread of control, which means improved concurrency but at the price of greater overhead. This integration approach proves how kernel support for user level threads processing lead to the improved performance while preserving the flexibility and controllability of the process. Our tool utilizes these threading models for performance and speed while employing POSIX threads for the parallelism required to scale across the system (Anderson et al. 1991) [1].

Chapter 4

Methodology

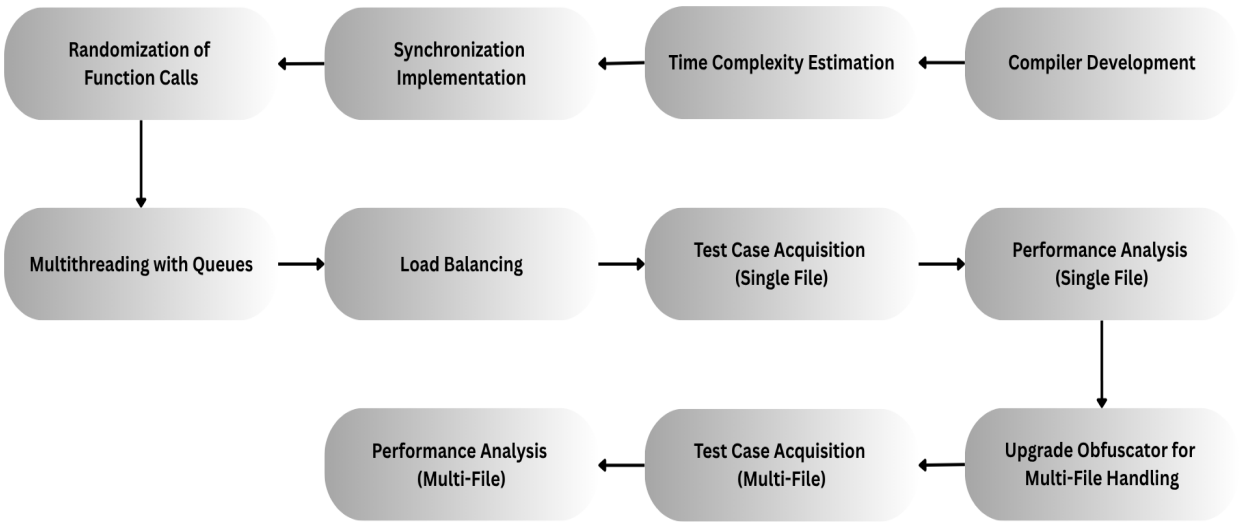

Figure 4.1: Methodology Flowchart

We use a multi-threading-based approach in this study to obfuscate source codes, using synchronization, randomization, and load balancing methods to progress the complexity of code and execution unpredictability. The methodology is explained into the following steps:

We started with the design of a custom, Clang-compiler-based compiler that acts as the important element of our obfuscator. This was a process of setting up the Clang compiler to closely interact with our obfuscation tool, where it reads the source code and uses the multi-threading obfuscation technique available. The flexibility and the ability to accommodate different code structures are the reasons why the compiler was selected.

As a prerequisite to the convenient redistribution of threads and workload distribution, we

estimated the time complexity of the functions of the provided source code. In undertaking this, we have used an AI model that had previously learned data on code analysis to predict the time complexity of functions. These forecasts proved crucial towards making sure that the functions that take longer were given priority in the multithreading arrangement and thus the optimum distribution of the available resources.

Synchronization principal also was put in place to deal with parallel access of shared resources within the obfuscation process. This guaranteed several threads were working in safe ways without leading to race conditioning and data inconsistencies. Protection of critical sections and correct sequencing of the resultant executions of functions were done using suitable synchronization primitives (e.g., mutexes, condition variables).

We have implemented the idea of randomizing function calls in the source code so that the flow of execution of the program becomes non-deterministic and more difficult to reverse-engineer. Randomization is another level of obfuscation because it destroys predictable execution patterns.

The queue data structure was utilized as a multithreading solution to obfuscated unities that are to be executed concurrently. Every function call was considered an independent task, and these tasks were put in a queue and are processed by more than one worker thread.

In order to provide the best performance throughout the obfuscation, we integrated the load-balancing approaches in the queue system. The balance load was carried out relying on the estimation of the time complexity of the functions so that the tasks with greater complexity were divided between the threads properly. This was to avoid bottlenecks in the process of obfuscation and also to provide an even workload on threads.

To test the obfuscator in the initial stages, we obtained single-code-base files that were to be used as our test cases. Then we performed an analysis of the performance of our obfuscator on those test cases.

Successfully testing the obfuscator using a single-file code base, we advanced the obfuscator to support a multi-file code base. This necessitated changes to the tool in order to enable obfuscation of cross-file functions and to manage cross-file dependencies. For the upgraded obfuscator, we acquired multi-file code bases as test cases. Lastly, we did a second performance test on our obfuscator with the multi-file test cases.

Chapter 5

Architecture

In this chapter, we are discussing the implementation and design of a multithreaded task execution framework, in which the obfuscated functions are executed in controlled and synchronized threads. The major issue that has been implemented here is performing different computational operations in the threads when the threads themselves only have a void* return type, can have different numbers of parameters, and synchronize among the functions. Restrictions are overcome with the help of locking systems and looping. The multithreaded mechanism allows users to execute their functions quickly in parallel without losing the sense of safety of threads and the correct synchronization of the threads.

5.1 Design Overview

The obfuscator works on a straight pipeline, as shown in the image. The process begins with the source code, which is then fed into a time complexity estimator that is equipped with the DeepSeek-Coder-V2 model [44]. In this step, each of the functions is analyzed to determine a complexity score depending on the depth of the loops, parameters, and data structures. The approximated complexities are further incorporated in the AST Traverser, and its application is a segmentation of the code into an abstract syntax tree. The foundation of transformation is the ASTMatcher and Clang Rewriter system. The AST Matcher locates target nodes, and the Clang Rewrite module modifies the AST in accordance with them, allowing the insertion of multithreading and structural obfuscations. This loop may proceed as long as all transformation goals are achieved. The output of the transformed AST then generates the obfuscated code, which is, in turn, sent off to be tested so as to prove correctness and integrity. Such design allows methodical examination, barrier change, and behavioral confirmation of the code at the time of the obfuscation (Figure 5.1).

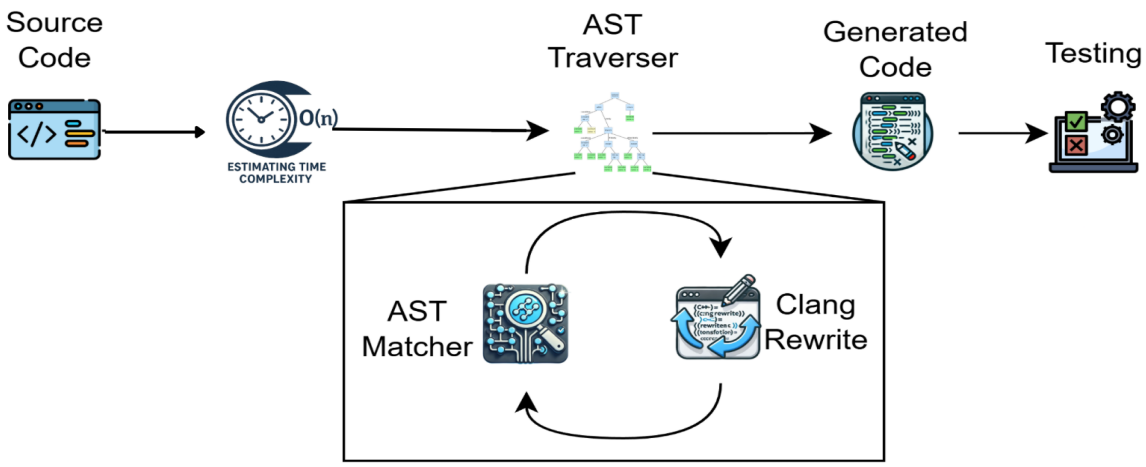

Figure 5.1: Multithreaded Obfuscator Pipeline Post Complexity Estimation

The time complexities of each function should be estimated before the obfuscator can undertake its duties. This is done with an AI model, DeepSeek-Coder-V2 [44], an MoE (Mixture-of-Experts) code language model that analyzes code and recognizes time complexity. It is estimated with indicators like loops, array sizes, and parameters, each having an equation associated with it, indicating the complexity of each function (Figure 5.2).

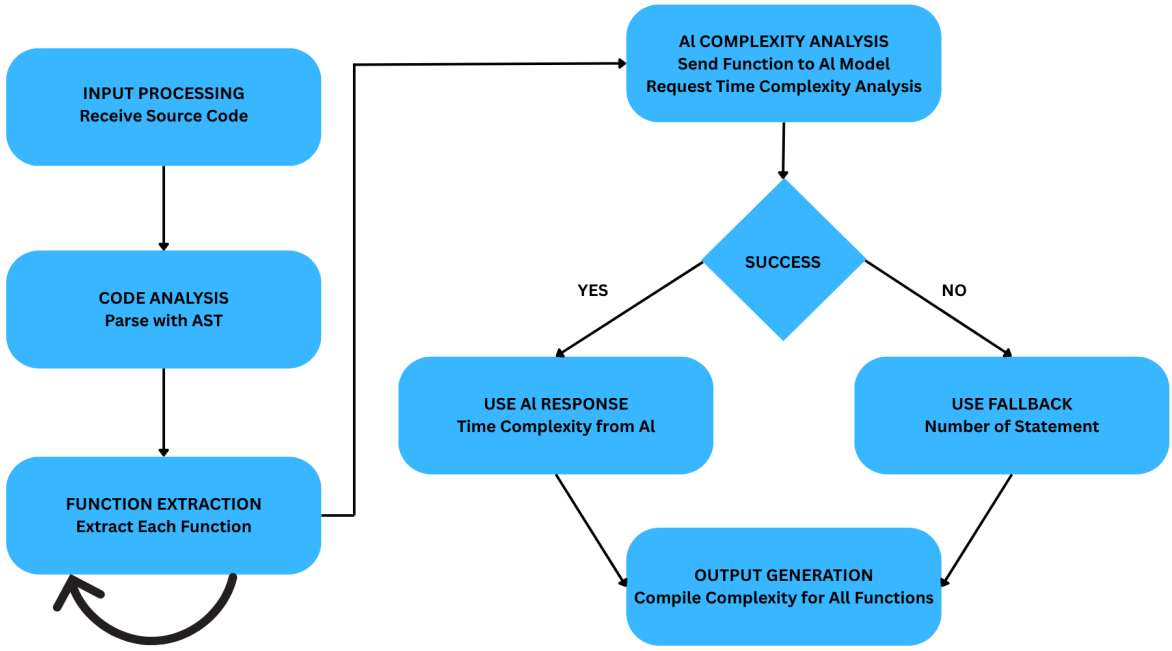

Figure 5.2: Time Complexity Estimation using DeepSeek-Coder-V2

DeepSeek-Coder-V2 is built off the DeepSeek-V2 intermediate checkpoint but has been

trained on an additional 6 trillion tokens to increase the quality of the model in reading and comprehending code [44]. It has more than 338 languages in this programming language and can expand the context size to 128K compared to 16K [44]. By traversing AST, it decomposes the structure of the function into parameters, statement count, body, name etc. and then stores it to a function object and then stores that object to a list. The model then takes every function, runs it, and finds out its time complexity through its sophisticated logical intelligence. Where the model is unable to give a valid result, a fallback approach is taken in which the amount of the statements in the function is taken instead of complexity. The deduced time complexities are later stored in a C++ dictionary format that is an inseparable part of the obfuscation process, and they guarantee consistent estimates in different coding environments.

The obfuscator is based on a multi-file source code, as evidenced by our test case. The following test case consists of five discrete functions: `funcA, funcB, funcC, funcD, and funcE`. After the time complexity estimation stage, the program structure obtained is transformed and restructured based on the complexity values estimated for each one of the functions **Listing 5.1**.

Listing 5.1: Function Time Complexity Mapping

```
inline const std::unordered_map<std::string, std::string>
    cppFunctionsMap = {
     {"funcD_ii", "4"},
     {"funcB", "2 + 1 + 2 + 1"},
     {"funcE_ii", "3"},
     {"funcC", "2 + 3"},
     {"funcA", "4"},
};
```

The rationale of this design is the necessity to perform obfuscated tasks in multiple threads while being able to balance the load among the threads. Every function that is to be run incorporates the computation given the dynamic set of parameters. If we can hide this in a way, we can also achieve data obfuscation. To achieve this, there is a corresponding `struct` that contains the parameters of the function, their returns, and the accomplishment status of the relevant functions in the system. The organization of these `structs` is in global vectors, e.g., `std::vector<funcD_ii_values> funcD_ii_params`, where each index is associated with all the parameters of a particular instance of tasks. This method enables threads to load and modify sets of parameters quickly without passing on complicated information directly to the thread method.

The 2-level index pool: There is an index pool per function to manage memory reuse. This is implemented using a `queue<int>`, a basic queue structure representing a list of indices

corresponding to reusable parameter slots. When a new task is launched, it either reuses an existing index from the pool or allocates a new entry in the respective `vector` if the pool is empty. Once the activity is over, then the index gets back to the pool, hence effective management of the memory. The most specific benefit of this index pooling method is that this system is easily increased because memory is not wasted, and it has limited wasteful allocations. The concerned adoption of the index pool and parameter vectors is illustrated in **Listing 5.2**.

Listing 5.2: Declaration of Index Pools and Parameter Vectors

```
queue<int> funcD_params_index_pool;
queue<int> funcE_params_index_pool;

vector<funcD_values> funcD_params;
vector<funcE_values> funcE_params;
```

5.2 Execution Flow

The initialization step entails the reservation of requisite `mutexes`, `condition_variables`, `queues`, and an array for tracking threads. The execution flow is managed using global atomic task counters and synchronization variables. Once the system is initialized, a fixed number of worker threads (e.g., two) are launched, each entering a continuous execution loop. These threads are responsible for executing tasks assigned to them and carrying out computations specific to each task. The coordination of task execution within each thread is handled by the `threadFunction(int thread_idx)` method, as shown in **Listing 5.3**.

Listing 5.3: Worker Thread Function Implementation

```
void threadFunction(int thread_idx) {
    while (true) {
        {
            std::unique_lock<std::mutex> lock(mutexes[thread_idx]);
            conditions[thread_idx].wait(lock, [&] {
                return !queues[thread_idx].empty() || stopThreads;
            });
        }

        if (stopThreads && queues[thread_idx].empty())
            break;

        execute(thread_idx);
```

```
    }
}
```

Task submission is done based on load balancing among the available threads. The system uses the `getBalancedRandomIndex()` method to determine which thread should execute a new task, as illustrated in **Listing 5.4**. During task preparation, each function initializes the necessary parameters by either creating a new `struct` or reusing an existing one from the global vectors. Once prepared, the task is dispatched to the appropriate thread's task queue using the `pushToThread` method. Each task carries both the function identifier and the parameter index, allowing the thread to locate and operate on the associated parameter set correctly.

Listing 5.4: Balanced Thread Index Selection Function

```
int getBalancedRandomIndex() {
    double sum = 0;
    std::vector<int> values(OBFUSCATION_THREADS);

    for (int i = 0; i < OBFUSCATION_THREADS; i++) {
        values[i] = vec[i].load();
        sum += values[i];
    }

    double avg = sum / OBFUSCATION_THREADS;
    double threshold = avg * 0.8;

    std::vector<int> candidateIndices;

    for (int i = 0; i < OBFUSCATION_THREADS; i++) {
        if (values[i] <= threshold) {
            candidateIndices.push_back(i);
        }
    }

    if (candidateIndices.empty()) {
        std::vector<int> sortedValues = values;
        std::sort(sortedValues.begin(), sortedValues.end());

        int median = sortedValues[OBFUSCATION_THREADS / 2];

        for (int i = 0; i < OBFUSCATION_THREADS; i++) {
```

```
28                 if (values[i] <= median) {
29                     candidateIndices.push_back(i);
30                 }
31             }
32         }
33 
34         std::uniform_int_distribution<int> dist(0, candidateIndices.size
               () - 1);
35         return candidateIndices[dist(rng)];
36 }
```

As soon as a task is dequeued by a thread, the thread begins execution by calling the respective function, such as `funcD_ii`, `funcB`, etc. Each of these functions acquires a lock using a thread-specific `mutex` to safely access shared resources. Once the logic of the function completes, the thread updates the corresponding result in the global `vector` and returns the index to the pool. The task then marks itself as complete, and the global atomic counter `inFlightTasks` is decremented. If all tasks have been completed, the system signals that the computation has concluded.

The logic of each function follows a consistent pattern. For instance, `funcD_ii` begins by locking the associated `mutex` and checking whether there are available indices in the `funcD_ii_params_index_pool`. If the pool is empty, a new `struct` is created and appended to the `funcD_ii_params` vector. The function then performs its computation and marks the task as complete. The used index is returned to the pool for future reuse. Other functions such as `funcB`, `funcE_ii`, and `funcA` follow a similar approach, each executing their specific logic and modifying global variables or results accordingly.

The system is initialized via the `main()` method, where the root task (e.g., `funcA`) is submitted to the task queue. The system's main responsibilities include ensuring all tasks are completed, handling synchronization through `condition_variables`, releasing resources, and terminating the program only after all computations are finalized.

5.3 Function Implementation Summary

The `funcD_ii` function is responsible for acquiring the thread-specific lock, initializing the parameters for `funcE_ii`, and then blocking until the execution of `funcE_ii` completes. Once `funcE_ii` has finished execution, `funcD_ii` proceeds to compute its result using the return value from `funcE_ii`, stores the result, and marks the task as complete by setting the `funcD_ii_done` flag. The index used for the task is then returned to the pool for reuse. The implementation of this logic is shown in **Listing 5.5**.

The `funcB` function follows a similar pattern. It sets up a task to be executed by `funcC`, blocks until that execution is finished, and then signals completion. However, unlike `funcD_ii`, the `funcB` function does not perform any computation on its own—it solely acts as a synchronization wrapper around `funcC`.

Listing 5.5: Function Implementations: funcE_ii and funcD_ii

```
void funcE_ii(int thread_idx, int param_index) {
    cout << "Inside E " << endl;
    {
        unique_lock<mutex> lock(mutexes[thread_idx]);
        x += 5;
    }
    funcE_ii_params[param_index].return_var =
        2 * funcE_ii_params[param_index].a +
        2 * funcE_ii_params[param_index].b;
    funcE_ii_params[param_index].funcE_ii_done = true;
    vec[thread_idx].fetch_sub(3);
}

void funcD_ii(int thread_idx, int param_index) {
    cout << "Start D " << endl;

    int index;
    {
        unique_lock<mutex> lock(mutexes[thread_idx]);
        if (funcE_ii_params_index_pool.empty()) {
            index = funcE_ii_params.size();
            funcE_ii_params.emplace_back();
        } else {
            index = funcE_ii_params_index_pool.front();
            funcE_ii_params_index_pool.pop();
        }
        funcE_ii_params[index] = {2, 3};
    }

    pushToThread(funcE_ii_enumidx, 3, index);

    while (!funcE_ii_params[index].funcE_ii_done) {
        if (!queues[thread_idx].empty())
            execute(thread_idx);
```

```
    }

    int res = funcE_ii_params[index].return_var;

    {
        unique_lock<mutex> lock(mutexes[thread_idx]);
        cout << x << endl;
    }

    cout << "End D " << res << " " << endl;

    funcD_ii_params[param_index].return_var =
        res - funcD_ii_params[param_index].a +
              funcD_ii_params[param_index].b;

    funcE_ii_params_index_pool.push(index);
    funcD_ii_params[param_index].funcD_ii_done = true;
    vec[thread_idx].fetch_sub(4);
}
```

The remaining functions, including `funcC` and `funcA`, follow the same structural framework. Each of these functions performs specific logic, modifies shared or global state (such as updating the variable `x`), and signals the completion of the task. The function parameters are accessed using a parameter index, which points to the corresponding `struct` in the appropriate global `vector`. Once a task is complete, the function updates the status and marks the task as finished. The implementation of `funcC` and `funcA` can be seen in **Listing 5.6**.

The primary function of the system is to initialize the environment and launch the root task. Synchronization is managed through the use of `condition_variables`, which ensures that all threads have completed execution before the program terminates.

Listing 5.6: Function Implementations: funcC and funcA

```
int x = 5;

void funcC(int thread_idx, int param_index) {
    cout << "Start C " << endl;

    int index;
    {
        unique_lock<mutex> lock(mutexes[thread_idx]);
        if (funcD_ii_params_index_pool.empty()) {
```

```
10              index = funcD_ii_params.size();
11              funcD_ii_params.emplace_back();
12          } else {
13              index = funcD_ii_params_index_pool.front();
14              funcD_ii_params_index_pool.pop();
15          }
16          funcD_ii_params[index] = {4, 5};
17      }
18
19      pushToThread(funcD_ii_enumidx, 4, index);
20
21      while (!funcD_ii_params[index].funcD_ii_done) {
22          if (!queues[thread_idx].empty())
23              execute(thread_idx);
24      }
25
26      funcD_ii_params[index].return_var;
27      cout << "End C " << endl;
28
29      funcD_ii_params_index_pool.push(index);
30      vec[thread_idx].fetch_sub(2 + 3);
31  }
32
33  void funcA(int thread_idx, int param_index) {
34      cout << "Start A " << endl;
35
36      int index;
37      {
38          unique_lock<mutex> lock(mutexes[thread_idx]);
39          if (funcB_params_index_pool.empty()) {
40              index = funcB_params.size();
41              funcB_params.emplace_back();
42          } else {
43              index = funcB_params_index_pool.front();
44              funcB_params_index_pool.pop();
45          }
46          funcB_params[index] = {};
47      }
48
49      pushToThread(funcB_enumidx, 2 + 1 + 2 + 1, index);
```

```
50      cout << "End A " << endl;
51
52      vec[thread_idx].fetch_sub(4);
53  }
```

5.4 Thread Synchronization

Thread synchronization in the system is achieved through two fundamental strategies.

First, shared global variables—such as `x`—that the system accesses or modifies are protected through thread-safe utilization by acquiring locks using `std::mutex`. This approach prevents race conditions and ensures that only one thread can work on a shared resource at any given time.

Second, when one function delegates work to another and needs to wait for its result (e.g., when `funcC` invokes `funcD_ii`, or `funcD_ii` calls `funcE_ii` using `pushToThread(...)`), the caller function blocks until the callee function completes. This is implemented by checking a completion flag in the corresponding parameter struct, such as `funcD_ii_params[index]` and `funcD_ii_done`.

Rather than idly spinning and wasting CPU cycles, the thread checks whether there are pending tasks in its own queue and executes them via `execute(thread_idx)`. This keeps the thread productively occupied while waiting on dependencies and preserves the correct execution order without requiring complex signaling mechanisms.

This two-tiered strategy offers both thread-safe access to shared data and efficient synchronization between interdependent tasks. The logic illustrating this behavior can be seen in **Listing 5.7**.

Listing 5.7: Index Allocation and Blocking on funcD_ii Completion

```
1  unique_lock<mutex> lock(mutexes[thread_idx]);
2  if (funcD_ii_params_index_pool.empty()) {
3      index = funcD_ii_params.size();
4      funcD_ii_params.emplace_back();
5  } else {
6      index = funcD_ii_params_index_pool.front();
7      funcD_ii_params_index_pool.pop();
8  }
9  funcD_ii_params[index] = {4, 5};
10
11 pushToThread(funcD_ii_enumidx, 4, index);
12
```

```
while (!funcD_ii_params[index].funcD_ii_done) {
    if (!queues[thread_idx].empty())
        execute(thread_idx);
}

funcD_ii_params[index].return_var;
```

5.5 Advantages of the Design

A lot of scalability also goes into its design, as it is able to increase the number of threads with little or no alteration to the current structure. The need to make memory allocations is often reduced by reusing the vector entry via index pool, thus enhancing memory efficiency. Safe parallelism is provided by introducing lock and while loop, which prevents any conflict or race condition. Data obfuscation is also achieved using struct and vector for parameters. It also has arbitrary logic per-function arrangement, allowing each function flexibility in the way the different tasks in computation are executed. This chapter defines a strong, thread-safe mechanism for running parameterized functions in a concurrent fashion based on index pools and structs to store state. Thread synchronization techniques are also used in the system, ensuring that in implementing parallel tasks, thread safety concepts are maintained and that complicated thread parameter passing mechanisms are avoided. With the system effectively utilizing index pools and structs to manage memory, its scaling capacity is sufficient, and it can run functions in parallel while ensuring proper synchronization.

Chapter 6

Implementation

This chapter reports the design and code obfuscation of the C++ code framework. This framework aims at converting a single-threaded C++ program into its multithreaded version with the addition of obfuscation. It works through function analysis, estimation of time complexity, rewriting with the help of AST, analysis of the multithreaded execution, and lastly, generation of the transformed source and support files. The chapter is arranged within a series of algorithmic modules, and the description of the algorithm implementation follows each of them.

6.1 Algorithm Overview

The entire obfuscation chain is separated into the following main steps:

- `collectInputFiles()` - Visit and gather input source files.
- `estimateFunctionComplexity()` - The computational complexity of extracted functions is estimated.
- `generateComplexityMapping()` - Keep the metadata of complexity mapping in a reusable header.
- `generateObfuscationEngine()` - Auto-generate run-time support files to support thread-safe running.
- `rewriteSourceCode()` - AST-based transformation of C++ functions.
- `saveOutputFiles()` - Export all rewritten and generated code files.

All the procedures are outlined below, and their roles, specific algorithms and applications are explained.

6.2 collectInputFiles()

Purpose:
To carbon copy the traversal of a specified accessible directory and retrieve all the correct C/C++ source codes to analyze.

Algorithm 1 Read all files in the input directory and organize them

```
1: Input: Directory path DIR
2: Output: Two lists, cppFiles and headerFiles
3: Initialize cppFiles as an empty list
4: Initialize headerFiles as an empty list
5: for each file in DIR do
6:     if file is a directory then
7:         Call Read all files in this directory recursively
8:     else
9:         if file ends with .cpp then
10:            Add file to cppFiles
11:        else if file ends with .h or .hpp then
12:            Add file to headerFiles
13:        end if
14:    end if
15: end for
```

Implementation:

This step, applied with the help of `os.walk()` in Python, selects files by their extension. The resulting file paths are stored in categorized lists and later parsed into functions using Clang's AST interfaces.

6.3 Function Complexity Estimation Algorithm

Purpose: In order to examine the logic of any extracted function and approximate its running cost in terms of language runtime,

Algorithm 2 Function Complexity Estimation Algorithm

Input: C++ source files (`cppFiles`)
for each function F in `cppFiles` **do**
    Pull the AST body of F
    Distinguish F's metadata: body, parameters, return type, and statement count
    Give the data to the LLM to estimate complexity
    Get the time complexity from the LLM
    Store the complexity in a result list
    if LLM fails **then**
        Repeat up to 5 times, using the feedback from prior errors
    end if
    if LLM still fails **then**
        Take the number of statements in F as a backup complexity
        Store the backup complexity in the result list
    end if
    Put the result into a map: `cppFunctionsMap`[F.name] $\leftarrow$ complexity expression
end for
Output: `cppFunctionsMap`

Implementation:
All the function bodies are fed to DeepSeek-Coder-V2-Lite to create executable C++ expressions containing the basic operations counts. Failure produces retries, and fallback values default to mere sentence counts.

6.4 generateComplexityMapping()

Purpose:
To embody long-standing complexity metadata in a head file to be used in thread scheduling.

Algorithm 3 Create cpp_functions.h and Initialize Data Structures

Input: None
Output: cpp_functions.h with data structures initialized
Create `cpp_functions.h` file
Initialize `cppFunctionsMap` as a map from function names to expressions of complexity
Initialize `cppFunctionNamesSet` as a set of target functions to obfuscate
End

Implementation:

This step builds a C++ header that encompasses a hash map and a set of functions, which are employed subsequently during rewriting and scheduling steps to ensure coherence in the estimation of thread tasks.

6.5 generateObfuscationEngine()

Purpose:
In order to auto-gen runtime support files capable of executing obfuscated functions in multithreading environments.

Algorithm 4 Obfuscator Thread Management Algorithm

Input: `obfuscator.hpp`, `obfuscator.cpp`
Output: Thread management and task execution for code obfuscation
Step 1: Compile `obfuscator.hpp`
1.1 Define `OBFUSCATION_THREADS`: Fixed number of worker threads
1.2 Define `enum FunctionID` with enumerated values {funcX1_enumidx1, funcX2_enumidx2, ...}
1.3 Define syntactic construct `funcX_values` with attributes: parameters, return_var, done flag
1.4 Declare the following objects:
- thread queues
- mutexes
- condition variables
- atomic counters to monitor thread load
- world job counter

1.5 Declare function prototypes:
- `initialize()`
- `exit()`
- `pushToThread()`
- ...

Step 2: Compile `obfuscator.cpp`
2.1 Function `initialize()`:
- Start threads
- Initialize the atomic load vector

2.2 Function `exit()`:
- Wait for all tasks to finish
- Join threads

2.3 Function `pushToThread(funcId, line_no, param_index)`:
- Push a call to the queue for the respective thread

2.4 Function `getBalancedRandomIndex()`:
- Choose a thread with the minimum load estimate

2.5 Function `threadFunction(thread_idx)`:
- Loop and execute the tasks assigned to the thread identified by `thread_idx`

2.6 Function `execute(thread_idx)`:
- Fetch and execute tasks from the queue for the respective thread

2.7 Function `taskFinished()`:
- Subtract the task counter
- Signal thread exit

Implementation:

The entire thread pool mechanism, load-balancing approaches, and scheduling functions are specified during this process. Vectors, queues, and atomic counters are created to be initialized to deal with parameters of functions, synchronization, and keeping of completion.

6.6 rewriteSourceCode()

Purpose: Restructure each function's language syntax and logic so that it can run inside the multithreaded engine.

Algorithm 5 Multithreaded Function Obfuscation Algorithm

for each function F in cppFiles **do**
  if $F.name \neq$ main **then**
    Change return type of F to void
    Rename F to $F_{\text{firstLetterOfEveryParametersDataType}}$
    Replace parameter list with (int thread_idx, int param_index)
    Redirect all parameter uses to: param_vector[param_index].¡param¿
    Replace return with: param_vector[param_index].return_var =< value >
    Wrap global variable accesses with mutex lock blocks
    Replace internal function calls with:
      Struct instance in parameter vector
      Dispatch using pushToThread()
    if return is needed **then**
      Wait using a .done flag loop while executing other tasks
    end if
  else
    Insert `initialize();` at the start of F
    for each function call in F **do**
      Process function call as described above
    end for
    Insert `exit();` before the return statement
  end if
end for

Implementation:

The AST matcher and rewriting tools in Clang are exploited to work on the function declarations, the types of return, and the parameter spans. The parameters used in the body of the function are redirected into a thread-safe structure. A thread-dispatch code will replace the procedure of calling other internal functions, and the values will be returned by redirecting them via shared variables with `.done` flags, making synchronization mandatory. Mutex blocks are used as guards on global variables. In the case of the primary activity, setting up

a multithreaded environment and tearing it down is done in the middle of the body.

6.7 saveOutputFiles()

Purpose: In order to save every transformed and created file into the project structure.

Algorithm 6 File Saving and Verification

1: Save copies of all rewritten .cpp and .h files in the source directory.
2: Ensure the following files are present:
    cpp_functions.h
    obfuscator.hpp
    obfuscator.cpp

Implementation:

Export is done through normal file I/O functions and forms. This includes header guards, regular alignment and area labelling.

6.8 Build and Execution Environment

The project will have a Makefile that contains rules to automate the compilation so that it can be compiled exactly on any platform and with the least effort. This Makefile connects the required libraries of Clang and LLVM and prepares the required flags. Furthermore, Docker and 'docker-compose.yml' files provide a containerized environment that has compilers, Python packages, and models pre-installed. This ensures that every build is done in a controlled environment without any more system-level dependencies.

6.9 Summary

This chapter describes our final implementation of a C++ obfuscator tool in an end-to-end manner. Whether the recurse file reading the aided degree of complexity estimate or the rewriting of Clang AST and the multithreaded execution of the runtime, the complication of each element has been done in a modular manner. The addition of Makefile and Docker helps boost reproducibility and usability between systems.

Chapter 7

Performance Analysis

We attempted performance analysis on five test cases. The evaluation matrices include Cyclomatic Complexity (CC), Control Flow Graph (CFG), Execution Path Entropy (EPE), Decompilation Complexity, Time Overhead, and Context Switching. We compare original and obfuscated codes to demonstrate how obfuscation adds complexity, interrupts consistent execution and adds non deterministic behavior by using multithreads. The analysis brings forward the effect such changes have on making the code more difficult to examine as well as the performance trade-offs of obfuscation.

7.1 Experimental Setup

To measure the overhead performance and efficiency of our proposed multithreaded code obfuscation method, we performed five C++ test cases. All test cases are individual structural and functional profiles, e.g., simple nested calls to complex system simulations. This diversity allows us to thoroughly evaluate how our obfuscator manages the various control flows, function signatures, standard variables, and input/output activities.

These test cases are: Basic: Shows nested calls of functions that share a global variable. Calculator: Carries out simple arithmetic and formatted output. Task Manager: Emulates the loading, execution, and logging of tasks. Temperature Converter: Performs temperature conversion. Library Manager: Emulates a library system composed of multi-step logic and state-keeping actions.

We have used two versions of each test case: the first unobfuscated multi-file version of the source code and the obfuscated version of this code produced with the help of our obfuscator. The **Table 7.1** summarises the purpose and behaviour of the two variants of each test case.

Table 7.1: Functionalities Across Five Test Cases

Label	**Test Case**	**Functionality (Original)**	**Functionality (Obfuscated)**
Test Case 1	**Basic**	Demonstrates nested function calls. `main()` calls `funcA()`, which calls `funcB()`, then `funcC()`, then `funcD()`, which uses global state and calls `funcE()`.	Each function is wrapped in a multithreaded execution block using parameter pools and thread queues. All calls are scheduled and executed through worker threads.
Test Case 2	**Calculator**	Implements arithmetic functions: `add`, `subtract`, `multiply`, `divide`. The main function performs all operations and prints results.	Each arithmetic operation is threaded. Instead of direct calls, threads manage parameter sets and output results after synchronization. Uses function-specific index pools.
Test Case 3	**Task Manager**	Simulates logger-init, task setup and execution, status logging, and cleanup. Has structured flow of 3 tasks.	Each task operation (`setup`, `compute`, `log`, etc.) is pushed to threads using queues. The full scheduler logic is threaded with balanced thread selection.
Test Case 4	**Temperature Converter**	Converts Celsius to Fahrenheit and Kelvin. Outputs original and converted temperatures with formatting.	Each conversion and print operation is implemented as a task. Execution is managed via thread pool and shared index queues with return variable struct.
Test Case 5	**Library Manager**	Full lifecycle: initialize, borrow, return, donate, fine calculation, and status output. Uses 10 functions with state updates and conditional logic.	Obfuscated version introduces threading for each function. Global state is managed with locking. Complex thread queue orchestration simulates original logic asynchronously.

7.2 Cyclomatic Complexity

To compare the original code base with its obfuscated version, we have determined the Cyclomatic Complexities (CC) [36] of both code bases. The complexity due to the obfuscation was estimated by computing the CC using an open-source tool, Lizard [7], that measures independent execution paths. CC can be found as follows: [36]:

$$\text{CC} = E - N + 2P$$

where E is the number of edges, N is the number of nodes, and P is the number of connected components of a control flow graph of a program. It determines the quantity of independent ways, thus determining the complexity and testable nature of the code.

For basic structured code, the formula can be simplified as [36]:

$$\text{CC} = \text{Number of decisions} + 1$$

Lizard also interprets the computed values using a scoring system, as shown in Table 7.2.

Table 7.2: Cyclomatic Complexity Scoring by Lizard

CC Range	**Rating**	**Interpretation**
1–5	A (Excellent)	Simple and easy to test
6–10	B (Good)	Slightly complex, still manageable
11–15	C (Moderate)	More complex; consider refactoring
16–20	D (High)	Complex; testing and understanding harder
21–30	E (Very High)	Very complex; significant refactoring needed
>30	F (Extreme)	Unmaintainable; should be redesigned

Having established the theoretical foundation of Cyclomatic Complexity and how it quantifies code complexity, we now apply this metric to analyze and compare the complexity of our sample programs. The following section presents the Cyclomatic Complexity results for the original files of the source code before obfuscation.

Figure 7.1 shows in Test Case 1, complexity rises to 3.05 in the obfuscated code over 1.00 in the source code. Test Case 2 is the one that shows a greater increase as the obfuscated code emerges, 5.35 as compared to 1.10 in the source. Test Case 3 displays a smaller growth as well, with the obfuscated one at 3.00 against 1.05 in the source. The highest gain appears at Test Case 5, with the obfuscated code attaining 6.80, as compared to 1.08 by the source

code. On the whole, the obfuscation method effectively increases the level of complexity of all test cases, and Test Case 5 is the one with the most significant increase, hence making the obfuscated piece of code more challenging to study.

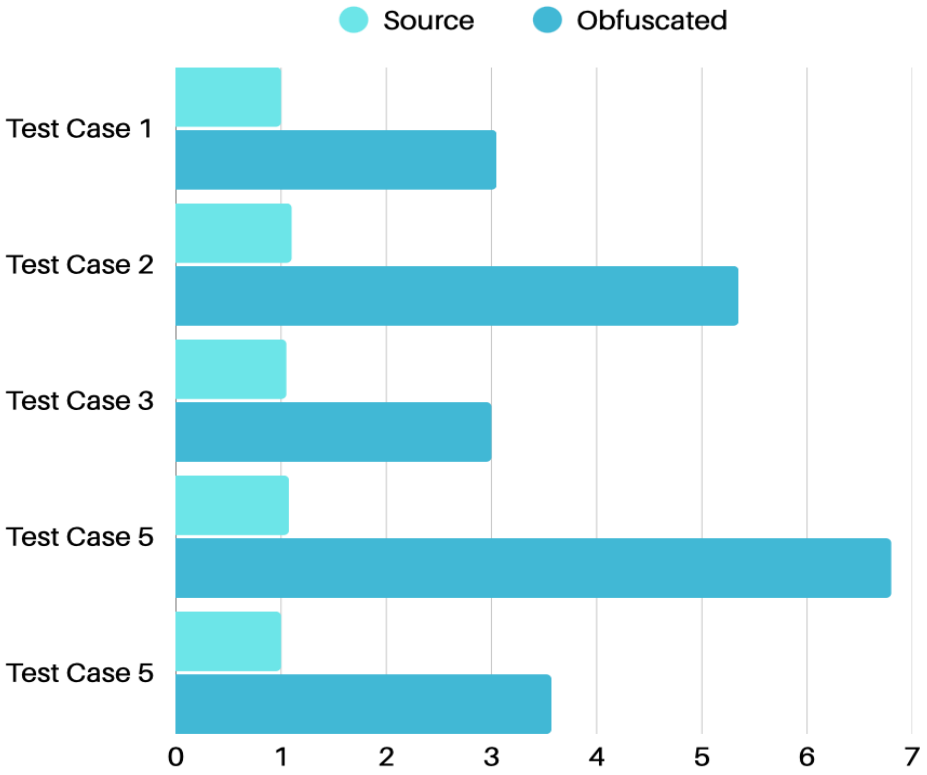

Figure 7.1: Comparison of cyclomatic complexity between the source code and its obfuscated version across five test cases

7.3 Control Flow Graph

In this section, we examine the consequences of obfuscation with multithreading and test its effects on the program run. The original code structure was first examined along with its call graph, and then with the obfuscated code version, where multithreading and randomization jargon have been implemented. The main idea of this assessment will be to show how multithreading not only brings in complexity, but it also complicates the flow of the program relative to the project, and is harder to predict and analyze.

The call graph of the original code is simple (Figure 7.2); it is a straight line. Its path of execution is simple since it calls the functions `funcA, funcB, funcC, funcD`, and `funcE` in succession. Destroying this any-point-can-lead-to-the-other matrix is simple: it is easy to follow the vein of execution, which can be potentially analyzed or reverse-engineered. There

is no multithreading, which implies the lack of parallel execution and randomness in the calls.

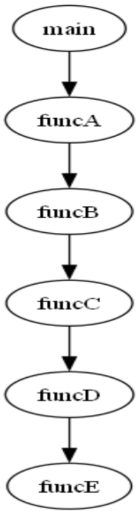

Figure 7.2: CFG of the original program

Multithreading has also been implemented on the obfuscated code in order to make the flow of the execution randomized. The complexities are added through multithreading. In this case, the primary thread will produce several threads (T1, T2, T3) (Figure 7.3), (Figure 7.4), (Figure 7.5), and each of them will run dissimilar sets of functions. Execution of the thread is not deterministic, and the order of calls may differ depending upon the manner in which the threads are scheduled at run time.

Such randomness in implementation is attained through logging the functions performed by each thread, together with the respective thread IDs. The logger functions record the names of the functions upon their entry, together with the thread ID, and hold the sequence of execution script in a text file. This will enable us to see what the execution flow looks like and how the predictable flow, as observed in the original code, is distorted with the introduction of multithreading.

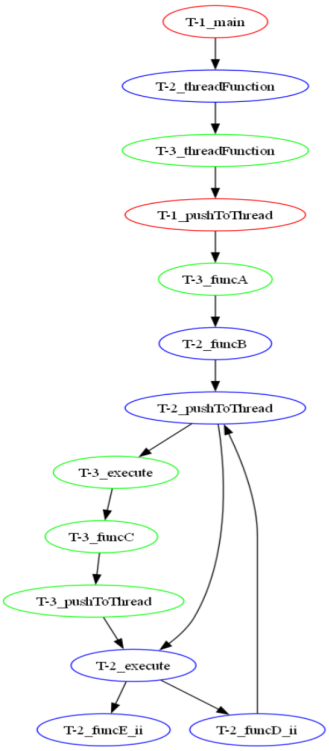

Figure 7.3: CFG of the obfuscated program (1st variant)

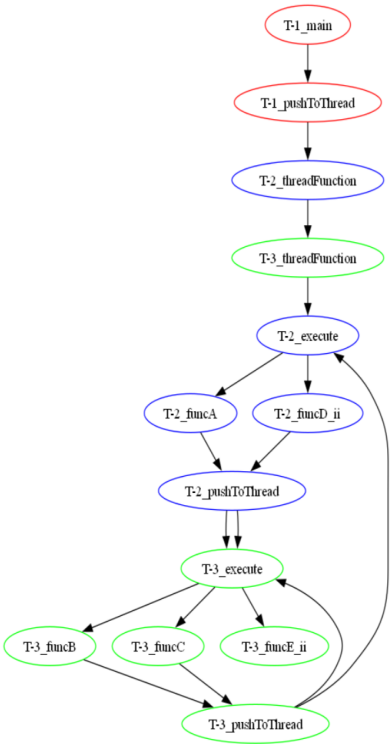

Figure 7.4: CFG of the obfuscated program (2nd variant)

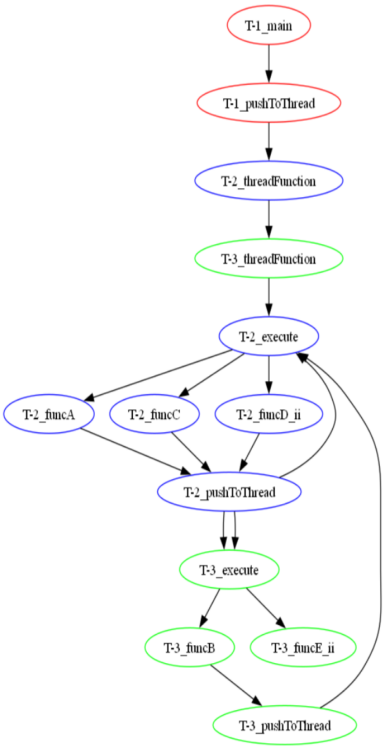

Figure 7.5: CFG of the obfuscated program (3rd variant)

By comparing the original call graph to the obfuscated versions, we can make several observations:

Randomized Execution Order: The order of the function's execution in the obfuscation code is no longer linear. As an example, it is possible to use both T1_pushToThread and T2_threadFunction at the same time, and the functions they may be used in (funcA, funcB, funcC, etc.) may be scattered across and be in separate threads. The reason is that due to the presence of various threads, which execute different operations at the same time, the control flow cannot be predicted.

Multithreading Complexity: There are more difficulties in following the execution flow in the presence of multithreading. As threads might also not be executed in any predetermined order, to analyze the execution of the program, we must have insight into the time each thread is executed and how it should be scheduled. The extra complexity is of use to obscure the logic of the program and indirectly makes the program more resistant to reverse-engineering.

Greater Entropy: The resulting graphs on the obfuscated code have a more complex structure, where there are more edges (function calls). The execution of more threads performing various tasks adds more execution paths, and this fact adds to increased entropy in the execution. Such arbitrariness in the control flow drastically makes the program's actions much more obscure.

To sum it up, the text obfuscation with the help of multithreading becomes beneficial

as it interferes even with the simple sequence of the initial program; it introduces abundant twists and unexpectedness to the potential analysis of the program. The illustration of the call graphs reflects the major peculiarities of the original version and the version of the program obfuscated, which proves that multithreading and randomization could be used to make the behavior of the program unidentifiable and more secure.

These numbers take the form of showing how introducing multithreading presents a form of randomness and complexity, which makes the obfuscated code much harder to analyse and reverse-engineer.

7.4 Entropy of Execution Path

Execution Path Entropy (EPE) [14] is a metric of the uncertainty of the execution path of a program, such as in multithreaded or obfuscated contexts. It is calculated by [14]:

1. **Counting unique address frequencies:** The number of times each instruction location is hit is recorded.

2. **Computing the probability p_i:**

$$p_i = \frac{\text{frequency}_i}{\text{total blocks}}$$

3. **Applying Shannon Entropy:**

$$H = -\sum p_i \log_2(p_i)$$

Higher H values indicate more complex, unpredictable logic. We tested entropy for the original code bases and the obfuscated code bases using the Intel Pin tool to evaluate the control flow's randomness.

This tool allowed us to trace instructions and count their execution in each program series. Instructions taken from several executions were studied using Shannon entropy [14] to measure the variation in execution paths. Each binary was executed 10 times using the Pin instrumentation tool [6], and the instruction counts were logged and analyzed using the entropy H.

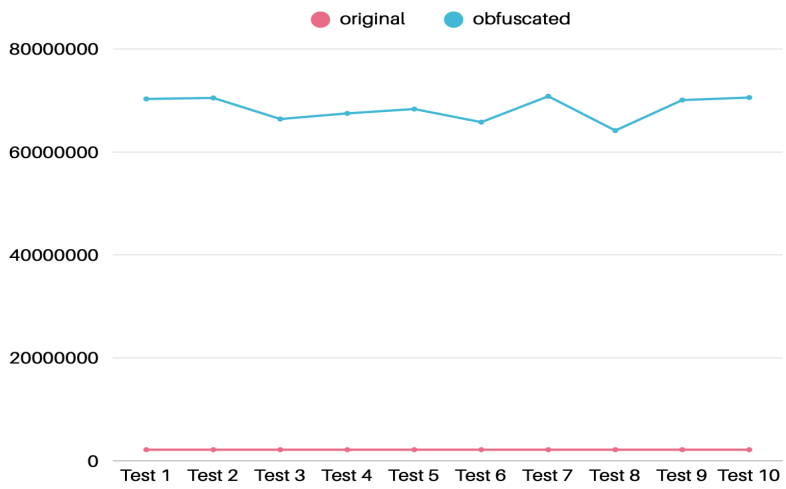

Figure 7.6: Higher entropy observed in obfuscated code base.

The run-time performance shows that there is an observable difference between the original program and the obfuscated program. All instructions in the original program were always 2,165,868, and the Shannon entropy of the execution pattern remained perfectly steady at 1.0000 bits, meaning that the pattern is predictable. The obfuscated program, in its turn, had a greater variance in the number of instructions, between 64,166,897 and 70,825,431, an average of 68,522,287.70 instructions (Figure 7.6) , and an entropy value of 3.3219 bits, indicating a greater level of unpredictability and complexity in the obfuscated and multithreaded application.

7.5 Decompilation Complexity

We applied Ghidra [32] to decompile the source code and its original version and analyzed the complexity between them. Here The decompiled version of two functions is shown below in brief.

Listing 7.1: Decompiled code of funcA() form source file

```
1  /* WARNING: Unknown calling convention -- yet parameter storage is locked */
2  /* funcA() */
3
4  void funcA(void)
5
6  {
7    long *plVar1;
8
9    std::__ostream_insert<>((ostream *)&std::cout,"Start A ",8);
10   plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
11   if (plVar1 != (long *)0x0) {
12     if ((char)plVar1[7] == '\0') {
```

```
13        std::ctype<char>::_M_widen_init();
14        if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
15          (**(code **)(*plVar1 + 0x30))(plVar1,10);
16        }
17      }
18      std::ostream::put('@');
19      std::ostream::flush();
20      funcB();
21      std::__ostream_insert<>((ostream *)&std::cout,"End A ",6);
22      plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
23      if (plVar1 != (long *)0x0) {
24        if ((char)plVar1[7] == '\0') {
25          std::ctype<char>::_M_widen_init();
26          if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
27            (**(code **)(*plVar1 + 0x30))(plVar1,10);
28          }
29        }
30        std::ostream::put('@');
31        std::ostream::flush();
32        return;
33      }
34    }
35                        /* WARNING: Subroutine does not return */
36    std::__throw_bad_cast();
37  }
```

this is a decompilation of `funcA` from the original source code. Here are no parameters needed and the function works with simple flow of control where it calls the `funcB()` and does simple output.

In the code shown below, this is decompilation of `funcA` from the obfuscated file. Which is quite hard to analyze, showing the effectiveness of our obfuscation methods.

Listing 7.2: Decompiled code of obfuscated funcA()

```
1  /* funcA(int, int) */
2
3  void funcA(int param_1,int param_2)
4
5  {
6    long *plVar1;
7    int iVar2;
8    pthread_t pVar3;
9    int *piVar4;
10   ostream *poVar5;
11   long in_FS_OFFSET;
12   timespec local_48;
13   long local_30;
14
15   local_30 = *(long *)(in_FS_OFFSET + 0x28);
16   std::__ostream_insert<>((ostream *)&std::cout,"[funcA] Start on thread ",0x18);
17   pVar3 = pthread_self();
18   if (pVar3 == 0) {
19     poVar5 = (ostream *)&std::cout;
20     std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
           x24);
```

```
21    }
22    else {
23      poVar5 = std::ostream::_M_insert<>(0x109040);
24    }
25    plVar1 = *(long **)(poVar5 + *(long *)(*(long *)poVar5 + -0x18) + 0xf0);
26    if (plVar1 != (long *)0x0) {
27      if ((char)plVar1[7] == '\0') {
28        std::ctype<char>::_M_widen_init();
29        if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
30          (**(code **)(*plVar1 + 0x30))(plVar1,10);
31        }
32      }
33      std::ostream::put((char)poVar5);
34      std::ostream::flush();
35      pushToThread(1,0x14,-1);
36      local_48.tv_sec = 10;
37      local_48.tv_nsec = 0;
38      do {
39        iVar2 = nanosleep(&local_48,&local_48);
40        if (iVar2 != -1) break;
41        piVar4 = __errno_location();
42      } while (*piVar4 == 4);
43      std::__ostream_insert<>((ostream *)&std::cout,"[funcA] End on thread ",0x16);
44      if (pVar3 == 0) {
45        std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
              x24);
46        poVar5 = (ostream *)&std::cout;
47      }
48      else {
49        poVar5 = std::ostream::_M_insert<>(0x109040);
50      }
51      plVar1 = *(long **)(poVar5 + *(long *)(*(long *)poVar5 + -0x18) + 0xf0);
52      if (plVar1 != (long *)0x0) {
53        if ((char)plVar1[7] == '\0') {
54          std::ctype<char>::_M_widen_init();
55          if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
56            (**(code **)(*plVar1 + 0x30))(plVar1,10);
57          }
58        }
59        std::ostream::put((char)poVar5);
60        std::ostream::flush();
61        LOCK();
62        piVar4 = (int *)(vec + (long)param_1 * 4);
63        *piVar4 = *piVar4 + -200;
64        UNLOCK();
65        if (local_30 == *(long *)(in_FS_OFFSET + 0x28)) {
66          return;
67        }
68        goto LAB_00103f26;
69      }
70    }
71    if (local_30 == *(long *)(in_FS_OFFSET + 0x28)) {
72      std::__throw_bad_cast();
73    }
74  LAB_00103f26:
75                        /* WARNING: Subroutine does not return */
76    __stack_chk_fail();
```

```
77 }
```

This is a decompilation of `funcA(int, int)` from the obfuscated source code. This obfuscated code, in turn, removes the original function signature to expect two integer parameters, adds multithreading using objects such as `pthread_t` and `nanosleep`, and adds some other trickiness, including locking (`LOCK()`, `UNLOCK()`) and thread condition checks. The behavior is also changed, now it calls `pushToThread()` and is in charge of timing delays, which denote more complex execution path. The obfuscated version increases error handling by doing extra checks and calls such as `__stack_chk_fail()`. In general, the obfuscation adds threading, injects logical complexity, and distorts control flow, so that the code becomes more difficult to examine than the original.

The obfuscation makes considerable differences between the decompiled codes of the original and obfuscated source code of `funcA`.

7.6 Time Overhead

We measured the time overhead using perf tool [12] in the original and obfuscated source code bases. The data below represent the outcomes on a single test case, such as we have used the original multi-file source code base under the name of simple, and the corresponding obfuscated version on it.

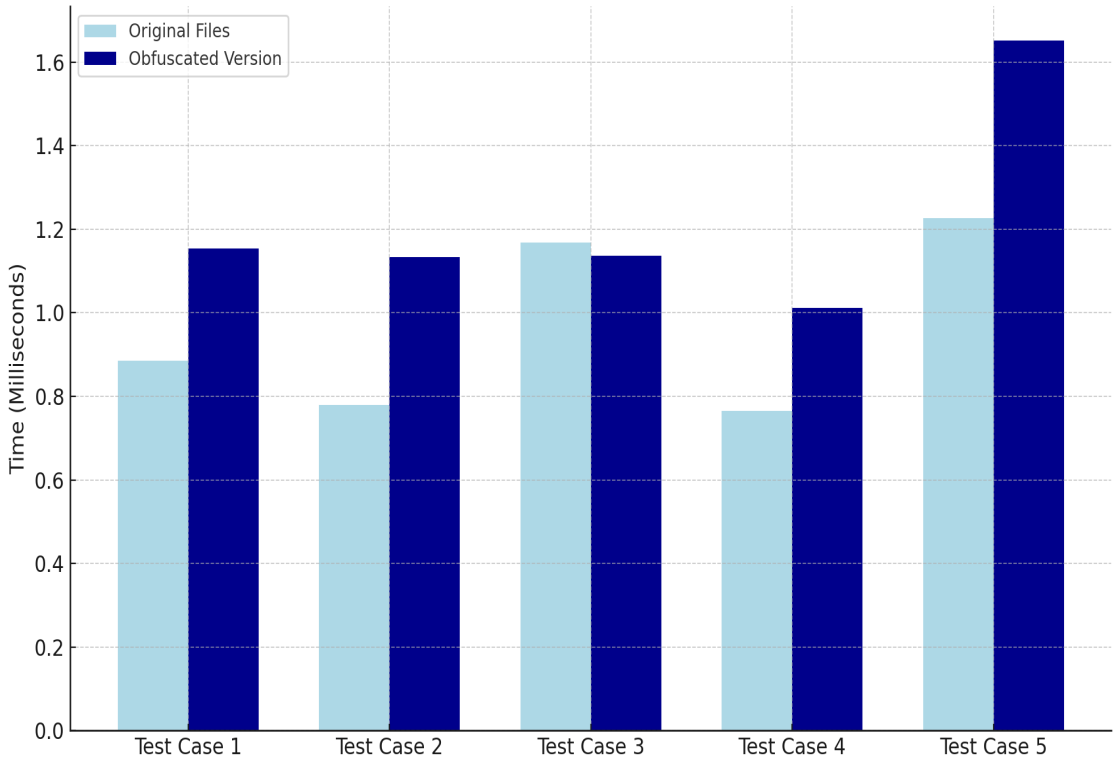

Figure 7.7: Time Overhead Analysis for Orignal and Obfuscated Codes

In Figure 7.7 the graph represents the overhead of time under the comparison of code versions in five test cases between the original code and the obfuscated code. The times are in milliseconds, and the difference brought out shows the effect of performance due to the obfuscation of code. In every test case, we have two bars: one for the time used to execute the program in the original file and the other for the time used to execute the obfuscated copy. Overall, there are greater execution times with the obfuscated versions, and this suggests the additional complexity that was added by obfuscation. As an example, in Test Case 5, the obfuscated version takes nearly 1.65 milliseconds against 1.23 milliseconds in the original one. The graph was developed by using performance profiling tools (read perf) to evaluate the execution times with the purpose of determining the trade-off between code obfuscation and the run-time efficiency. This analysis of performance has been important in the interpretation of the mix between score and computational burdens with obfuscation methods.

We did so on five test cases and averaged out the time overheads of the original source codes and obfuscated ones. A python program was developed to calculate the average time lapse taken in the five multi-file simple source codes and their corresponding obfuscated counterparts. The script will then calculate the percentage increment in time overhead of the obfuscated codes in comparison to the original codes. And the result indicates that the

time overhead is the increase of 35 percent on average, which is typically high.

7.7 Context Switching

We downloaded the perf tool [12] to track the context switching in the original source code bases and in the obfuscated versions of the source code bases. The test was done on the basis of five test cases and the result of one of the test cases was reported below. In case with the initial code base, there was no context switching since the code ran on one thread. Conversely, the obfuscated code had 15 context switches because the multithreading was instigated.

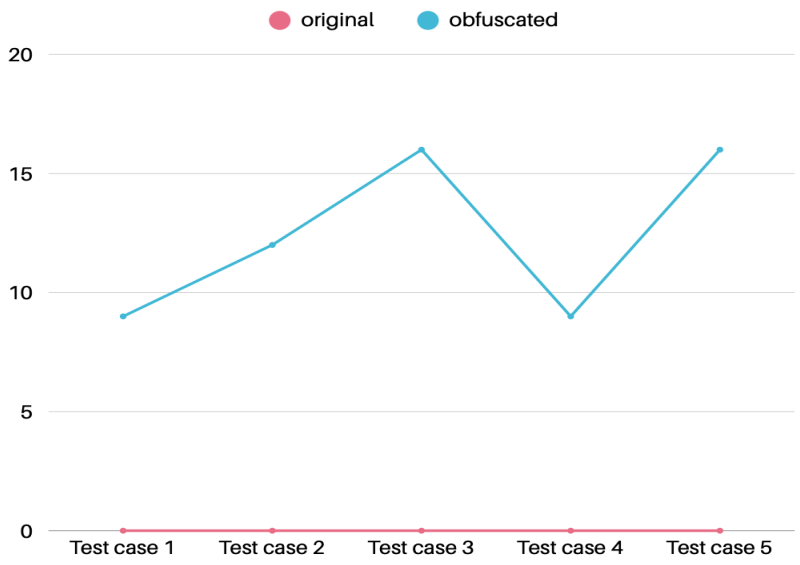

Figure 7.8: Increase of Context Switches in Obfuscated Source Code

The graph in Figure 7.8 indicates a comparison of context switches of the original and obfuscated versions of the source code of five test cases. Context switch count is 0 in all test cases in the case of the original code. Alternatively, the obfuscated code has a different number of context switches: each of the Test Case 1 and the Test Case 4 contains 9 context switches, and each of the Test Case 2 and the Test Case 5 contains 16 context switches, whereas 12 context switches can also be observed in the Test Case 3. This is evident to show that more context switches are created by the obfuscation process, and this depicts how programming becomes more complex and non-deterministic in the obfuscated code.

7.8 Code Similarity

In this evaluation, Joern [20], a tool of static analysis, and Jaccard Similarity [30], a measure of assessing set similarity, are employed to evaluate code obfuscation effectiveness of five test cases. The goal is to measure the extent to which the original program has been modified to its obfuscated version, thus making it harder to reverse engineer. Joern maintains the following code similarity score range:

Similarity Score Range	Interpretation
0	No similarity.
0.1 - 0.3	Low similarity.
0.4 - 0.6	Moderate similarity.
0.7 - 0.9	High similarity.
1	Full similarity.

Table 7.3: Joern Code Similarity Score Range

The tool has examined the quantity of methods (functions or methods), the number of function calls, the utilization of variables in the code, and the existence of control structures where the loops, conditionals, and other control flow providers are involved. These measurements played a vital role in determining the level of change in the structure of the code that resulted due to the obfuscation. Using these values in the original code and the obfuscated code, Joern provided the possibility of an in-depth analysis of how obfuscation affects the overall value of the complexity and readability of the code.

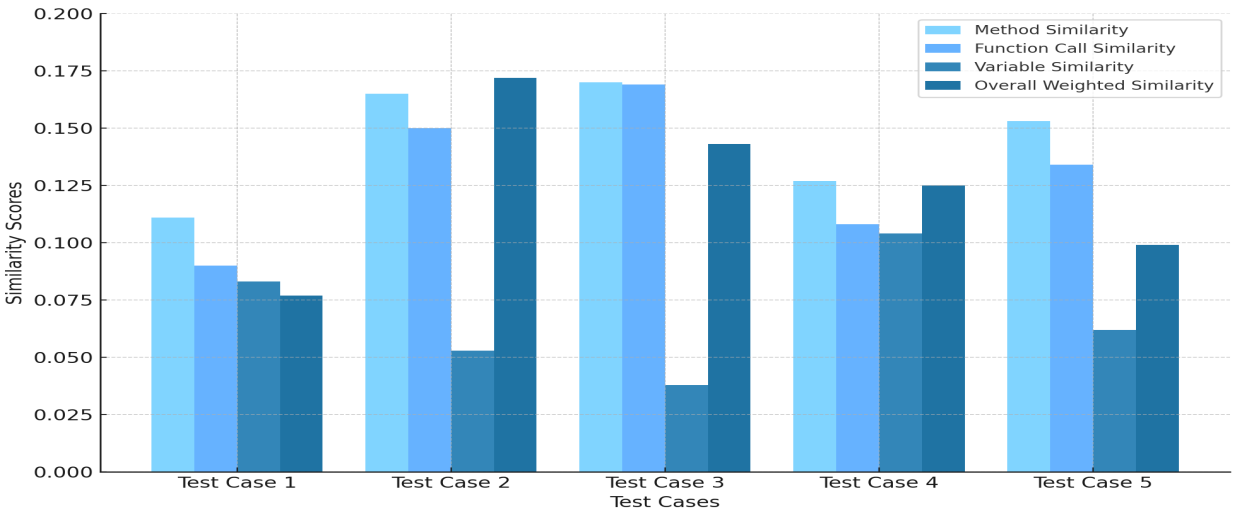

Figure 7.9: Jaccard Similarity for Obfuscated Code in Different Test Cases

The graph (Figure 7.9) identifies the results of a comparison of the similarity metric of a multithreaded obfuscated code according to five test cases with the help of the tools of Joern and the Jaccard similarity measure. The metrics considered are method similarity, function call similarity, variable similarity, and overall weighted similarity. As an example, in Test Case 1, we can see Method Similarity = 0.111 and Variable Similarity = 0.083, which means that it is strongly obfuscated. This is not the case with Test Case 4, though, as its Function Call Similarity value is higher (0.169), which indicates that a certain degree of transformation is lower in that aspect. The less the similarity scores on these metrics, the better the obfuscation is because the code is subjected to a lot of transformation, such that it becomes difficult to reverse-engineer. These findings justify the effectiveness of the obfuscation method to modify the structure of the code without affecting the functionality of the code, thus improving security against statically analyzing the code.

The employed Joern and Jaccard Similarity analysis results indicate that the obfuscation strategies used in the research were effective. A low value of the similarity scores in all the metrics, especially in the domains of variables and calls to functions, proves that the code is substantially changed; therefore, it is not easy to reverse engineer by attackers.

In this chapter, the code obfuscation performance in five test cases was examined based on metrics like cyclomatic complexity, the control flow graph, the entropy of the execution paths, time overhead, context switching, and code similarity. The findings indicate that code obfuscation makes code more complicated; thus, it is more difficult to reverse engineer, particularly through multithreading and randomization. There was also an increase in time overheads and context switching, which is a trade-off between security and performance. Decompilation analysis also made it clear that it is not an easy task to comprehend obfuscated code. In general, code obfuscation is a good and effective way to heat up the code security with some performance penalty.

Chapter 8

Conclusion

This study described a new technique of code obfuscation, which is by introduction of multithreading systems, with the hope of improving the safety of application software. The idea was to make reverse engineering and dynamic analysis more difficult by taking advantage of the complex nature of multithreaded execution. The proposed model contributed to a substantial complication of code behavior comprehension and analysis in real-time and facilitated more complicated reconstruction of the original logic of program, which is used in demonstrating program behavior in Theorist. The multithreaded obfuscation creates the randomness and unpredictability in the code execution, which breaks the traditional static and dynamic analysis tool.We showed that our obfuscator is very effective by merely obfuscating C programs which resulted in a noticeable increase in both structural and logical complexity. Method of evaluation of the code in the form of cyclomatic complexity, entropy of execution paths, and control flow analysis all indicated the significant increase in complexity that was attributed to multithreading. Also, when we evaluated time overhead and context switching, we reinforced the trade-off between increased security and performance.One of the future directions will be to work out performance overheads and improve the obfuscation algorithms to enable practical and faster deployment onto real systems. Moreover, we intend to investigate concurrent deobfuscation methods that would enable attaining a compromise between security and usability in large systems. On the whole, the current study has an implication for the further development of the obfuscation methods and enables making software less susceptible to reverse engineering, offering an efficient means of protecting important software systems.

Bibliography

[1] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 95–109, 1991.

[2] Arini Balakrishnan and Chloe Schulze. Code obfuscation literature survey. *CS701 Construction of compilers*, 19:31, 2005.

[3] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Annual international cryptology conference*, pages 1–18. Springer, 2001.

[4] James Bartusek, Zvika Brakerski, and Vinod Vaikuntanathan. Quantum state obfuscation from classical oracles. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pages 1009–1017, 2024.

[5] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[6] Madhuparna Bhowmik, Madhumitha Nara, and Biju R Mohan. Similarity calculation of executable using intel pin instrumentation framework. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 169–170, 2020.

[7] W. Charoenwet, P. Thongtanunam, V. T. Pham, and C. Treude. An empirical study of static analysis tools for secure code review. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 691–703. ACM, September 2024.

[8] Chih-Fan Chen, Theofilos Petsios, Marios Pomonis, and Adrian Tang. Confuse: Llvm-based code obfuscation, 2013.

[9] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225)*, pages 28–38. IEEE, 1998.

[10] Mila Dalla Preda and Roberto Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Automata, Languages and Programming: 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005. Proceedings 32*, pages 1325–1336. Springer, 2005.

[11] K David. The real story of stuxnet, how kaspersky lab tracked down the malware that stymied irans nuclear-fuel enrichment program. *IEEE Spectrum*, pages 1–6, 2013.

[12] A. C. De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, September 2010.

[13] Antonio Pedro Freitas Fortuna dos Santos, Rui Miguel Silvares Ribeiro, and Filipe Manuel Gomes Silva. Web application protection, October 9 2018. US Patent 10,095,846.

[14] David Ellerman. An introduction to logical entropy and its relation to shannon entropy. *International Journal of Semantic Computing*, 7(02):121–145, 2013.

[15] Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, and Manoj Gaur. Android code protection via obfuscation techniques: past, present and future directions. *arXiv preprint arXiv:1611.10231*, 2016.

[16] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Evaluating control-flow restricting defenses. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 117–137. 2018.

[17] Andres Guadamuz. Trouble with prime numbers: Decss, dvd and the protection of proprietary encryption tools. *Journal of Information, Law & Technology*, 3, 2002.

[18] Muhammad Hataba, Reem Elkhouly, and Ahmed El-Mahdy. Diversified remote code execution using dynamic obfuscation of conditional branches. In *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*, pages 120–127. IEEE, 2015.

[19] Hongjoo Jin, Jiwon Lee, Sumin Yang, Kijoong Kim, and Dong Hoon Lee. A framework to quantify the quality of source code obfuscation. *Applied Sciences*, 14(12):5056, 2024.

[20] Joern. Getting started with static code analysis using joern. https://medium.com/, 2021. Accessed: 2025-06-18.

[21] G Joy Persial, M Prabhu, and R Shanmugalakshmi. Side channel attack-survey. *Int. J. Adv. Sci. Res. Rev*, 1(4):54–57, 2011.

[22] Pratik Kanani, Kriti Srivastava, Janan Gandhi, Disha Parekh, and Meeth Gala. Obfuscation: maze of code. In *2017 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA)*, pages 11–16. IEEE, 2017.

[23] N. Lawson. How the ps3 hypervisor was hacked, January 27 2010. Accessed: 2024-10-13.

[24] Haoyu Ma, Xinjie Ma, Weijie Liu, Zhipeng Huang, Debin Gao, and Chunfu Jia. Control flow obfuscation using neural network to fight concolic testing. In *International Conference on Security and Privacy in Communication Systems*, pages 287–304. Springer, 2014.

[25] Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, Arun Kumar Sangaiah, and Aniello Cimitile. Evaluating model checking for cyber threats code obfuscation identification. *Journal of Parallel and Distributed Computing*, 119:203–218, 2018.

[26] Attila MESTER. Malware analysis and static call graph generation with radare2. *Studia Universitatis Babeș-Bolyai Informatica*, pages 5–20, 2023.

[27] Frank Mueller et al. A library implementation of posix threads under unix. In *Usenix winter*, pages 29–42. Citeseer, 1993.

[28] Jasvir Nagra and Christian Collberg. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection.* Pearson Education, 2009.

[29] Rasha Salah Omar, Ahmed El-Mahdy, and Erven Rohou. Thread-based obfuscation through control-flow mangling. *arXiv preprint arXiv:1311.0044*, 2013.

[30] T. Pang-Ning, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Pearson Addison Wesley, Boston, 2006.

[31] Yuxue Piao, Jin-hyuk Jung, and Jeong Hyun Yi. Structural and functional analyses of proguard obfuscation tool. *The Journal of Korean Institute of Communications and Information Sciences*, 38(8):654–662, 2013.

[32] R. Rohleder. Hands-on ghidra: A tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 77–78, November 2019.

[33] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3055–3073, 2022.

[34] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding: 13th International Conference, IH 2011, Prague, Czech Republic, May 18-20, 2011, Revised Selected Papers 13*, pages 270–284. Springer, 2011.

[35] Leonard Schuetz, Marcel Joss, and Pascal Honegger. *Tama Compiler Overhaul*. PhD thesis, OST Ostschweizer Fachhochschule, 2023.

[36] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[37] Matthieu Tofighi Shirazi. *Analysis of obfuscation transformations on binary code*. PhD thesis, Université Grenoble Alpes, 2019.

[38] Patrick Strookappe. Improving javascript performance using call graphs and graph databases.

[39] Alessio Viticchié, Leonardo Regano, Marco Torchiano, Cataldo Basile, Mariano Ceccato, Paolo Tonella, and Roberto Tiella. Assessment of source code obfuscation techniques. In *2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM)*, pages 11–20. IEEE, 2016.

[40] Daniel Votipka, Mary Nicole Punzalan, Seth M Rabin, Yla Tausczik, and Michelle L Mazurek. An investigation of online reverse engineering community discussions in the context of ghidra. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–20. IEEE, 2021.

[41] Kurtis Wilhoite. Code obfuscation: methods and practicality within automation. 2023.

[42] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R Lyu. On secure and usable program obfuscation: A survey. *arXiv preprint arXiv:1710.01139*, 2017.

[43] Jiajia Yi, Lirong Chen, Haitao Zhang, Yun Li, and Huanyu Zhao. A security model and implementation of embedded software based on code obfuscation. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1606–1613. IEEE, 2020.

[44] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

Appendix A: Source Code Listings

func.h

Listing 1: Header File: func.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <iostream>
using namespace std;

// Declaration of the global variable defined in input.cpp
extern int x;

// Function declarations
int funcE(int a, int b);
void funcC();
void funcA();
int funcD(int a, int b);
void funcB();

#endif
```

funcs.cpp

Listing 2: Implementation File: funcs.cpp

```
#include <iostream>
#include "func.h"

using namespace std;

int funcD(int a, int b){
    cout << "Start D " << endl;

    int res = funcE(2, 3);

    cout << x << endl;
    cout << "End D " << res << " " << endl;
    return res - a + b;
}

void funcB(){
    cout << "Start B " << endl;

    funcC();

    cout << "End B " << endl;
}
```

input.cpp

Listing 3: Main Source File: input.cpp

```cpp
#include <iostream>
#include "func.h"

using namespace std;

int x = 5;

int funcE(int a, int b){
    cout << "Inside E " << endl;
    x += 5;
    return 2 * a + 2 * b;
}

void funcC(){
    cout << "Start C " << endl;

    funcD(4, 5);

    cout << "End C " << endl;
}

void funcA(){
    cout << "Start A " << endl;

    funcB();

    cout << "End A " << endl;
}

int main(){
    funcA();
    return 0;
}
```

Appendix B: Obfuscated Source Code Listings

Obfuscated func.h

Listing 4: Obfuscated Header File: func.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <iostream>
using namespace std;

// Declaration of the global variable defined in input.cpp
extern int x;

// Function declarations
void funcE_ii(int thread_idx, int param_index);
void funcC(int thread_idx, int param_index);
void funcA(int thread_idx, int param_index);
void funcD_ii(int thread_idx, int param_index);
void funcB(int thread_idx, int param_index);

#endif
```

Obfuscated funcs.cpp

Listing 5: Obfuscated Implementation File: funcs.cpp

```
#include "obfuscator.hpp"
#include <iostream>
#include "func.h"

using namespace std;

void funcD_ii(int thread_idx, int param_index){
    cout << "Start D " << endl;

int index;
 {
unique_lock<mutex> lock(mutexes[thread_idx]);
if (funcE_ii_params_index_pool.empty()){
 index = funcE_ii_params.size();
funcE_ii_params.emplace_back();
 }
 else {
 index = funcE_ii_params_index_pool.front();
funcE_ii_params_index_pool.pop();
 }
funcE_ii_params[index] = {2, 3};
 }
;
 pushToThread(funcE_ii_enumidx,3, index);
```

```
while (!funcE_ii_params[index].funcE_ii_done) {
 if(!queues[thread_idx].empty()) execute(thread_idx);
}
    int res = funcE_ii_params[index].return_var;

{ unique_lock<mutex> lock(mutexes[thread_idx]);    cout << x << endl;  }
    cout << "End D " << res << " " << endl;
    funcD_ii_params[param_index].return_var =  res - funcD_ii_params[param_index].a +
        funcD_ii_params[param_index].b;
funcE_ii_params_index_pool.push(index);
funcD_ii_params[param_index].funcD_ii_done = true;
vec[thread_idx].fetch_sub(4);}

void funcB(int thread_idx, int param_index){
    cout << "Start B " << endl;

int index;
 {
unique_lock<mutex> lock(mutexes[thread_idx]);
if (funcC_params_index_pool.empty()){
 index = funcC_params.size();
funcC_params.emplace_back();
 }
 else {
 index = funcC_params_index_pool.front();
funcC_params_index_pool.pop();
 }
funcC_params[index] = {};
 }
;
 pushToThread(funcC_enumidx,2 + 3, index);


    cout << "End B " << endl;
vec[thread_idx].fetch_sub(2 + 1 + 2 + 1);}
```

Obfuscated input.cpp

Listing 6: Obfuscated Main Source File: input.cpp

```
#include "obfuscator.hpp"
#include <iostream>
#include "func.h"

using namespace std;

int x = 5;

void funcE_ii(int thread_idx, int param_index){
    cout << "Inside E " << endl;
{ unique_lock<mutex> lock(mutexes[thread_idx]);    x += 5; }
    funcE_ii_params[param_index].return_var =  2 * funcE_ii_params[param_index].a + 2 *
        funcE_ii_params[param_index].b;
funcE_ii_params[param_index].funcE_ii_done = true;
vec[thread_idx].fetch_sub(3);}
```

```
15
16 void funcC(int thread_idx, int param_index){
17     cout << "Start C " << endl;
18
19 int index;
20  {
21 unique_lock<mutex> lock(mutexes[thread_idx]);
22 if (funcD_ii_params_index_pool.empty()){
23  index = funcD_ii_params.size();
24 funcD_ii_params.emplace_back();
25  }
26  else {
27  index = funcD_ii_params_index_pool.front();
28 funcD_ii_params_index_pool.pop();
29  }
30 funcD_ii_params[index] = {4, 5};
31  }
32 ;
33  pushToThread(funcD_ii_enumidx,4, index);
34 while (!funcD_ii_params[index].funcD_ii_done) {
35  if(!queues[thread_idx].empty()) execute(thread_idx);
36 }
37     funcD_ii_params[index].return_var;
38
39     cout << "End C " << endl;
40 funcD_ii_params_index_pool.push(index);
41 vec[thread_idx].fetch_sub(2 + 3);}
42
43 void funcA(int thread_idx, int param_index){
44     cout << "Start A " << endl;
45
46 int index;
47  {
48 unique_lock<mutex> lock(mutexes[thread_idx]);
49 if (funcB_params_index_pool.empty()){
50  index = funcB_params.size();
51 funcB_params.emplace_back();
52  }
53  else {
54  index = funcB_params_index_pool.front();
55 funcB_params_index_pool.pop();
56  }
57 funcB_params[index] = {};
58  }
59 ;
60  pushToThread(funcB_enumidx,2 + 1 + 2 + 1, index);
61
62
63     cout << "End A " << endl;
64 vec[thread_idx].fetch_sub(4);}
65
66 int main(){initialize();
67 int index;
68  {
69 if (funcA_params_index_pool.empty()){
70  index = funcA_params.size();
71 funcA_params.emplace_back();
```

```
 }
 else {
 index = funcA_params_index_pool.front();
funcA_params_index_pool.pop();
 }
funcA_params[index] = {};
 }
;
 pushToThread(funcA_enumidx,4, index);

    exit();return 0;
}
```

Appendix C: Ghidra Decompilation Report of Original Source Files

__do_global_dtors_aux(void)

Listing 7: Decompiled code of __do_global_dtors_aux(void)

```
void __do_global_dtors_aux(void)

{
  if (completed.0 != '\0') {
    return;
  }
  __cxa_finalize(__dso_handle);
  deregister_tm_clones();
  completed.0 = 1;
  return;
}
```

_fini(void)

Listing 8: Decompiled code of _fini(void)

```
void _fini(void)

{
  return;
}
```

_init(EVP_PKEY_CTX *ctx)

Listing 9: Decompiled code of _init(EVP_PKEY_CTX *ctx)

```
int _init(EVP_PKEY_CTX *ctx)

{
  int iVar1;

  iVar1 = __gmon_start__();
  return iVar1;
}
```

processEntry_start(undefined8 param_1,undefined8 param_2)

_2)

Listing 10: Decompiled code of processEntry_start(undefined8 param_1

```
void processEntry _start(undefined8 param_1,undefined8 param_2)

{
  undefined1 auStack_8 [8];

  __libc_start_main(main,param_2,&stack0x00000008,0,0,param_1,auStack_8);
  do {
                    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}
```

deregister_tm_clones(void)

Listing 11: Decompiled code of deregister_tm_clones(void)

```
/* WARNING: Removing unreachable block (ram,0x00101163) */
/* WARNING: Removing unreachable block (ram,0x0010116f) */

void deregister_tm_clones(void)

{
  return;
}
```

frame_dummy(void)

Listing 12: Decompiled code of frame_dummy(void)

```
void frame_dummy(void)

{
  register_tm_clones();
  return;
}
```

FUN_00101020(void)

Listing 13: Decompiled code of FUN_00101020(void)

```
void FUN_00101020(void)

{
  (*(code *)(undefined *)0x0)();
  return;
}
```

funcA(void)

Listing 14: Decompiled code of funcA(void)

```
/* WARNING: Unknown calling convention -- yet parameter storage is locked */
/* funcA() */

void funcA(void)

{
  long *plVar1;

  std::__ostream_insert<>((ostream *)&std::cout,"Start A ",8);
  plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
  if (plVar1 != (long *)0x0) {
    if ((char)plVar1[7] == '\0') {
      std::ctype<char>::_M_widen_init();
      if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
        (**(code **)(*plVar1 + 0x30))(plVar1,10);
      }
    }
    std::ostream::put('@');
    std::ostream::flush();
    funcB();
    std::__ostream_insert<>((ostream *)&std::cout,"End A ",6);
    plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
    if (plVar1 != (long *)0x0) {
      if ((char)plVar1[7] == '\0') {
        std::ctype<char>::_M_widen_init();
        if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
          (**(code **)(*plVar1 + 0x30))(plVar1,10);
        }
      }
      std::ostream::put('@');
      std::ostream::flush();
      return;
    }
  }
                    /* WARNING: Subroutine does not return */
  std::__throw_bad_cast();
}
```

funcB(void)

Listing 15: Decompiled code of funcB(void)

```
/* WARNING: Unknown calling convention -- yet parameter storage is locked */
/* funcB() */

void funcB(void)

{
  long *plVar1;

```

```
  std::__ostream_insert<>((ostream *)&std::cout,"Start B ",8);
  plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
  if (plVar1 != (long *)0x0) {
    if ((char)plVar1[7] == '\0') {
      std::ctype<char>::_M_widen_init();
      if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
        (**(code **)(*plVar1 + 0x30))(plVar1,10);
      }
    }
    std::ostream::put('@');
    std::ostream::flush();
    funcC();
    std::__ostream_insert<>((ostream *)&std::cout,"End B ",6);
    plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
    if (plVar1 != (long *)0x0) {
      if ((char)plVar1[7] == '\0') {
        std::ctype<char>::_M_widen_init();
        if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
          (**(code **)(*plVar1 + 0x30))(plVar1,10);
        }
      }
      std::ostream::put('@');
      std::ostream::flush();
      return;
    }
  }
                    /* WARNING: Subroutine does not return */
  std::__throw_bad_cast();
}
```

funcC(void)

Listing 16: Decompiled code of funcC(void)

```
/* WARNING: Unknown calling convention -- yet parameter storage is locked */
/* funcC() */

void funcC(void)

{
  long *plVar1;

  std::__ostream_insert<>((ostream *)&std::cout,"Start C ",8);
  plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
  if (plVar1 != (long *)0x0) {
    if ((char)plVar1[7] == '\0') {
      std::ctype<char>::_M_widen_init();
      if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
        (**(code **)(*plVar1 + 0x30))(plVar1,10);
      }
    }
    std::ostream::put('@');
    std::ostream::flush();
    funcD(4,5);
    std::__ostream_insert<>((ostream *)&std::cout,"End C ",6);
```

```
    plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
    if (plVar1 != (long *)0x0) {
      if ((char)plVar1[7] == '\0') {
        std::ctype<char>::_M_widen_init();
        if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
          (**(code **)(*plVar1 + 0x30))(plVar1,10);
        }
      }
      std::ostream::put('@');
      std::ostream::flush();
      return;
    }
  }
                    /* WARNING: Subroutine does not return */
  std::__throw_bad_cast();
}
```

funcD(void)

Listing 17: Decompiled code of funcD(void)

```
/* WARNING: Unknown calling convention -- yet parameter storage is locked */
/* funcD() */

void funcD(void)

{
  long *plVar1;

  std::__ostream_insert<>((ostream *)&std::cout,"Start C ",8);
  plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
  if (plVar1 != (long *)0x0) {
    if ((char)plVar1[7] == '\0') {
      std::ctype<char>::_M_widen_init();
      if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
        (**(code **)(*plVar1 + 0x30))(plVar1,10);
      }
    }
    std::ostream::put('@');
    std::ostream::flush();
    funcD(4,5);
    std::__ostream_insert<>((ostream *)&std::cout,"End C ",6);
    plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
    if (plVar1 != (long *)0x0) {
      if ((char)plVar1[7] == '\0') {
        std::ctype<char>::_M_widen_init();
        if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
          (**(code **)(*plVar1 + 0x30))(plVar1,10);
        }
      }
      std::ostream::put('@');
      std::ostream::flush();
      return;
    }
  }
```

```
35                      /* WARNING: Subroutine does not return */
36    std::__throw_bad_cast();
37  }
```

funcE(void)

Listing 18: Decompiled code of funcE(void)

```
1   /* funcE(int, int) */
2
3   int funcE(int param_1,int param_2)
4
5   {
6     long *plVar1;
7
8     std::__ostream_insert<>((ostream *)&std::cout,"Inside E ",9);
9     plVar1 = *(long **)(&DAT_00104130 + *(long *)(std::cout + -0x18));
10    if (plVar1 == (long *)0x0) {
11                      /* WARNING: Subroutine does not return */
12      std::__throw_bad_cast();
13    }
14    if ((char)plVar1[7] == '\0') {
15      std::ctype<char>::_M_widen_init();
16      if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
17        (**(code **)(*plVar1 + 0x30))(plVar1,10);
18      }
19    }
20    std::ostream::put('@');
21    std::ostream::flush();
22    return (param_1 + param_2) * 2;
23  }
```

main(void)

Listing 19: Decompiled code of main(void)

```
1   undefined8 main(void)
2
3   {
4     funcA();
5     return 0;
6   }
```

register_tm_clones(void)

Listing 20: Decompiled code of register_tm_clones(void)

```
1   /* WARNING: Removing unreachable block (ram,0x001011a4) */
2   /* WARNING: Removing unreachable block (ram,0x001011b0) */
3
```

```
4  void register_tm_clones(void)
5
6  {
7    return;
8  }
```

Appendix D: Ghidra Decompilation Report of Obfuscated Files

__do_global_dtors_aux(void)

Listing 21: Decompiled code of __do_global_dtors_aux(void)

```
void __do_global_dtors_aux(void)

{
  if (completed.0 != '\0') {
    return;
  }
  __cxa_finalize(__dso_handle);
  deregister_tm_clones();
  completed.0 = 1;
  return;
}
```

__tcf_0(void)

Listing 22: Decompiled code of __tcf_0(void)

```
void __tcf_0(void)

{
  std::_Deque_base<>::~_Deque_base((_Deque_base<> *)&DAT_0010bb70);
  std::_Deque_base<>::~_Deque_base((_Deque_base<> *)&queues);
  return;
}
```

__tcf_1(void)

Listing 23: Decompiled code of __tcf_1(void)

```
void __tcf_1(void)

{
  std::condition_variable::~condition_variable((condition_variable *)(conditions + 0x30));
  std::condition_variable::~condition_variable((condition_variable *)conditions);
  return;
}
```

_fini(void)

Listing 24: Decompiled code of _fini(void)

```
void _fini(void)

{
  return;
}
```

_GLOBAL__sub_I_vec(void)

Listing 25: Decompiled code of _GLOBAL__sub_I_vec(void)

```
/* WARNING: Removing unreachable block (ram,0x00102a92) */
/* WARNING: Globals starting with '_' overlap smaller symbols at the same address */

void _GLOBAL__sub_I_vec(void)

{
  undefined8 *puVar1;
  long lVar2;
  uint uVar3;
  void *pvVar4;
  ulong uVar5;
  undefined8 *puVar6;
  long lVar7;
  long in_FS_OFFSET;
  bool bVar8;

  lVar7 = 1;
  lVar2 = *(long *)(in_FS_OFFSET + 0x28);
  puVar6 = &queues;
  do {
    *puVar6 = 0;
    puVar6[1] = 8;
    *(undefined1 (*) [16])(puVar6 + 2) = (undefined1  [16])0x0;
    *(undefined1 (*) [16])(puVar6 + 4) = (undefined1  [16])0x0;
    *(undefined1 (*) [16])(puVar6 + 6) = (undefined1  [16])0x0;
    *(undefined1 (*) [16])(puVar6 + 8) = (undefined1  [16])0x0;
                    /* try { // try from 00102986 to 0010298a has its CatchHandler @ 00102
                       bc0 */
    pvVar4 = operator.new(0x40);
    *puVar6 = pvVar4;
    puVar1 = (undefined8 *)((long)pvVar4 + (puVar6[1] * 4 - 4U & 0xfffffffffffffff8));
                    /* try { // try from 001029a7 to 001029ab has its CatchHandler @ 00102
                       bb4 */
    pvVar4 = operator.new(0x200);
    *puVar1 = pvVar4;
    puVar6[2] = pvVar4;
    puVar6[3] = pvVar4;
    puVar6[4] = (long)pvVar4 + 0x200;
    puVar6[5] = puVar1;
    puVar6[6] = pvVar4;
    puVar6[7] = pvVar4;
    puVar6[8] = (long)pvVar4 + 0x200;
    puVar6[9] = puVar1;
```

```
42      bVar8 = lVar7 != 0;
43      lVar7 = lVar7 + -1;
44      puVar6 = puVar6 + 10;
45    } while (bVar8);
46    __cxa_atexit(__tcf_0,0,&__dso_handle);
47    std::condition_variable::condition_variable((condition_variable *)conditions);
48    std::condition_variable::condition_variable((condition_variable *)(conditions + 0x30));
49    __cxa_atexit(__tcf_1,0,&__dso_handle);
50    std::condition_variable::condition_variable((condition_variable *)g_allTasksDoneCV);
51    __cxa_atexit(std::condition_variable::~condition_variable,g_allTasksDoneCV,&__dso_handle);
52                      /* try { // try from 00102a84 to 00102a88 has its CatchHandler @ 00102
                             bcc */
53    std::random_device::_M_init((string *)rd);
54    __cxa_atexit(std::random_device::~random_device,rd,&__dso_handle);
55    uVar3 = std::random_device::_M_getval();
56    lVar7 = 1;
57    uVar5 = (ulong)uVar3;
58    rng._0_8_ = uVar5;
59    do {
60      uVar5 = (ulong)(((uint)(uVar5 >> 0x1e) ^ (uint)uVar5) * 0x6c078965 + (int)lVar7);
61      *(ulong *)(rng + lVar7 * 8) = uVar5;
62      lVar7 = lVar7 + 1;
63    } while (lVar7 != 0x270);
64    rng._4992_8_ = 0x270;
65    std::queue<>::queue<>((queue<> *)funcD_params_index_pool);
66    __cxa_atexit(std::queue<>::~queue,funcD_params_index_pool,&__dso_handle);
67    std::queue<>::queue<>((queue<> *)funcE_params_index_pool);
68    __cxa_atexit(std::queue<>::~queue,funcE_params_index_pool,&__dso_handle);
69    _funcD_params = (undefined1  [16])0x0;
70    DAT_00109190 = 0;
71    __cxa_atexit(std::vector<>::~vector,&funcD_params,&__dso_handle);
72    DAT_00109170 = 0;
73    _funcE_params = (undefined1  [16])0x0;
74    if (lVar2 != *(long *)(in_FS_OFFSET + 0x28)) {
75                      /* WARNING: Subroutine does not return */
76      __stack_chk_fail();
77    }
78    __cxa_atexit(std::vector<>::~vector,&funcE_params,&__dso_handle);
79    return;
80  }
```

_GLOBAL__sub_I_vec.cold(void)

Listing 26: Decompiled code of _GLOBAL__sub_I_vec.cold(void)

```
1  void _GLOBAL__sub_I_vec.cold(void)
2
3  {
4    undefined8 uVar1;
5    _Deque_base<> *this;
6    ulong unaff_R12;
7    _Deque_base<> *unaff_R13;
8    long in_FS_OFFSET;
9    long param_1;
10
```

```
  __cxa_begin_catch();
  if (param_1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
                    /* try { // try from 00102663 to 00102667 has its CatchHandler @ 001026
                       db */
  uVar1 = __cxa_rethrow();
                    /* catch(type#1 @ 00000000) { ... } // from try @ 0010271c with catch @
                       00102668
                       */
  __cxa_end_catch();
  this = unaff_R13 + (unaff_R12 ^ 1) * 0x50;
  while (this != unaff_R13) {
    this = this + -0x50;
    std::_Deque_base<>::~_Deque_base(this);
  }
  if (param_1 == *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    _Unwind_Resume(uVar1);
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

_init(EVP_PKEY_CTX *ctx)

Listing 27: Decompiled code of _init(EVP_PKEY_CTX *ctx)

```
int _init(EVP_PKEY_CTX *ctx)

{
  int iVar1;

  iVar1 = __gmon_start__();
  return iVar1;
}
```

processEntry _start(undefined8 param_1,undefined8 param_2)

_2)

Listing 28: Decompiled code of processEntry _start(undefined8 param_1

```
void processEntry _start(undefined8 param_1,undefined8 param_2)

{
  undefined1 auStack_8 [8];

  __libc_start_main(main,param_2,&stack0x00000008,0,0,param_1,auStack_8);
  do {
                    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}
```

deregister_tm_clones(void)

Listing 29: Decompiled code of processEntry deregister_tm_clones(void)

```
/* WARNING: Removing unreachable block (ram,0x00102c23) */
/* WARNING: Removing unreachable block (ram,0x00102c2f) */

void deregister_tm_clones(void)

{
  return;
}
```

execute(int param_1)

Listing 30: Decompiled code of processEntry execute(int param_1)

```
/* WARNING: Restarted to delay deadcode elimination for space: ram */
/* execute(int) */

void execute(int param_1)

{
  pthread_mutex_t *__mutex;
  long *plVar1;
  int iVar2;
  long *plVar3;
  pthread_t pVar4;
  undefined8 *puVar5;
  long lVar6;
  int *piVar7;
  long lVar8;
  ostream *poVar9;
  long lVar10;
  int iVar11;
  int iVar12;
  timespec *ptVar13;
  int iVar14;
  ostream *poVar15;
  long in_FS_OFFSET;
  int iStackY_a8;
  int iStackY_94;
  timespec tStackY_90;

  lVar8 = (long)param_1 * 0x50;
  piVar7 = *(int **)(&DAT_0010bb30 + lVar8);
  if (piVar7 == *(int **)(&DAT_0010bb50 + lVar8)) {
    return;
  }
  iVar11 = *piVar7;
  iVar14 = piVar7[1];
  if (piVar7 == (int *)(*(long *)(&DAT_0010bb40 + lVar8) + -8)) {
    operator.delete(*(void **)(&DAT_0010bb38 + lVar8),0x200);
    piVar7 = *(int **)(*(long *)(&DAT_0010bb48 + lVar8) + 8);
```

```
38      *(long *)(&DAT_0010bb48 + lVar8) = *(long *)(&DAT_0010bb48 + lVar8) + 8;
39      *(int **)(&DAT_0010bb38 + lVar8) = piVar7;
40      *(int **)(&DAT_0010bb40 + lVar8) = piVar7 + 0x80;
41    }
42    else {
43      piVar7 = piVar7 + 2;
44    }
45    *(int **)(&DAT_0010bb30 + (long)param_1 * 0x50) = piVar7;
46    switch(iVar11) {
47    case 0:
48      funcA(param_1,iVar14);
49      break;
50    case 1:
51      funcB(param_1,iVar14);
52      break;
53    case 2:
54      funcC(param_1,iVar14);
55      break;
56    case 3:
57      funcD(param_1,iVar14);
58      break;
59    case 4:
60      funcE(param_1,iVar14);
61      break;
62    default:
63      std::__ostream_insert<>((ostream *)&std::cout,"Unknown funcId ",0xf);
64      iVar14 = 0x109040;
65      plVar3 = (long *)std::ostream::operator<<((ostream *)&std::cout,iVar11);
66      plVar1 = *(long **)((long)plVar3 + *(long *)(*plVar3 + -0x18) + 0xf0);
67      if (plVar1 != (long *)0x0) {
68        if ((char)plVar1[7] == '\0') {
69          std::ctype<char>::_M_widen_init();
70          if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
71            (**(code **)(*plVar1 + 0x30))(plVar1,10);
72          }
73        }
74        std::ostream::put((char)plVar3);
75        std::ostream::flush();
76        break;
77      }
78      std::__throw_bad_cast();
79      lVar8 = *(long *)(in_FS_OFFSET + 0x28);
80      std::__ostream_insert<>((ostream *)&std::cout,"[funcD] Start on thread ",0x18);
81      pVar4 = pthread_self();
82      if (pVar4 == 0) {
83        iVar12 = 0x1060f0;
84        iVar2 = 0x109040;
85        poVar9 = (ostream *)&std::cout;
86        std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
              x24);
87      }
88      else {
89        iVar2 = 0x109040;
90        iVar12 = (int)pVar4;
91        poVar9 = std::ostream::_M_insert<>(0x109040);
92      }
93      plVar1 = *(long **)(poVar9 + *(long *)(*(long *)poVar9 + -0x18) + 0xf0);
```

```
94      if (plVar1 != (long *)0x0) {
95        if ((char)plVar1[7] == '\0') {
96          std::ctype<char>::_M_widen_init();
97          if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
98            (**(code **)(*plVar1 + 0x30))(plVar1,10);
99          }
100       }
101       std::ostream::put((char)poVar9);
102       std::ostream::flush();
103       tStackY_90.tv_sec = 1;
104       tStackY_90.tv_nsec = 0;
105       do {
106         ptVar13 = &tStackY_90;
107         iVar2 = nanosleep(&tStackY_90,&tStackY_90);
108         iVar12 = (int)ptVar13;
109         if (iVar2 != -1) break;
110         piVar7 = __errno_location();
111       } while (*piVar7 == 4);
112       lVar10 = (long)iVar14;
113       __mutex = (pthread_mutex_t *)(mutexes + lVar10 * 0x28);
114       tStackY_90.tv_sec = (__time_t)__mutex;
115       iVar2 = pthread_mutex_lock(__mutex);
116       lVar6 = funcE_params;
117       if (iVar2 == 0) {
118         tStackY_90.tv_nsec = CONCAT71(tStackY_90.tv_nsec._1_7_,1);
119         if (funcE_params_index_pool._16_8_ == funcE_params_index_pool._48_8_) {
120           iStackY_a8 = (int)((long)DAT_00109168 - funcE_params >> 4);
121           iStackY_94 = iStackY_a8;
122           if (DAT_00109168 == DAT_00109170) {
123             std::vector<>::_M_realloc_insert<>((vector<> *)&funcE_params);
124             lVar6 = funcE_params;
125           }
126           else {
127             *DAT_00109168 = 0;
128             *(undefined4 *)(DAT_00109168 + 1) = 0;
129             *(undefined1 *)((long)DAT_00109168 + 0xc) = 0;
130             DAT_00109168 = DAT_00109168 + 2;
131           }
132         }
133         else {
134           iStackY_a8 = *(int *)funcE_params_index_pool._16_8_;
135           iStackY_94 = iStackY_a8;
136           if (funcE_params_index_pool._16_8_ == funcE_params_index_pool._32_8_ + -4) {
137             operator.delete((void *)funcE_params_index_pool._24_8_,0x200);
138             funcE_params_index_pool._16_8_ = *(long *)(funcE_params_index_pool._40_8_ + 8);
139             funcE_params_index_pool._40_8_ = funcE_params_index_pool._40_8_ + 8;
140             funcE_params_index_pool._32_8_ = funcE_params_index_pool._16_8_ + 0x200;
141             lVar6 = funcE_params;
142             funcE_params_index_pool._24_8_ = funcE_params_index_pool._16_8_;
143           }
144           else {
145             funcE_params_index_pool._16_8_ = funcE_params_index_pool._16_8_ + 4;
146           }
147         }
148         puVar5 = (undefined8 *)(lVar6 + (long)iStackY_a8 * 0x10);
149         *puVar5 = 0x300000002;
150         *(undefined4 *)(puVar5 + 1) = 0;
```

```
151         *(undefined1 *)((long)puVar5 + 0xc) = 0;
152         pthread_mutex_unlock(__mutex);
153         pushToThread(4,0x14,iStackY_a8);
154         while (lVar6 = funcE_params + (long)iStackY_a8 * 0x10, *(char *)(lVar6 + 0xc) == '\0
                ') {
155           do {
156           } while (*(long *)(&DAT_0010bb50 + lVar10 * 0x50) ==
157                    *(long *)(&DAT_0010bb30 + lVar10 * 0x50));
158           execute(iVar14);
159         }
160         iVar14 = *(int *)(lVar6 + 8);
161         piVar7 = (int *)(funcD_params + (long)iVar11 * 0x10);
162         piVar7[2] = (iVar14 - *piVar7) + piVar7[1];
163         std::__ostream_insert<>((ostream *)&std::cout,"[funcD] End on thread ",0x16);
164         poVar9 = (ostream *)std::ostream::operator<<((ostream *)&std::cout,iVar14);
165         std::__ostream_insert<>(poVar9," ",1);
166         if (pVar4 == 0) {
167           iVar12 = 0x1060f0;
168           poVar15 = poVar9;
169           std::__ostream_insert<>(poVar9,"thread::id of a non-executing thread",0x24);
170           iVar2 = (int)poVar15;
171         }
172         else {
173           poVar15 = poVar9;
174           iVar12 = (int)pVar4;
175           poVar9 = std::ostream::_M_insert<>((ulong)poVar9);
176           iVar2 = (int)poVar15;
177         }
178         plVar1 = *(long **)(poVar9 + *(long *)(*(long *)poVar9 + -0x18) + 0xf0);
179         if (plVar1 != (long *)0x0) {
180           if ((char)plVar1[7] == '\0') {
181             std::ctype<char>::_M_widen_init();
182             if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
183               (**(code **)(*plVar1 + 0x30))(plVar1,10);
184             }
185           }
186           std::ostream::put((char)poVar9);
187           std::ostream::flush();
188           if (funcE_params_index_pool._48_8_ == funcE_params_index_pool._64_8_ + -4) {
189             std::deque<>::_M_push_back_aux<int_const&>
190                       ((deque<> *)funcE_params_index_pool,&iStackY_94);
191           }
192           else {
193             *(int *)funcE_params_index_pool._48_8_ = iStackY_a8;
194             funcE_params_index_pool._48_8_ = funcE_params_index_pool._48_8_ + 4;
195           }
196           *(undefined1 *)(funcD_params + 0xc + (long)iVar11 * 0x10) = 1;
197           LOCK();
198           piVar7 = (int *)(vec + lVar10 * 4);
199           *piVar7 = *piVar7 + -0x14;
200           UNLOCK();
201           if (lVar8 == *(long *)(in_FS_OFFSET + 0x28)) {
202             return;
203           }
204           goto LAB_00104a22;
205         }
206       }
```

```
207         else {
208           if (lVar8 != *(long *)(in_FS_OFFSET + 0x28)) goto LAB_00104a22;
209           std::__throw_system_error(iVar2);
210         }
211       }
212       if (lVar8 == *(long *)(in_FS_OFFSET + 0x28)) {
213         std::__throw_bad_cast();
214         funcD(iVar2,iVar12,lVar8);
215         return;
216       }
217 LAB_00104a22:
218                       /* WARNING: Subroutine does not return */
219       __stack_chk_fail();
220     }
221     taskFinished();
222     return;
223 }
```

frame_dummy(void)

Listing 31: Decompiled code of processEntry frame_dummy(void)

```
1  void frame_dummy(void)
2
3  {
4    register_tm_clones();
5    return;
6  }
```

FUN_00102020(void)

Listing 32: Decompiled code of FUN_00102020(void)

```
1  void FUN_00102020(void)
2
3  {
4    (*(code *)(undefined *)0x0)();
5    return;
6  }
```

FUN_00102539(void)

Listing 33: Decompiled code of FUN_00102539(void)

```
1  void FUN_00102539(void)
2
3  {
4    vector<> *unaff_R13;
5    long in_FS_OFFSET;
6    long param_3;
```

```

  std::vector<>::~vector((vector<> *)&stack0x00000060);
  std::vector<>::~vector(unaff_R13);
  std::vector<>::~vector((vector<> *)&stack0x00000020);
  if (param_3 == *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    _Unwind_Resume();
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

FUN_001025c2(void)

Listing 34: Decompiled code of FUN_001025c2(void)

```
void FUN_001025c2(void)

{
  long in_FS_OFFSET;
  char param_1;
  long in_stack_00000020;
  long in_stack_00000028;
  long in_stack_00000038;

  if (param_1 != '\0') {
    std::unique_lock<std::mutex>::unlock();
  }
  if ((in_stack_00000028 == 0) && (in_stack_00000020 == 0)) {
    if (in_stack_00000038 == *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
      _Unwind_Resume();
    }
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
                    /* WARNING: Subroutine does not return */
  std::terminate();
}
```

FUN_001025c2(void)

Listing 35: Decompiled code of FUN_001025c2(void)

```
void FUN_001025c2(void)

{
  long in_FS_OFFSET;
  char param_1;
  long in_stack_00000020;
  long in_stack_00000028;
  long in_stack_00000038;
```

```
9
10    if (param_1 != '\0') {
11      std::unique_lock<std::mutex>::unlock();
12    }
13    if ((in_stack_00000028 == 0) && (in_stack_00000020 == 0)) {
14      if (in_stack_00000038 == *(long *)(in_FS_OFFSET + 0x28)) {
15                        /* WARNING: Subroutine does not return */
16        _Unwind_Resume();
17      }
18                        /* WARNING: Subroutine does not return */
19      __stack_chk_fail();
20    }
21                        /* WARNING: Subroutine does not return */
22    std::terminate();
23  }
```

funcA(int param_1,int param_2)

_2)

Listing 36: Decompiled code of funcA(int param_1

```
1   /* funcA(int, int) */
2
3   void funcA(int param_1,int param_2)
4
5   {
6     long *plVar1;
7     int iVar2;
8     pthread_t pVar3;
9     int *piVar4;
10    ostream *poVar5;
11    long in_FS_OFFSET;
12    timespec local_48;
13    long local_30;
14
15    local_30 = *(long *)(in_FS_OFFSET + 0x28);
16    std::__ostream_insert<>((ostream *)&std::cout,"[funcA] Start on thread ",0x18);
17    pVar3 = pthread_self();
18    if (pVar3 == 0) {
19      poVar5 = (ostream *)&std::cout;
20      std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
            x24);
21    }
22    else {
23      poVar5 = std::ostream::_M_insert<>(0x109040);
24    }
25    plVar1 = *(long **)(poVar5 + *(long *)(*(long *)poVar5 + -0x18) + 0xf0);
26    if (plVar1 != (long *)0x0) {
27      if ((char)plVar1[7] == '\0') {
28        std::ctype<char>::_M_widen_init();
29        if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
30          (**(code **)(*plVar1 + 0x30))(plVar1,10);
31        }
32      }
```

```
33      std::ostream::put((char)poVar5);
34      std::ostream::flush();
35      pushToThread(1,0x14,-1);
36      local_48.tv_sec = 10;
37      local_48.tv_nsec = 0;
38      do {
39        iVar2 = nanosleep(&local_48,&local_48);
40        if (iVar2 != -1) break;
41        piVar4 = __errno_location();
42      } while (*piVar4 == 4);
43      std::__ostream_insert<>((ostream *)&std::cout,"[funcA] End on thread ",0x16);
44      if (pVar3 == 0) {
45        std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
              x24);
46        poVar5 = (ostream *)&std::cout;
47      }
48      else {
49        poVar5 = std::ostream::_M_insert<>(0x109040);
50      }
51      plVar1 = *(long **)(poVar5 + *(long *)(*(long *)poVar5 + -0x18) + 0xf0);
52      if (plVar1 != (long *)0x0) {
53        if ((char)plVar1[7] == '\0') {
54          std::ctype<char>::_M_widen_init();
55          if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
56            (**(code **)(*plVar1 + 0x30))(plVar1,10);
57          }
58        }
59        std::ostream::put((char)poVar5);
60        std::ostream::flush();
61        LOCK();
62        piVar4 = (int *)(vec + (long)param_1 * 4);
63        *piVar4 = *piVar4 + -200;
64        UNLOCK();
65        if (local_30 == *(long *)(in_FS_OFFSET + 0x28)) {
66          return;
67        }
68        goto LAB_00103f26;
69      }
70    }
71    if (local_30 == *(long *)(in_FS_OFFSET + 0x28)) {
72      std::__throw_bad_cast();
73    }
74  LAB_00103f26:
75                        /* WARNING: Subroutine does not return */
76    __stack_chk_fail();
77  }
```

funcB(int param_1,int param_2)

_2)

Listing 37: Decompiled code of funcB(int param_1

```
1  /* funcB(int, int) */
2
```

```
3  void funcB(int param_1,int param_2)
4
5  {
6    long *plVar1;
7    int iVar2;
8    pthread_t pVar3;
9    int *piVar4;
10   ostream *poVar5;
11   long in_FS_OFFSET;
12   timespec local_48;
13   long local_30;
14
15   local_30 = *(long *)(in_FS_OFFSET + 0x28);
16   std::__ostream_insert<>((ostream *)&std::cout,"[funcB] Start on thread ",0x18);
17   pVar3 = pthread_self();
18   if (pVar3 == 0) {
19     poVar5 = (ostream *)&std::cout;
20     std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
           x24);
21   }
22   else {
23     poVar5 = std::ostream::_M_insert<>(0x109040);
24   }
25   plVar1 = *(long **)(poVar5 + *(long *)(*(long *)poVar5 + -0x18) + 0xf0);
26   if (plVar1 != (long *)0x0) {
27     if ((char)plVar1[7] == '\0') {
28       std::ctype<char>::_M_widen_init();
29       if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
30         (**(code **)(*plVar1 + 0x30))(plVar1,10);
31       }
32     }
33     std::ostream::put((char)poVar5);
34     std::ostream::flush();
35     pushToThread(2,0x14,-1);
36     local_48.tv_sec = 2;
37     local_48.tv_nsec = 0;
38     do {
39       iVar2 = nanosleep(&local_48,&local_48);
40       if (iVar2 != -1) break;
41       piVar4 = __errno_location();
42     } while (*piVar4 == 4);
43     std::__ostream_insert<>((ostream *)&std::cout,"[funcB] End on thread ",0x16);
44     if (pVar3 == 0) {
45       std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
             x24);
46       poVar5 = (ostream *)&std::cout;
47     }
48     else {
49       poVar5 = std::ostream::_M_insert<>(0x109040);
50     }
51     plVar1 = *(long **)(poVar5 + *(long *)(*(long *)poVar5 + -0x18) + 0xf0);
52     if (plVar1 != (long *)0x0) {
53       if ((char)plVar1[7] == '\0') {
54         std::ctype<char>::_M_widen_init();
55         if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
56           (**(code **)(*plVar1 + 0x30))(plVar1,10);
57         }
```

```
58        }
59        std::ostream::put((char)poVar5);
60        std::ostream::flush();
61        LOCK();
62        piVar4 = (int *)(vec + (long)param_1 * 4);
63        *piVar4 = *piVar4 + -0x14;
64        UNLOCK();
65        if (local_30 == *(long *)(in_FS_OFFSET + 0x28)) {
66          return;
67        }
68        goto LAB_00103cce;
69      }
70    }
71    if (local_30 == *(long *)(in_FS_OFFSET + 0x28)) {
72      std::__throw_bad_cast();
73    }
74  LAB_00103cce:
75                      /* WARNING: Subroutine does not return */
76    __stack_chk_fail();
77  }
```

funcC(int param_1,int param_2,long param_3)

_2_3)

Listing 38: Decompiled code of funcC(int param_1

```
1  /* funcC(int, int) [clone .cold] */
2
3  void funcC(int param_1,int param_2,long param_3)
4
5  {
6    long in_FS_OFFSET;
7
8    std::unique_lock<std::mutex>::unlock();
9    if (param_3 == *(long *)(in_FS_OFFSET + 0x28)) {
10                     /* WARNING: Subroutine does not return */
11     _Unwind_Resume();
12   }
13                     /* WARNING: Subroutine does not return */
14   __stack_chk_fail();
15 }
```

funcC(int param_1,int param_2)

_2)

Listing 39: Decompiled code of funcC(int param_1

```
1  /* WARNING: Restarted to delay deadcode elimination for space: ram */
2  /* funcC(int, int) */
3
4  void funcC(int param_1,int param_2)
5
```

```
{
  pthread_mutex_t *__mutex;
  long *plVar1;
  int iVar2;
  pthread_t pVar3;
  undefined8 *puVar4;
  long lVar5;
  int *piVar6;
  ostream *poVar7;
  long lVar8;
  int iVar9;
  timespec *ptVar10;
  ostream *poVar11;
  long in_FS_OFFSET;
  int local_7c;
  int local_5c;
  timespec local_58;
  long lVar12;

  lVar12 = *(long *)(in_FS_OFFSET + 0x28);
  std::__ostream_insert<>((ostream *)&std::cout,"[funcC] Start on thread ",0x18);
  pVar3 = pthread_self();
  if (pVar3 == 0) {
    iVar9 = 0x1060f0;
    iVar2 = 0x109040;
    poVar7 = (ostream *)&std::cout;
    std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
        x24);
  }
  else {
    iVar2 = 0x109040;
    iVar9 = (int)pVar3;
    poVar7 = std::ostream::_M_insert<>(0x109040);
  }
  plVar1 = *(long **)(poVar7 + *(long *)(*(long *)poVar7 + -0x18) + 0xf0);
  if (plVar1 != (long *)0x0) {
    if ((char)plVar1[7] == '\0') {
      std::ctype<char>::_M_widen_init();
      if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
        (**(code **)(*plVar1 + 0x30))(plVar1,10);
      }
    }
    std::ostream::put((char)poVar7);
    std::ostream::flush();
    local_58.tv_sec = 1;
    local_58.tv_nsec = 0;
    do {
      ptVar10 = &local_58;
      iVar2 = nanosleep(&local_58,&local_58);
      iVar9 = (int)ptVar10;
      if (iVar2 != -1) break;
      piVar6 = __errno_location();
    } while (*piVar6 == 4);
    lVar8 = (long)param_1;
    __mutex = (pthread_mutex_t *)(mutexes + lVar8 * 0x28);
    local_58.tv_sec = (__time_t)__mutex;
    iVar2 = pthread_mutex_lock(__mutex);
```

```
62      lVar5 = funcD_params;
63      if (iVar2 == 0) {
64        local_58.tv_nsec = CONCAT71(local_58.tv_nsec._1_7_,1);
65        if (funcD_params_index_pool._48_8_ == funcD_params_index_pool._16_8_) {
66          local_7c = (int)((long)DAT_00109188 - funcD_params >> 4);
67          local_5c = local_7c;
68          if (DAT_00109188 == DAT_00109190) {
69                      /* try { // try from 00104379 to 0010437d has its CatchHandler @ 001043
                           bb */
70            std::vector<>::_M_realloc_insert<>((vector<> *)&funcD_params);
71            lVar5 = funcD_params;
72          }
73          else {
74            *DAT_00109188 = 0;
75            *(undefined4 *)(DAT_00109188 + 1) = 0;
76            *(undefined1 *)((long)DAT_00109188 + 0xc) = 0;
77            DAT_00109188 = DAT_00109188 + 2;
78          }
79        }
80        else {
81          local_7c = *(int *)funcD_params_index_pool._16_8_;
82          local_5c = local_7c;
83          if (funcD_params_index_pool._16_8_ == funcD_params_index_pool._32_8_ + -4) {
84            operator.delete((void *)funcD_params_index_pool._24_8_,0x200);
85            funcD_params_index_pool._16_8_ = *(long *)(funcD_params_index_pool._40_8_ + 8);
86            funcD_params_index_pool._40_8_ = funcD_params_index_pool._40_8_ + 8;
87            funcD_params_index_pool._32_8_ = funcD_params_index_pool._16_8_ + 0x200;
88            lVar5 = funcD_params;
89            funcD_params_index_pool._24_8_ = funcD_params_index_pool._16_8_;
90          }
91          else {
92            funcD_params_index_pool._16_8_ = funcD_params_index_pool._16_8_ + 4;
93          }
94        }
95        puVar4 = (undefined8 *)(lVar5 + (long)local_7c * 0x10);
96        *puVar4 = 0x500000004;
97        *(undefined4 *)(puVar4 + 1) = 0;
98        *(undefined1 *)((long)puVar4 + 0xc) = 0;
99        pthread_mutex_unlock(__mutex);
100       pushToThread(3,0x14,local_7c);
101       while (lVar5 = funcD_params + (long)local_7c * 0x10, *(char *)(lVar5 + 0xc) == '\0') {
102         do {
103         } while (*(long *)(&DAT_0010bb50 + lVar8 * 0x50) == *(long *)(&DAT_0010bb30 + lVar8
                * 0x50))
104         ;
105         execute(param_1);
106       }
107       iVar9 = *(int *)(lVar5 + 8);
108       std::__ostream_insert<>((ostream *)&std::cout,"[funcC] End on thread ",0x16);
109       poVar7 = (ostream *)std::ostream::operator<<((ostream *)&std::cout,iVar9);
110       std::__ostream_insert<>(poVar7," ",1);
111       if (pVar3 == 0) {
112         iVar9 = 0x1060f0;
113         poVar11 = poVar7;
114         std::__ostream_insert<>(poVar7,"thread::id of a non-executing thread",0x24);
115         iVar2 = (int)poVar11;
116       }
```

```
117       else {
118         poVar11 = poVar7;
119         iVar9 = (int)pVar3;
120         poVar7 = std::ostream::_M_insert<>((ulong)poVar7);
121         iVar2 = (int)poVar11;
122       }
123       plVar1 = *(long **)(poVar7 + *(long *)(*(long *)poVar7 + -0x18) + 0xf0);
124       if (plVar1 != (long *)0x0) {
125         if ((char)plVar1[7] == '\0') {
126           std::ctype<char>::_M_widen_init();
127           if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
128             (**(code **)(*plVar1 + 0x30))(plVar1,10);
129           }
130         }
131         std::ostream::put((char)poVar7);
132         std::ostream::flush();
133         if (funcD_params_index_pool._48_8_ == funcD_params_index_pool._64_8_ + -4) {
134           std::deque<>::_M_push_back_aux<int_const&>((deque<> *)funcD_params_index_pool,&
                local_5c);
135         }
136         else {
137           *(int *)funcD_params_index_pool._48_8_ = local_7c;
138           funcD_params_index_pool._48_8_ = funcD_params_index_pool._48_8_ + 4;
139         }
140         LOCK();
141         piVar6 = (int *)(vec + lVar8 * 4);
142         *piVar6 = *piVar6 + -0x14;
143         UNLOCK();
144         if (lVar12 == *(long *)(in_FS_OFFSET + 0x28)) {
145           return;
146         }
147         goto LAB_0010438a;
148       }
149     }
150     else {
151       if (lVar12 != *(long *)(in_FS_OFFSET + 0x28)) goto LAB_0010438a;
152       std::__throw_system_error(iVar2);
153     }
154   }
155   if (lVar12 == *(long *)(in_FS_OFFSET + 0x28)) {
156     std::__throw_bad_cast();
157                     /* catch() { ... } // from try @ 00104379 with catch @ 001043bb */
158     funcC(iVar2,iVar9,lVar12);
159     return;
160   }
161 LAB_0010438a:
162                     /* WARNING: Subroutine does not return */
163   __stack_chk_fail();
164 }
```

funcD(int param_1,int param_2,long param_3)

_2_3)

Listing 40: Decompiled code of funcD(int param_1

```
/* funcD(int, int) [clone .cold] */

void funcD(int param_1,int param_2,long param_3)

{
  long in_FS_OFFSET;

  std::unique_lock<std::mutex>::unlock();
  if (param_3 == *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    _Unwind_Resume();
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

funcD(int param_1,int param_2)

_2)

Listing 41: Decompiled code of funcD(int param_1

```
/* WARNING: Restarted to delay deadcode elimination for space: ram */
/* funcD(int, int) */

void funcD(int param_1,int param_2)

{
  pthread_mutex_t *__mutex;
  long *plVar1;
  int iVar2;
  pthread_t pVar3;
  undefined8 *puVar4;
  long lVar5;
  int *piVar6;
  ostream *poVar7;
  long lVar8;
  int iVar9;
  timespec *ptVar10;
  ostream *poVar11;
  long in_FS_OFFSET;
  int local_70;
  int local_5c;
  timespec local_58;
  long lVar12;

  lVar12 = *(long *)(in_FS_OFFSET + 0x28);
  std::__ostream_insert<>((ostream *)&std::cout,"[funcD] Start on thread ",0x18);
  pVar3 = pthread_self();
  if (pVar3 == 0) {
    iVar9 = 0x1060f0;
    iVar2 = 0x109040;
    poVar7 = (ostream *)&std::cout;
    std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
        x24);
```

```
33    }
34    else {
35      iVar2 = 0x109040;
36      iVar9 = (int)pVar3;
37      poVar7 = std::ostream::_M_insert<>(0x109040);
38    }
39    plVar1 = *(long **)(poVar7 + *(long *)(*(long *)poVar7 + -0x18) + 0xf0);
40    if (plVar1 != (long *)0x0) {
41      if ((char)plVar1[7] == '\0') {
42        std::ctype<char>::_M_widen_init();
43        if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
44          (**(code **)(*plVar1 + 0x30))(plVar1,10);
45        }
46      }
47      std::ostream::put((char)poVar7);
48      std::ostream::flush();
49      local_58.tv_sec = 1;
50      local_58.tv_nsec = 0;
51      do {
52        ptVar10 = &local_58;
53        iVar2 = nanosleep(&local_58,&local_58);
54        iVar9 = (int)ptVar10;
55        if (iVar2 != -1) break;
56        piVar6 = __errno_location();
57      } while (*piVar6 == 4);
58      lVar8 = (long)param_1;
59      __mutex = (pthread_mutex_t *)(mutexes + lVar8 * 0x28);
60      local_58.tv_sec = (__time_t)__mutex;
61      iVar2 = pthread_mutex_lock(__mutex);
62      lVar5 = funcE_params;
63      if (iVar2 == 0) {
64        local_58.tv_nsec = CONCAT71(local_58.tv_nsec._1_7_,1);
65        if (funcE_params_index_pool._16_8_ == funcE_params_index_pool._48_8_) {
66          local_70 = (int)((long)DAT_00109168 - funcE_params >> 4);
67          local_5c = local_70;
68          if (DAT_00109168 == DAT_00109170) {
69                        /* try { // try from 00104a11 to 00104a15 has its CatchHandler @ 00104
                              a53 */
70            std::vector<>::_M_realloc_insert<>((vector<> *)&funcE_params);
71            lVar5 = funcE_params;
72          }
73          else {
74            *DAT_00109168 = 0;
75            *(undefined4 *)(DAT_00109168 + 1) = 0;
76            *(undefined1 *)((long)DAT_00109168 + 0xc) = 0;
77            DAT_00109168 = DAT_00109168 + 2;
78          }
79        }
80        else {
81          local_70 = *(int *)funcE_params_index_pool._16_8_;
82          local_5c = local_70;
83          if (funcE_params_index_pool._16_8_ == funcE_params_index_pool._32_8_ + -4) {
84            operator.delete((void *)funcE_params_index_pool._24_8_,0x200);
85            funcE_params_index_pool._16_8_ = *(long *)(funcE_params_index_pool._40_8_ + 8);
86            funcE_params_index_pool._40_8_ = funcE_params_index_pool._40_8_ + 8;
87            funcE_params_index_pool._32_8_ = funcE_params_index_pool._16_8_ + 0x200;
88            lVar5 = funcE_params;
```

```
89            funcE_params_index_pool._24_8_ = funcE_params_index_pool._16_8_;
90          }
91          else {
92            funcE_params_index_pool._16_8_ = funcE_params_index_pool._16_8_ + 4;
93          }
94        }
95        puVar4 = (undefined8 *)(lVar5 + (long)local_70 * 0x10);
96        *puVar4 = 0x300000002;
97        *(undefined4 *)(puVar4 + 1) = 0;
98        *(undefined1 *)((long)puVar4 + 0xc) = 0;
99        pthread_mutex_unlock(__mutex);
100       pushToThread(4,0x14,local_70);
101       while (lVar5 = funcE_params + (long)local_70 * 0x10, *(char *)(lVar5 + 0xc) == '\0') {
102         do {
103         } while (*(long *)(&DAT_0010bb50 + lVar8 * 0x50) == *(long *)(&DAT_0010bb30 + lVar8
              * 0x50))
104         ;
105         execute(param_1);
106       }
107       iVar9 = *(int *)(lVar5 + 8);
108       piVar6 = (int *)(funcD_params + (long)param_2 * 0x10);
109       piVar6[2] = (iVar9 - *piVar6) + piVar6[1];
110       std::__ostream_insert<>((ostream *)&std::cout,"[funcD] End on thread ",0x16);
111       poVar7 = (ostream *)std::ostream::operator<<((ostream *)&std::cout,iVar9);
112       std::__ostream_insert<>(poVar7," ",1);
113       if (pVar3 == 0) {
114         iVar9 = 0x1060f0;
115         poVar11 = poVar7;
116         std::__ostream_insert<>(poVar7,"thread::id of a non-executing thread",0x24);
117         iVar2 = (int)poVar11;
118       }
119       else {
120         poVar11 = poVar7;
121         iVar9 = (int)pVar3;
122         poVar7 = std::ostream::_M_insert<>((ulong)poVar7);
123         iVar2 = (int)poVar11;
124       }
125       plVar1 = *(long **)(poVar7 + *(long *)(*(long *)poVar7 + -0x18) + 0xf0);
126       if (plVar1 != (long *)0x0) {
127         if ((char)plVar1[7] == '\0') {
128           std::ctype<char>::_M_widen_init();
129           if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
130             (**(code **)(*plVar1 + 0x30))(plVar1,10);
131           }
132         }
133         std::ostream::put((char)poVar7);
134         std::ostream::flush();
135         if (funcE_params_index_pool._48_8_ == funcE_params_index_pool._64_8_ + -4) {
136           std::deque<>::_M_push_back_aux<int_const&>((deque<> *)funcE_params_index_pool,&
                local_5c);
137         }
138         else {
139           *(int *)funcE_params_index_pool._48_8_ = local_70;
140           funcE_params_index_pool._48_8_ = funcE_params_index_pool._48_8_ + 4;
141         }
142         *(undefined1 *)(funcD_params + 0xc + (long)param_2 * 0x10) = 1;
143         LOCK();
```

```
144           piVar6 = (int *)(vec + lVar8 * 4);
145           *piVar6 = *piVar6 + -0x14;
146           UNLOCK();
147           if (lVar12 == *(long *)(in_FS_OFFSET + 0x28)) {
148             return;
149           }
150           goto LAB_00104a22;
151         }
152       }
153       else {
154         if (lVar12 != *(long *)(in_FS_OFFSET + 0x28)) goto LAB_00104a22;
155         std::__throw_system_error(iVar2);
156       }
157     }
158     if (lVar12 == *(long *)(in_FS_OFFSET + 0x28)) {
159       std::__throw_bad_cast();
160                     /* catch() { ... } // from try @ 00104a11 with catch @ 00104a53 */
161       funcD(iVar2,iVar9,lVar12);
162       return;
163     }
164 LAB_00104a22:
165                     /* WARNING: Subroutine does not return */
166     __stack_chk_fail();
167 }
```

funcE(int param_1,int param_2)

_2)

Listing 42: Decompiled code of funcE(int param_1

```
1  /* funcE(int, int) */
2
3  void funcE(int param_1,int param_2)
4
5  {
6    long *plVar1;
7    pthread_t pVar2;
8    int *piVar3;
9    ostream *poVar4;
10
11   poVar4 = (ostream *)&std::cout;
12   std::__ostream_insert<>((ostream *)&std::cout,"[funcE] Start on thread ",0x18);
13   pVar2 = pthread_self();
14   if (pVar2 == 0) {
15     std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
         x24);
16   }
17   else {
18     poVar4 = std::ostream::_M_insert<>(0x109040);
19   }
20   plVar1 = *(long **)(poVar4 + *(long *)(*(long *)poVar4 + -0x18) + 0xf0);
21   if (plVar1 == (long *)0x0) {
22     std::__throw_bad_cast();
23     std::_Deque_base<>::~_Deque_base((_Deque_base<> *)&DAT_0010bb70);
```

```
    std::_Deque_base<>::~_Deque_base((_Deque_base<> *)&queues);
    return;
  }
  if ((char)plVar1[7] == '\0') {
    std::ctype<char>::_M_widen_init();
    if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
      (**(code **)(*plVar1 + 0x30))(plVar1,10);
    }
  }
  std::ostream::put((char)poVar4);
  std::ostream::flush();
  piVar3 = (int *)((long)param_2 * 0x10 + funcE_params);
  *(undefined1 *)(piVar3 + 3) = 1;
  piVar3[2] = (piVar3[1] + *piVar3) * 2;
  LOCK();
  piVar3 = (int *)(vec + (long)param_1 * 4);
  *piVar3 = *piVar3 + -0x14;
  UNLOCK();
  return;
}
```

getBalancedRandomIndex(void)

Listing 43: Decompiled code of getBalancedRandomIndex(void)

```
/* WARNING: Unknown calling convention -- yet parameter storage is locked */
/* getBalancedRandomIndex() [clone .cold] */

void getBalancedRandomIndex(void)

{
  code *pcVar1;

                    /* WARNING: Does not return */
  pcVar1 = (code *)invalidInstructionException();
  (*pcVar1)();
}
```

getBalancedRandomIndex

Listing 44: Decompiled code of getBalancedRandomIndex

```
/* WARNING: Unknown calling convention */
/* getBalancedRandomIndex() */

int getBalancedRandomIndex
              (undefined8 param_1,undefined4 param_2,undefined8 param_3,undefined8 param_4,
              undefined8 param_5,undefined8 param_6)

{
  int iVar1;
  code *pcVar2;
```

```
11    undefined1 auVar3 [16];
12    int *piVar4;
13    int iVar5;
14    int *__src;
15    int *piVar6;
16    int iVar7;
17    long lVar8;
18    int *piVar9;
19    int *piVar10;
20    ulong uVar11;
21    long in_FS_OFFSET;
22    ulong local_b0;
23    undefined4 uStack_9c;
24    int *local_98;
25    int *local_90;
26    int *local_88;
27    int *local_78;
28    int *piStack_70;
29    int *local_68;
30    undefined1 local_58 [8];
31    int *piStack_50;
32    int *local_48;
33    long local_40;
34
35    lVar8 = 0;
36    local_40 = *(long *)(in_FS_OFFSET + 0x28);
37    __src = (int *)operator.new(8);
38    piVar9 = (int *)0x0;
39    local_90 = __src + 2;
40    __src[0] = 0;
41    __src[1] = 0;
42    iVar5 = *vec;
43    *__src = iVar5;
44    iVar1 = vec[1];
45    auVar3._12_4_ = 0;
46    auVar3._0_12_ = stack0xfffffffffffffac;
47    _local_58 = auVar3 << 0x20;
48    local_68 = (int *)0x0;
49    __src[1] = iVar1;
50    local_78 = (int *)0x0;
51    piStack_70 = (int *)0x0;
52    local_98 = __src;
53    local_88 = local_90;
54    do {
55      iVar7 = (int)lVar8;
56      if (((double)iVar1 + (double)iVar5 + 0.0) * 0.5 * 0.8 < (double)__src[lVar8]) {
57  LAB_001033f0:
58        local_58._0_4_ = iVar7 + 1;
59        piVar6 = local_78;
60      }
61      else {
62        if (piVar9 == local_68) {
63                      /* try { // try from 001034c5 to 0010369b has its CatchHandler @ 001036
                           e0 */
64          std::vector<>::_M_realloc_insert<int_const&>((vector<> *)&local_78,piVar9,local_58);
65          piVar9 = piStack_70;
66          goto LAB_001033f0;
```

```
67        }
68        *piVar9 = iVar7;
69        piVar9 = piVar9 + 1;
70        local_58._0_4_ = iVar7 + 1;
71        piVar6 = local_78;
72        piStack_70 = piVar9;
73      }
74      local_78 = piVar6;
75      if (lVar8 != 0) break;
76      lVar8 = 1;
77    } while( true );
78    if (piVar6 != piVar9) goto LAB_00103448;
79    local_48 = (int *)0x0;
80    _local_58 = (undefined1  [16])0x0;
81    local_b0 = (long)local_90 - (long)__src;
82    uVar11 = (long)local_b0 >> 2;
83    if (uVar11 == 0) {
84                      /* WARNING: Does not return */
85      pcVar2 = (code *)invalidInstructionException();
86      (*pcVar2)();
87    }
88    if (local_b0 < 0x7ffffffffffffffd) {
89      piVar6 = (int *)operator.new(local_b0);
90      piVar9 = (int *)((long)piVar6 + local_b0);
91      piStack_50 = piVar6;
92      local_58 = (undefined1  [8])piVar6;
93      local_48 = piVar9;
94      if (local_b0 == 4) {
95        piStack_50 = piVar9;
96        *piVar6 = *__src;
97        std::__introsort_loop<>(piVar6,piVar9,0);
98        local_b0 = 4;
99  LAB_00103630:
100       std::__insertion_sort<>(piVar6,piVar9);
101     }
102     else {
103       memcpy(piVar6,__src,local_b0);
104       lVar8 = 0x3f;
105       if (uVar11 != 0) {
106         for (; uVar11 >> lVar8 == 0; lVar8 = lVar8 + -1) {
107         }
108       }
109       piStack_50 = piVar9;
110       std::__introsort_loop<>(piVar6,piVar9,(long)(int)lVar8 * 2);
111       if ((long)local_b0 < 0x41) goto LAB_00103630;
112       piVar10 = piVar6 + 0x10;
113       std::__insertion_sort<>(piVar6,piVar10);
114       for (; piVar9 != piVar10; piVar10 = piVar10 + 1) {
115         iVar5 = *piVar10;
116         iVar1 = piVar10[-1];
117         piVar4 = piVar10;
118         while (iVar5 < iVar1) {
119           *piVar4 = iVar1;
120           iVar1 = piVar4[-2];
121           piVar4 = piVar4 + -1;
122         }
123         *piVar4 = iVar5;
```

```
124         }
125       }
126       uStack_9c = 0;
127       iVar5 = piVar6[1];
128       if (*__src <= iVar5) {
129         if (local_68 == piStack_70) {
130           std::vector<>::_M_realloc_insert<int_const&>((vector<> *)&local_78,piStack_70,&
                uStack_9c);
131         }
132         else {
133           *piStack_70 = 0;
134           piStack_70 = piStack_70 + 1;
135         }
136       }
137       uStack_9c = 1;
138       if (__src[1] <= iVar5) {
139         if (piStack_70 == local_68) goto LAB_0010369c;
140         *piStack_70 = 1;
141         piStack_70 = piStack_70 + 1;
142       }
143     }
144     else {
145       if (local_40 != *(long *)(in_FS_OFFSET + 0x28)) {
146                       /* WARNING: Subroutine does not return */
147         __stack_chk_fail();
148       }
149       std::__throw_bad_array_new_length();
150   LAB_0010369c:
151                       /* try { // try from 001036a9 to 001036c4 has its CatchHandler @ 001036
                            d4 */
152       std::vector<>::_M_realloc_insert<int_const&>((vector<> *)&local_78);
153     }
154     operator.delete(piVar6,local_b0);
155     piVar9 = piStack_70;
156   LAB_00103448:
157     piVar6 = local_78;
158     iVar5 = std::uniform_int_distribution<int>::operator()
159                       ((uniform_int_distribution<int> *)rng,(mersenne_twister_engine *)0x0,
160                        (param_type *)(ulong)((int)((long)piVar9 - (long)local_78 >> 2) - 1));
161     iVar5 = piVar6[iVar5];
162     if (piVar6 != (int *)0x0) {
163       operator.delete(piVar6,(long)local_68 - (long)piVar6);
164     }
165     operator.delete(__src,(long)local_88 - (long)__src);
166     if (local_40 != *(long *)(in_FS_OFFSET + 0x28)) {
167                       /* WARNING: Subroutine does not return */
168       __stack_chk_fail();
169     }
170     return iVar5;
171   }
```

initializeArray(void)

Listing 45: Decompiled code of initializeArray(void)

```
/* WARNING: Unknown calling convention -- yet parameter storage is locked */
/* initializeArray() */

void initializeArray(void)

{
  vec = (undefined4 *)operator.new[](8);
  LOCK();
  *vec = 0;
  UNLOCK();
  LOCK();
  vec[1] = 0;
  UNLOCK();
  return;
}
```

main(void)

Listing 46: Decompiled code of main(void)

```
undefined8 main(void)

{
  int iVar1;
  long lVar2;
  long in_FS_OFFSET;
  undefined8 local_70;
  long *local_68;
  char local_60;
  undefined1 local_58 [16];
  long local_40;

  local_40 = *(long *)(in_FS_OFFSET + 0x28);
  local_58 = (undefined1  [16])0x0;
                    /* try { // try from 0010275b to 00102797 has its CatchHandler @
                       00102907 */
  initializeArray();
  lVar2 = 0;
  while( true ) {
    local_70 = 0;
    local_68 = (long *)operator.new(0x18);
    *local_68 = (long)&PTR_~_State_impl_00108c40;
    *(int *)(local_68 + 1) = (int)lVar2;
    local_68[2] = (long)threadFunction;
                    /* try { // try from 001027b2 to 001027b6 has its CatchHandler @ 0010291
                       f */
    std::thread::_M_start_thread(&local_70,&local_68,std::thread::_M_thread_deps_never_run);
    if (local_68 != (long *)0x0) {
      (**(code **)(*local_68 + 8))();
    }
    if (*(long *)(local_58 + lVar2 * 8) != 0) goto LAB_001028cf;
    *(undefined8 *)(local_58 + lVar2 * 8) = local_70;
    if (lVar2 != 0) break;
    lVar2 = 1;
  }
```

```
34                    /* try { // try from 001027f2 to 001027f6 has its CatchHandler @
                         00102907 */
35    pushToThread(0,200,-1);
36    local_60 = '\0';
37    local_68 = (long *)g_allTasksDoneMtx;
38    iVar1 = pthread_mutex_lock((pthread_mutex_t *)g_allTasksDoneMtx);
39    if (iVar1 == 0) {
40      local_60 = '\x01';
41      while (g_inFlightTasks != 0) {
42                    /* try { // try from 0010282e to 0010288c has its CatchHandler @
                         00102913 */
43        std::condition_variable::wait((unique_lock *)g_allTasksDoneCV);
44      }
45      iVar1 = pthread_mutex_lock((pthread_mutex_t *)stopMutex);
46      if (iVar1 == 0) {
47        stopThreads = 1;
48        pthread_mutex_unlock((pthread_mutex_t *)stopMutex);
49        std::condition_variable::notify_all();
50        std::thread::join();
51        std::condition_variable::notify_all();
52        std::thread::join();
53        if (local_60 != '\0') {
54          std::unique_lock<std::mutex>::unlock();
55        }
56        if ((local_58._8_8_ != 0) || (local_58._0_8_ != 0)) {
57  LAB_001028cf:
58                    /* WARNING: Subroutine does not return */
59          std::terminate();
60        }
61        if (local_40 == *(long *)(in_FS_OFFSET + 0x28)) {
62          return 0;
63        }
64        goto LAB_00102902;
65      }
66    }
67    else {
68      if (local_40 != *(long *)(in_FS_OFFSET + 0x28)) goto LAB_00102902;
69                    /* try { // try from 001028e6 to 001028ea has its CatchHandler @
                         00102907 */
70      iVar1 = std::__throw_system_error(iVar1);
71    }
72    if (local_40 == *(long *)(in_FS_OFFSET + 0x28)) {
73                    /* try { // try from 001028fd to 00102901 has its CatchHandler @
                         00102913 */
74      std::__throw_system_error(iVar1);
75    }
76  LAB_00102902:
77                    /* WARNING: Subroutine does not return */
78    __stack_chk_fail();
79  }
```

main.cold(void)

Listing 47: Decompiled code of main.cold(void)

```
void main.cold(void)

{
  long in_FS_OFFSET;
  long *param_1;
  long param_2;
  long param_3;
  long param_4;

  if (param_1 != (long *)0x0) {
    (**(code **)(*param_1 + 8))();
  }
  if ((param_3 == 0) && (param_2 == 0)) {
    if (param_4 == *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
      _Unwind_Resume();
    }
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
                    /* WARNING: Subroutine does not return */
  std::terminate();
}
```

pushToThread(int param_1,int param_2,int param_3)

_2_3)

Listing 48: Decompiled code of pushToThread(int param_1

```
/* pushToThread(int, int, int) [clone .cold] */

void pushToThread(int param_1,int param_2,int param_3)

{
  pthread_mutex_t *unaff_R15;

  pthread_mutex_unlock(unaff_R15);
                    /* WARNING: Subroutine does not return */
  _Unwind_Resume();
}
```

pushToThread(int param_1,int param_2,int param_3)

_2_3)

Listing 49: Decompiled code of pushToThread(int param_1

```
/* pushToThread(int, int, int) */

void pushToThread(int param_1,int param_2,int param_3)

{
  int *piVar1;
```

```
  long lVar2;
  int iVar3;
  int *piVar4;
  void *pvVar5;
  long lVar6;
  long *__dest;
  size_t sVar7;
  undefined8 unaff_RBX;
  long lVar8;
  undefined8 unaff_RBP;
  long lVar9;
  long *__src;
  ulong uVar10;
  undefined8 unaff_R12;
  undefined8 unaff_R15;
  undefined1 uVar11;
  undefined4 in_stack_ffffffffffffff98;
  ulong local_60;
  undefined8 in_stack_ffffffffffffffa8;
  void *local_48;

  iVar3 = getBalancedRandomIndex
                    (CONCAT44(param_2,in_stack_ffffffffffffff98),
                     (int)((ulong)in_stack_ffffffffffffffa8 >> 0x20),unaff_RBX,unaff_RBP,
                         unaff_R12,
                     unaff_R15);
  lVar8 = (long)iVar3;
  iVar3 = pthread_mutex_lock((pthread_mutex_t *)(mutexes + lVar8 * 0x28));
  if (iVar3 != 0) {
LAB_00103a6b:
    std::__throw_system_error(iVar3);
LAB_00103a72:
                    /* WARNING: Subroutine does not return */
                    /* try { // try from 00103a79 to 00103a7d has its CatchHandler @ 00103
                         a7e */
    std::__throw_length_error("cannot create std::deque larger than max_size()");
  }
  lVar9 = lVar8 * 0x50;
  piVar1 = *(int **)(&DAT_0010bb50 + lVar9);
  if (piVar1 != (int *)(*(long *)(&DAT_0010bb60 + lVar9) + -8)) {
    *piVar1 = param_1;
    piVar4 = piVar1 + 2;
    piVar1[1] = param_3;
    goto LAB_0010375d;
  }
  local_60 = *(ulong *)(&DAT_0010bb68 + lVar9);
  __src = *(long **)(&DAT_0010bb48 + lVar9);
  lVar6 = local_60 - (long)__src;
  if (((long)piVar1 - *(long *)(&DAT_0010bb58 + lVar9) >> 3) +
      ((lVar6 >> 3) + -1 + (ulong)(local_60 == 0)) * 0x40 +
      (*(long *)(&DAT_0010bb40 + lVar9) - *(long *)(&DAT_0010bb30 + lVar9) >> 3) ==
      0xffffffffffffffff) goto LAB_00103a72;
  uVar10 = (&DAT_0010bb28)[lVar8 * 10];
  if (uVar10 - ((long)(local_60 - (&queues)[lVar8 * 10]) >> 3) < 2) {
    __dest = (long *)((lVar6 >> 3) + 2);
    if ((ulong)((long)__dest * 2) < uVar10) {
      __dest = (long *)((&queues)[lVar8 * 10] + (uVar10 - (long)__dest >> 1) * 8);
```

```
62        sVar7 = (local_60 + 8) - (long)__src;
63        if (__dest < __src) {
64          if ((long)sVar7 < 9) {
65            if (sVar7 == 8) {
66              *__dest = *__src;
67            }
68          }
69          else {
70            __dest = (long *)memmove(__dest,__src,sVar7);
71          }
72        }
73        else if ((long)sVar7 < 9) {
74          if (sVar7 == 8) {
75            *(long *)((long)__dest + lVar6) = *__src;
76          }
77        }
78        else {
79          memmove((void *)((long)__dest + ((lVar6 + 8) - sVar7)),__src,sVar7);
80        }
81      }
82      else {
83        if (uVar10 == 0) {
84          local_60 = 3;
85          uVar10 = 0x18;
86  LAB_001038b9:
87          local_48 = operator.new(uVar10);
88          __dest = (long *)((long)local_48 + (local_60 - (long)__dest >> 1) * 8);
89          __src = *(long **)(&DAT_0010bb48 + lVar8 * 0x50);
90          sVar7 = (*(long *)(&DAT_0010bb68 + lVar8 * 0x50) + 8) - (long)__src;
91          uVar11 = sVar7 == 8;
92          if ((long)sVar7 < 9) goto LAB_00103a32;
93          __dest = (long *)memmove(__dest,__src,sVar7);
94        }
95        else {
96          local_60 = (uVar10 + 1) * 2;
97          if (local_60 < 0x1000000000000000) {
98            uVar10 = (uVar10 + 1) * 0x10;
99            goto LAB_001038b9;
100         }
101         uVar11 = local_60 >> 0x3d == 0;
102         if ((bool)uVar11) {
103           iVar3 = std::__throw_bad_alloc();
104           goto LAB_00103a6b;
105         }
106         std::__throw_bad_array_new_length();
107 LAB_00103a32:
108         if ((bool)uVar11) {
109           *__dest = *__src;
110         }
111       }
112       operator.delete((void *)(&queues)[lVar8 * 10],(&DAT_0010bb28)[lVar8 * 10] << 3);
113       (&queues)[lVar8 * 10] = local_48;
114       (&DAT_0010bb28)[lVar8 * 10] = local_60;
115     }
116     lVar2 = *__dest;
117     *(long **)(&DAT_0010bb48 + lVar9) = __dest;
118     *(long *)(&DAT_0010bb38 + lVar9) = lVar2;
```

```
119       *(long *)(&DAT_0010bb40 + lVar9) = lVar2 + 0x200;
120       lVar2 = *(long *)((long)__dest + lVar6);
121       *(long **)(&DAT_0010bb68 + lVar9) = (long *)((long)__dest + lVar6);
122       *(long *)(&DAT_0010bb58 + lVar9) = lVar2;
123       *(long *)(&DAT_0010bb60 + lVar9) = lVar2 + 0x200;
124       local_60 = *(ulong *)(&DAT_0010bb68 + lVar8 * 0x50);
125     }
126                       /* try { // try from 0010382d to 00103a6a has its CatchHandler @ 00103
                              a7e */
127     pvVar5 = operator.new(0x200);
128     *(void **)(local_60 + 8) = pvVar5;
129     piVar1 = *(int **)(&DAT_0010bb50 + lVar8 * 0x50);
130     lVar6 = *(long *)(&DAT_0010bb68 + lVar8 * 0x50);
131     *piVar1 = param_1;
132     piVar1[1] = param_3;
133     piVar4 = *(int **)(lVar6 + 8);
134     *(long *)(&DAT_0010bb68 + lVar9) = lVar6 + 8;
135     *(int **)(&DAT_0010bb58 + lVar9) = piVar4;
136     *(int **)(&DAT_0010bb60 + lVar9) = piVar4 + 0x80;
137   LAB_0010375d:
138     *(int **)(&DAT_0010bb50 + lVar8 * 0x50) = piVar4;
139     LOCK();
140     piVar1 = (int *)(vec + lVar8 * 4);
141     *piVar1 = *piVar1 + param_2;
142     UNLOCK();
143     LOCK();
144     g_inFlightTasks = g_inFlightTasks + 1;
145     UNLOCK();
146     pthread_mutex_unlock((pthread_mutex_t *)(mutexes + lVar8 * 0x28));
147     std::condition_variable::notify_one();
148     return;
149   }
```

register_tm_clones(void)

Listing 50: Decompiled code of register_tm_clones(void)

```
1  /* WARNING: Removing unreachable block (ram,0x00102c64) */
2  /* WARNING: Removing unreachable block (ram,0x00102c70) */
3
4  void register_tm_clones(void)
5
6  {
7    return;
8  }
```

taskFinished(void)

Listing 51: Decompiled code of taskFinished(void)

```
1  /* WARNING: Unknown calling convention -- yet parameter storage is locked */
2  /* taskFinished() */
```

```
3
4  void taskFinished(void)
5
6  {
7    long *plVar1;
8    int iVar2;
9    pthread_t pVar3;
10   int *piVar4;
11   ostream *poVar5;
12   int in_ESI;
13
14   LOCK();
15   g_inFlightTasks = g_inFlightTasks + -1;
16   UNLOCK();
17   if (g_inFlightTasks != 0) {
18     return;
19   }
20   iVar2 = pthread_mutex_lock((pthread_mutex_t *)g_allTasksDoneMtx);
21   if (iVar2 != 0) {
22     std::__throw_system_error(iVar2);
23     poVar5 = (ostream *)&std::cout;
24     std::__ostream_insert<>((ostream *)&std::cout,"[funcE] Start on thread ",0x18);
25     pVar3 = pthread_self();
26     if (pVar3 == 0) {
27       std::__ostream_insert<>((ostream *)&std::cout,"thread::id of a non-executing thread",0
           x24);
28     }
29     else {
30       poVar5 = std::ostream::_M_insert<>(0x109040);
31     }
32     plVar1 = *(long **)(poVar5 + *(long *)(*(long *)poVar5 + -0x18) + 0xf0);
33     if (plVar1 == (long *)0x0) {
34       std::__throw_bad_cast();
35       std::_Deque_base<>::~_Deque_base((_Deque_base<> *)&DAT_0010bb70);
36       std::_Deque_base<>::~_Deque_base((_Deque_base<> *)&queues);
37       return;
38     }
39     if ((char)plVar1[7] == '\0') {
40       std::ctype<char>::_M_widen_init();
41       if (*(code **)(*plVar1 + 0x30) != std::ctype<char>::do_widen) {
42         (**(code **)(*plVar1 + 0x30))(plVar1,10);
43       }
44     }
45     std::ostream::put((char)poVar5);
46     std::ostream::flush();
47     piVar4 = (int *)((long)in_ESI * 0x10 + funcE_params);
48     *(undefined1 *)(piVar4 + 3) = 1;
49     piVar4[2] = (piVar4[1] + *piVar4) * 2;
50     LOCK();
51     piVar4 = (int *)(vec + (long)iVar2 * 4);
52     *piVar4 = *piVar4 + -0x14;
53     UNLOCK();
54     return;
55   }
56   std::condition_variable::notify_all();
57   pthread_mutex_unlock((pthread_mutex_t *)g_allTasksDoneMtx);
58   return;
```

```
}
```

threadFunction(int param_1,char param_2,long param_3)

_2_3)

Listing 52: Decompiled code of threadFunction(int param_1

```
/* threadFunction(int) [clone .cold] */

void threadFunction(int param_1,char param_2,long param_3)

{
  long in_FS_OFFSET;

  if (param_2 != '\0') {
    std::unique_lock<std::mutex>::unlock();
  }
  if (param_3 == *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    _Unwind_Resume();
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

threadFunction(int param_1)

Listing 53: Decompiled code of threadFunction(int param_1)

```
/* threadFunction(int) */

void threadFunction(int param_1)

{
  pthread_mutex_t *__mutex;
  long lVar1;
  int iVar2;
  long lVar3;
  long lVar4;
  long in_FS_OFFSET;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  lVar3 = (long)param_1;
  __mutex = (pthread_mutex_t *)(mutexes + lVar3 * 0x28);
  lVar4 = lVar3 * 0x50;
  do {
    iVar2 = pthread_mutex_lock(__mutex);
    if (iVar2 != 0) {
      if (lVar1 == *(long *)(in_FS_OFFSET + 0x28)) {
        std::__throw_system_error(iVar2);
      }
LAB_00104b64:
```

```
24                      /* WARNING: Subroutine does not return */
25        __stack_chk_fail();
26      }
27      while ((*(long *)(&DAT_0010bb50 + lVar4) == *(long *)(&DAT_0010bb30 + lVar4) &&
28              (stopThreads == '\0'))) {
29                      /* try { // try from 00104b15 to 00104b19 has its CatchHandler @ 00104
                           b69 */
30        std::condition_variable::wait((unique_lock *)(conditions + lVar3 * 0x30));
31      }
32      if (__mutex != (pthread_mutex_t *)0x0) {
33        pthread_mutex_unlock(__mutex);
34      }
35      if ((stopThreads != '\0') &&
36         (*(long *)(&DAT_0010bb50 + lVar4) == *(long *)(&DAT_0010bb30 + lVar4))) {
37        if (lVar1 == *(long *)(in_FS_OFFSET + 0x28)) {
38          return;
39        }
40        goto LAB_00104b64;
41      }
42      execute(param_1);
43    } while( true );
44  }
```