



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A Hybrid Approach for Automatic Recognition of C++ Objects in Optimized Binaries

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING -
INGEGNERIA INFORMATICA

Author: **Samuele Negrini**

Student ID: 968469

Advisor: Prof. Mario Polino

Co-advisor: Prof. Stefano Zanero

Academic Year: 2022-23

Abstract

In this study, we present a tool for the recognition of C++ objects in optimized binaries, where essential information like symbols and function calls is often missing. Existing approaches for recognizing C++ objects have limitations, as they rely on brittle information, do not effectively handle optimized binaries, or focus on specific classes and platforms.

To address this, we propose a hybrid approach composed of a stack access model and two heuristics to recognize objects allocated on the stack. The first heuristic employs the stack access model and leverages robust objects' features to infer the type and location of objects on the stack. In this way, it generates a set of candidate objects for which we build a track that identifies their usage and lifecycle within the binary. The second heuristic improves the accuracy of the object recognition process by detecting inline methods, associated with the candidate objects, through the analysis of the memory access offsets found in the tracks.

Our tool builds upon a two-phase system consisting of fingerprint generation and fingerprint matching. First, it automatically generates and analyzes appropriate binaries to extract relevant features of objects and methods and stores them in fingerprints. Then it utilizes the heuristics, along with the stored fingerprints, to recognize objects in a target binary.

We evaluate our tool on a dataset of 497 GitHub C++ projects compiled with different optimization levels. We achieve promising results, although further work is required to improve the precision, especially for simple classes.

We successfully recognize objects of the `std::string` class, with an F1-score of 62% with -O2 optimization level, which remains similar in -O3 and slightly lower in -Os. For the `std::thread` class, the F1-score reaches 76% in -O2, with small fluctuations in -O3 and -Os. However, for the `std::vector` class, the F1-score drops to 30% in -O2, showing similar results in -O3 and slightly higher results in -Os.

Keywords: reverse engineering, C++, optimized binaries, object recognition, memory access offsets, inline functions.

Abstract in lingua italiana

In questo studio presentiamo uno strumento per il riconoscimento degli oggetti C++ nei binari ottimizzati, dove spesso mancano informazioni essenziali come simboli e chiamate di funzione. Gli approcci esistenti per il riconoscimento degli oggetti C++ hanno delle limitazioni, in quanto si basano su informazioni fragili, non gestiscono efficacemente i binari ottimizzati o si concentrano su classi e piattaforme specifiche.

Per risolvere questo problema, proponiamo un approccio ibrido composto da un modello di accesso allo stack e da due euristiche per riconoscere gli oggetti allocati sullo stack. La prima euristica utilizza il modello di memoria personalizzato e sfrutta le caratteristiche robuste degli oggetti per dedurre il tipo e la posizione degli oggetti sullo stack. In questo modo, genera un insieme di oggetti candidati, per i quali costruiamo una traccia che identifica il loro utilizzo e il loro ciclo di vita all'interno del binario. La seconda euristica migliora l'accuratezza del processo di riconoscimento degli oggetti rilevando i metodi inline, associati agli oggetti candidati, mediante l'analisi degli offset di accessi in memoria trovati nelle tracce.

Il nostro strumento si basa su un sistema a due fasi composto dalla generazione di impronte digitali e dalla loro corrispondenza. In primo luogo, genera e analizza automaticamente binari appropriati per estrarre le caratteristiche rilevanti di oggetti e metodi e memorizzarle nelle impronte digitali. Successivamente, utilizza le euristiche insieme alle impronte digitali memorizzate per riconoscere gli oggetti in un determinato binario.

Valutiamo il nostro strumento su un set di dati composto da 497 progetti C++ di GitHub compilati con diversi livelli di ottimizzazione. Otteniamo risultati promettenti, sebbene siano necessari ulteriori lavori per migliorare la precisione, soprattutto per le classi semplici.

Riconosciamo con successo gli oggetti della classe `std::string`, con un'F1-score del 62% con il livello di ottimizzazione `-O2`, che è rimasta simile in `-O3` e leggermente inferiore in `-Os`. Per la classe `std::thread`, l'F1-score raggiunge il 76% in `-O2`, con piccole fluttuazioni in `-O3` e `-Os`. Tuttavia, per la classe `std::vector`, l'F1-score scende al 30% in `-O2`, con risultati simili in `-O3` e leggermente superiori in `-Os`.

Parole chiave: reverse engineering, C++, binari ottimizzati, riconoscimento di oggetti, offset di accesso in memoria, funzioni inline.

Contents

Abstract	i
Abstract in lingua italiana	iii
Contents	v
1 Introduction	1
2 Background and Motivation	5
2.1 Background	5
2.1.1 Reverse Engineering	5
2.1.2 Object Memory Allocation	5
2.1.3 Inline Functions	6
2.2 Problem Statement	7
2.3 State of the Art	8
2.3.1 Object Recognition	8
2.4 Goals and Challenges	9
3 Approach	11
3.1 Overview	11
3.2 Building Custom Stack Frame	12
3.3 Inferring Object Position	14
3.4 Matching Inline Methods	16
4 Implementation Details	17
4.1 System Architecture	17
4.2 System Details	20
4.2.1 Parser	20
4.2.2 Code Generator	22
4.2.3 Compiler	24

4.2.4	Method Fingerprinter	24
4.2.5	Object Structure Fingerprinter	26
4.2.6	Recognizer	27
5	Experimental Validation	31
5.1	Goals	31
5.2	Dataset	32
5.3	Experimental Setup	33
5.4	Experiments	33
5.4.1	Class <code>std::string</code>	34
5.4.2	Class <code>std::thread</code>	36
5.4.3	Class <code>std::vector</code>	38
6	Limitations	41
6.1	Simple Objects	41
6.2	Methods with Low Offsets Count	41
7	Future Works	43
7.1	Object Recognition in Non-Optimized Binaries	43
7.2	Inline Functions Recognition	43
8	Conclusions	45
	Bibliography	47
	List of Figures	51
	List of Tables	53
	Listings	55
	Ringraziamenti	57

1 | Introduction

Reverse engineering is a field in binary analysis that involves the examination of binary code of a program to extract information about its behavior and structure, especially when the corresponding source code is unavailable. As computer programs and systems become more complex, the need for automatic, reliable and efficient reverse engineering techniques becomes more pressing.

In recent years, the use of C++ has become increasingly widespread in the development of software, making it an important language to consider also in the reverse engineering field. However, C++ introduces additional challenges to reverse engineering due to its support for object-oriented programming (OOP). Objects are the fundamental building blocks of object-oriented programming, encapsulating data and behaviors within a single entity. They allow for the creation of reusable and modular code, facilitating code organization and maintainability. For these reasons, being able to recognize objects within a binary is of fundamental importance and would provide reverse engineers with valuable information and insights into the program's structure and behavior. In fact, if we can recognize their location, we can then better understand the types of data they handle and how they are manipulated i.e., the methods that are called on the objects.

Numerous studies explore the generic area of type inference [5], focusing on diverse aspects such as the identification of primitive types [3], complex data structures and class hierarchies [9]. Some works [10, 21] focus on recovering custom C++ classes but rely on run-time type information (RTTI) which may not always be available. Other works try to identify C++ objects in optimized binaries. ObjDigger [13] and OOAnalyzer [17] employ symbolic execution and inter-procedural data flow analysis to retrieve information about C++ abstractions, but their performance may be affected by the computational complexity of symbolic execution. TIARA [20] uses deep learning to recover container types in COTS C++ binaries. However, it focuses only on specific C++ containers and Windows binaries. None of these state-of-the-art approaches specifically address the recognition of C++ objects in optimized binaries by relying on universally available information and can be easily extended to handle a wider range of classes.

In this study, we address the problem of recognizing C++ objects in optimized or stripped binaries, where relevant information, such as symbols and function calls, is missing or unavailable. These binaries are generated by compilers with aggressive optimization levels to improve the program’s performance. Compiler optimizations such as function inlining, make it more complex to recover functions and function calls that can otherwise be used as a starting point for object recognition. Indeed, while the recognition of functions in binaries has been subject of previous research with some remarkable results [6, 7], the recognition of inline functions has received limited attention until recent times [1, 2, 4]. Therefore, we believe it is necessary to approach the problem from a different perspective.

We propose a static analysis hybrid-approach based on memory access offsets for recognizing C++ objects in optimized binaries. First, we construct a stack access model based on memory accesses performed on the stack. Next, we use a heuristic to infer the type and location of objects on the stack. Then, starting from the object’s base address we follow the usage of the object and build a track that identifies the object’s lifecycle in the binary. Last, we apply a second heuristic to match inline functions that can be found inside the object’s track. This last step allows us to validate the object type and location, and improve the overall accuracy. We implement the tool as a two-phase system consisting of fingerprint generation and fingerprint matching. In the first phase, we automatically generate and analyze appropriate binaries to extract relevant features and store them in fingerprints. In the second phase, we implement the heuristics that use the stored fingerprints to recognize objects in a target binary. This approach allows us to efficiently identify objects in an optimized binary based on the features captured in the stored fingerprints.

We test our tool on a dataset of 497 C++ GitHub projects compiled with different optimization levels targeting three different C++ classes `std::string`, `std::thread` and `std::vector`. We verify each object’s match by comparing it with the ground truth extracted from the DWARF debugging information. Our tool achieves a precision of 62% and a recall of 63% for `std::string` with `-O2` optimization. These values are similar for `-O3` and slightly lower for `-Os`. For `std::thread`, we do not rely on the inline function matching heuristic and achieve a precision of 90% and a recall of 66% in `-O2` with slightly lower and higher results in `-O3` and `-Os`, respectively. The precision drops to 21% while the recall remains at 51% for `std::vector` in `-O2` with similar results in `-O3` and slightly higher precision in `-Os`. This reveals promising results, although further work is needed to improve the precision, especially when considering simple classes.

In summary, we make the following contributions:

- We present a new approach based on a stack access model and two heuristics to infer the position of objects starting from the memory access offsets performed on the stack pointer and detect the associated inline methods.
- We develop a new tool able to automatically fingerprint and match objects and inline functions from C++ classes.
- We propose two novel types of fingerprints: one based on the in-memory structure of an object; and another one based on memory access offsets of functions. Combined, they allow for the recognition of objects in optimized binaries.
- We evaluate the performance of our tool on a dataset of C++ projects compiled with different optimization levels and compare its results against the ground truth extracted from the DWARF debugging information.

2 | Background and Motivation

In this chapter, we present the background information related to the topic of this work. Then, we analyze the main aspects of the problem and its challenges. We also review the latest approaches proposed by state-of-the-art studies and discuss their potential applications in this context.

2.1. Background

In this section, we provide a description of the concepts related to reverse engineering, object memory allocation and inline functions.

2.1.1. Reverse Engineering

Reverse engineering is the process of analyzing the binary code of a program to understand its behavior. This becomes essential when the corresponding source code is unavailable to recover the internal functioning of the program. It is a time-consuming task and requires expertise and advanced technical skills. Reverse engineers utilize tools to assist them in gradually uncovering the program's logic and structure: from the low-level byte sequence to higher-level abstractions like assembly instructions, variables and functions. As software becomes more complex, the task of reverse engineering becomes even more time-consuming. Furthermore, the growing use of common libraries in modern software can result in a significant amount of time being spent on repeatedly analyzing the same libraries. Although complex, it is important to improve tools that can automate repetitive and tedious tasks in the reverse engineer workflow.

2.1.2. Object Memory Allocation

Objects are the fundamental building blocks of object-oriented programming (OOP). An object is created by instantiating a class which is the blueprint that defines the properties (attributes) and behaviors (methods) of the object. We can instantiate multiple objects from the same class, each representing a unique instance of that class. These objects

share the same structure and behavior defined by the class but maintain their own data.

As specified by the C++ International Standard [12], objects in C++ have an associated storage duration. This characteristic specifies the minimum potential lifespan of the storage in which the object is stored. The Standard distinguishes objects based on the storage duration, hence by the construct used to create them. Despite the lifespan of an object is still relevant to us, for the purpose of our work it is more suitable to distinguish objects based on their memory allocation.

Global objects are allocated inside the `data` segment of the binary at compile-time. They are declared outside any functions, hence they have a global scope and can be accessed from anywhere within the program. They are created at program startup and destroyed when the program exits.

Stack-allocated objects are allocated on the stack and may also be referred to as local objects since they are declared inside a function or a block. They are created when the program execution enters the block where they are declared and exist only within the scope of that block. Once the program execution exits the block, they are automatically deallocated.

Heap-allocated objects are created dynamically using the `new` keyword in C++. Memory for these objects is allocated on the heap. These objects have a wider scope than stack-allocated objects and can be accessed from anywhere in the program as long as a valid pointer to the object is available. They must be manually deallocated by the programmer using the `delete` keyword.

2.1.3. Inline Functions

Inline functions are the result of an optimization performed by the compiler called function inlining. An inline function is a function whose code is directly inserted at the point where there would be the traditional function call, thus replacing the function call itself. By doing so, the compiler eliminates the overhead caused by parameter passing and jumping to a different location. This results in improved performance and faster execution times. The compiler employs several heuristics to decide whether to inline a function call or not: for instance, inlining could happen when a function is small and frequently called. Programmers may specify the `inline` keyword while declaring a function to suggest the compiler to inline the given function, but it is just a hint since the compiler can still choose not to inline the function (e.g. if the function is too big).

Classes' methods are often considered to be good candidates for inlining, especially those of the C++ Standard Library as they are very small and frequently called.

2.2. Problem Statement

In the field of reverse engineering, recognizing C++ objects in memory is a relatively simple task when dealing with unoptimized binaries. In these binaries, the presence of function calls simplifies the identification of functions within the code, thus allowing reliable detection of objects on which these functions might be invoked.

On the other hand, the problem becomes significantly more challenging when working with optimized binaries. Binaries compiled with aggressive optimization levels often have a complex control flow, making it difficult to understand the program's logic and posing significant challenges to the task of reverse engineering. Compilers perform a variety of optimizations such as function inlining to reduce the program's execution time and memory usage. As a result, we completely lose the information given by the presence of function calls and the identification of functions in the binary.

As C++ is primarily an object-oriented programming language, understanding the layout and structure of objects is essential for effective reverse engineering. Objects in C++ can have complex arrangements of data members, including inheritance and virtual function tables. They are typically created and destroyed by invoking constructors and destructors, respectively. Constructors initialize the object's data members and set up its internal state while destructors are responsible for cleaning up resources and freeing memory. Identifying those methods could be useful to accurately track object lifetimes and identify the points of object creation and destruction. However, in optimized binaries, the direct calls to constructors and destructors may be eliminated and replaced with an inline function, thus complicating their usage for object recognition.

Moreover, by recognizing an object we can provide reverse engineers with insights about the organization of data within objects and their memory layout. This information could be useful for several analysis tasks, such as data flow analysis, pointer analysis and vulnerability detection. For example, knowing the layout of objects can help identify potential memory corruptions or ensure proper handling of pointers and references to object members.

In conclusion, C++ object recognition in optimized binaries is a complex and challenging problem for reverse engineers. Addressing the aspects discussed above is essential to enable effective reverse engineering.

2.3. State of the Art

This section provides an overview of the current state of research in the field of type inference, with a specific focus on object recognition.

2.3.1. Object Recognition

The recognition of objects and classes falls under a wider topic called type inference which involves the identification of primitive types as well as complex data structures. For many years, type inference has been recognized as a challenging problem in binary analysis and has been the subject of extensive research [5]. Some works [10, 21] focus on recovering custom C++ classes, but they rely on run-time type information (RTTI) which is not always available.

Balakrishnan et al. [3] propose a tool to discover variables in a binary by analyzing memory accesses combined with Value-Set Analysis (VSA) which is a data-flow analysis used to track the values that each variable may assume in each step of the program execution. We find their work particularly relevant as we acknowledge that memory accesses are a feature that may not be eliminated during compiler optimizations. However, their work focuses on recovering primitive type variables while our concern is to recognize objects.

Erinfolami et al. [9] aim to solve a slightly different problem: recovering class hierarchy from optimized C++ binaries. Their inference engine, DeClassifier, is based on code features such as virtual functions that cannot be optimized away by the compiler. Hence they propose novel techniques to identify and analyze constructors, destructors and objects layout. However, their work relies heavily on virtual functions which are not always available.

ObjDigger [13] employs symbolic execution and inter-procedural data flow analysis to retrieve information about object instances, data members and methods within a class. It tracks the usage of the *this* pointer, which refers to the current object instance, throughout the code. This allows ObjDigger to identify and associate the relevant elements belonging to the same class, even in optimized C++ binaries. Nevertheless, as symbolic execution can be complex and computationally expensive, this approach may suffer from poor performance, especially when dealing with large and complex binaries. Its successor, OOAnalyzer [17], also employs symbolic execution and a Prolog-based reasoning system to recover C++ abstractions from executables. Although it recovers information about all classes and methods, it suffers from the same problems related to the symbolic execution of its predecessor, and it is tailored to analyze only Windows executables.

One recent study [20], proposes a type-relevant slicing algorithm and makes use of a Graph Convolutional Network to recover container types in COTS C++ binaries. Results given by this study are really promising, however, it focuses only on specific C++ containers and tests are performed only for Windows binaries.

2.4. Goals and Challenges

This work aims to create a tool capable of detecting potential object instances of C++ classes within an optimized binary application. By achieving this goal we want to ease the process of reverse engineering, especially in the context of object recognition. During the development of this tool, we encountered several challenges.

First, finding features that remain preserved in optimized binaries. Compiler optimizations can alter the binary's structure and behavior, making it difficult to identify stable and consistent characteristics. Addressing this challenge involves carefully analyzing and selecting recognizable and reliable features that persist across different optimization levels. This is crucial for ensuring the effectiveness and accuracy of our tool.

Second, addressing the lack of symbols. Binaries typically do not contain debug information; rather, most of the time they are stripped of original names of functions, variables and other symbols, and are called stripped binaries. This makes it difficult to understand the binary's structure and identify specific functions and their connections to objects. Without symbolic information available, reverse engineering becomes even harder. This requires alternative approaches to detect objects even when symbols are not available.

Last, dealing with classes that present simple layouts. For example, classes that have only one or two attributes and are small in size are really complex to recognize. The simpler a class is, the more it can be mistaken with another one of similar size. These classes lack complex structures or distinctive patterns, making it harder to recognize them successfully. Overcoming this challenge requires techniques that make the most of the limited information available to successfully distinguish these classes.

3 | Approach

In this chapter, we present our approach for the recognition of C++ objects in optimized binary applications. First, we propose an overview of the approach, then we explain in detail its three main components: Building Custom Stack Frame, Inferring Object Position, and Matching Inline Methods.

3.1. Overview

The general idea of the approach is to leverage features that are preserved in optimized binaries to identify potential object candidates for the classes we aim to recognize. These features include the size and structure of the allocated objects, as well as memory access offsets. A similar approach, also based on memory access offsets, is adopted by MemRec [2] to track objects' usage for the recognition of inline methods of C++ template classes. In contrast, our goal is to recognize C++ objects. Specifically, we focus on objects allocated on the stack, which are locally bound to the stack frame of a function.

First, we propose a technique to build a stack access model based on memory access offsets applied to the stack pointer. This model, called *Custom Stack Frame*, resembles the structure of a function's stack frame. Then, starting from the Custom Stack Frame we employ a heuristic to infer the positions where objects are likely to be allocated. We utilize several object features such as the size and internal structure of the object, and the size and location of its attributes. These features allow us to comprehend the object's memory representation and identify the potential locations where it can be accessed and modified. Therefore, we generate a set of Candidate Objects that have their base location and attributes' location aligned with the memory accesses within the Custom Stack Frame. Finally, we employ a second heuristic to refine the object candidates previously generated. Starting from the base location of a Candidate Object, we track its usage inside the binary and extract every memory access performed onto it. We examine the number, type, and offset values of the memory accesses to identify potential inline methods associated with the object. Should we find a match for an inline method, it strongly suggests that the candidate object represents an instance of the targeted class.

```

1 void stack_example() {
2     int64_t val;
3     std::string name;
4
5     // Function uses val and name
6     // [...]
7 }

```

Listing 3.1: Function with two local variables.

3.2. Building Custom Stack Frame

We can gain insight into how the stack frame of a function is organized by analyzing memory access offsets applied to the stack pointer. This helps us understand the memory footprint of data and, particularly in our case, objects instantiated inside it. First, we need to make some considerations about the stack frame and how it is accessed in an executable file.

The stack frame is a region of the stack that holds all the data related to the execution of a function. It is typically allocated when a function is called and deallocated when the function exits. Hence, the local objects created within the function have a limited lifespan, lasting until the function finishes executing. For example, in Figure 3.1, we illustrate the stack frame for the function in Listing 3.1 which has two local variables.

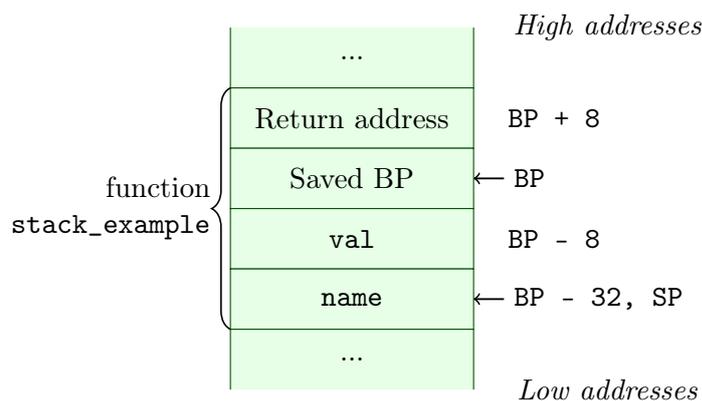


Figure 3.1: Stack frame for the function `stack_example`.

Data on the stack frame can be accessed using two memory access notations: `[SP + offset]` and `[BP - offset]`. These notations represent the memory locations in the stack frame where we can find the data. The `[SP + offset]` notation is used to access a memory location by adding an offset value to the Stack Pointer (SP) which points to the top of the stack. On the other hand, the `[BP - offset]` notation involves subtracting an

offset value from the Base Pointer (BP) which points to the base of the function's frame. However, in optimized binaries, compilers often omit the base pointer to reduce the code size and enhance the speed of execution of the program. The Base Pointer becomes a general-purpose register, effectively increasing the total number of available registers by one. Thus, we only consider the $[SP + \text{offset}]$ notation to access data within the stack frame.

We design a stack access model that resembles the structure of the actual stack frame of a function, and we call it *Custom Stack Frame*. This stack access model is composed of a set of offsets related to memory accesses made on the stack during the execution of the function. Hence, we consider the offsets applied to the stack pointer with the notation $[SP + \text{offset}]$. The offsets give us an insight into how the stack frame is organized and allow us to understand the memory footprint of the objects instantiated into it. Moreover, when available, we also take into consideration the maximum frame size found in the function's prologue. The maximum frame size indicates the total amount of space required to allocate local variables of the function and we use this value as an upper limit for our stack access model. We utilize the offsets to delineate distinct memory regions within the stack frame. These memory regions represent areas in the stack where no memory access has occurred. We can see an example of Custom Stack Frame in Figure 3.2 with a given set of memory access offsets. Analyzing the regions provides valuable information about the layout and structure of the stack frame, allowing us to infer the presence of potential objects within the function.

Memory access offsets: 8, 16, 24, 40.
Frame size: 48.

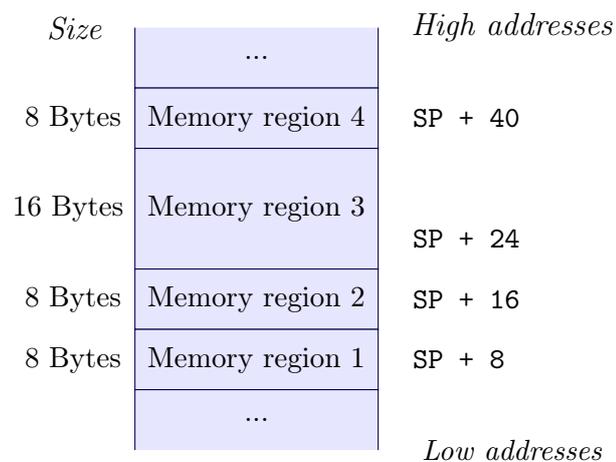


Figure 3.2: Example of Custom Stack Frame.

3.3. Inferring Object Position

We present a heuristic that allows us to identify potential candidate objects within a Custom Stack Frame. Before doing so, it is necessary to introduce additional elements into our analysis, specifically some characteristics related to the type of object (class) we want to recognize.

First, we consider the size of the object when it is allocated in memory. The size, typically measured in bytes and known at compile-time, provides valuable information about the object's memory footprint. For example, we can use the size as a boundary to identify memory regions that are likely to correspond to instances of the target object. However, size alone is not enough to successfully distinguish the locations where an object is allocated. In fact, memory accesses do not solely identify accesses to the base location of an object but also to member attributes of the object, and so memory regions could be related to these as well.

Therefore, we shall add to the features also the internal structure of the object. This includes identifying the location and size of each attribute within the object and, in case the attributes are not primitive types, any sub-attributes. This feature allows us to understand how the object is represented in memory and at what points it could be accessed and modified. By aligning the overall structure of the object with the memory regions of the Custom Stack Frame, we can effectively infer the presence and location of objects.

Our heuristic tries to identify a point within the Custom Stack Frame that might be suitable for the base position of the object. Any value within the set of offsets in the custom stack frame is considered a good candidate, including any other positive value up to the maximum offset. Then, we proceed to check that from the base location up to the boundary size of the object there are only exclusively memory accesses at the location of the object's attributes. We expect an object to ideally have only memory access exactly corresponding to its attributes. If all memory accesses within the object range correspond to the object's attributes, we call the object thus inferred a *Candidate Object*. Instead, in case we find a memory access in a location in the object range but not corresponding to one of the attributes, we discard the candidate base object location. This is because the memory regions do not align with the expected object structure.

Let us consider again the Custom Stack Frame in Figure 3.2. Now starting from this stack access model, we want to infer the position of one object of the class `std::string`. As features, we know that its size must be 32 Bytes and it has 3 member attributes at

offsets 0, 8 and 16 Bytes from the base address with sizes 8, 8 and 16 Bytes respectively. The possible base addresses for the object are 0, 8 and 16; the other values (24 and 40) are immediately discarded because the object would not have enough space on the stack frame. Then, for each remaining base address, the heuristic attempts to match the object's member attributes with the corresponding memory regions. Considering the base address of 0, we can immediately see that the third attribute should start at offset 16 and end at offset 32, but the corresponding memory region 2 is only 8 Bytes in size due to the presence of another memory access at offset 24. This is incompatible with the object memory structure and the base address 0 is discarded. The same goes for base address 16. It remains only the base address 8 as a valid option. Figure 3.3 shows the attributes aligned at offset 8, 16 and 24 compatible with the object structure: this is a Candidate Object.

Memory access offsets: **8, 16, 24, 40**
 Frame size: **48**
 Target class: **std::string**
 Target class size: **32 Bytes**
 Target class attributes' location: **0, 8, 16**

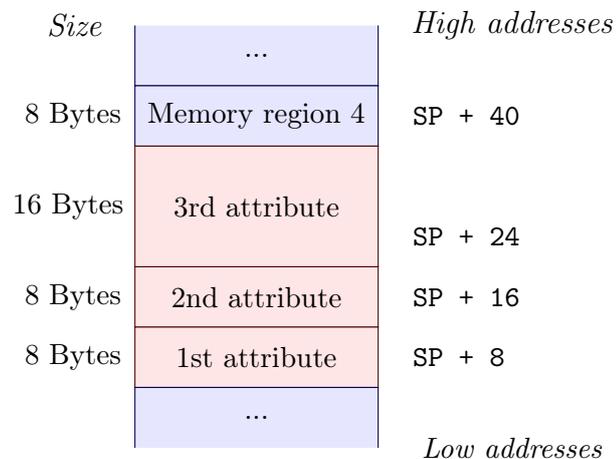


Figure 3.3: Example of Inferred Object Position.

It is important to notice that this heuristic helps us identify Candidate Objects that exhibit similar memory access patterns with the object type that we want to recognize. As a consequence, it also introduces an error in the possible case that other classes besides the targeted one share similar memory access offsets.

3.4. Matching Inline Methods

The heuristic proposed in Section 3.3 produces a substantial number of Object Candidates which can impact the accuracy of the results. So, it becomes necessary to further filter the candidates. To address this, we propose an additional heuristic based on the recognition of methods called on the objects. However, since we are dealing with optimized binaries, these methods will often be inlined by the compiler. Hence, the problem translates into the recognition of inline methods which is a much more complex task.

This heuristic leverage the knowledge of the methods defined in the classes we are targeting, in particular, their behaviors and expected memory access offsets when called onto the objects.

Thus, we consider a Candidate Object, and we track all the memory access offsets related to it, starting from its base location, inside the binary. Next, we try to match the inline method by comparing the number, type, and offset value of the memory accesses of the Candidate Object with the ones of known methods of the class. The number of memory accesses indicates the frequency of interaction with the object. The type indicates the nature of the operation performed on the memory location (read or write), while the value indicates the memory location (attribute) where the access is performed. This comparison allows us to recognize the inline methods by their corresponding memory access patterns. Therefore, if we find at least one match of an inline method, it is highly likely that the Candidate Object truly represents an instance of the targeted class. Otherwise, we discard the Candidate Object from the list of potential object instances.

4 | Implementation Details

In this chapter, we illustrate the architecture of the system as a whole and discuss the role of each module within the system. We also include the technical details of the implementation of each module, including the tools and libraries used and the algorithms employed.

4.1. System Architecture

In Chapter 3, we highlight the need to have prior knowledge related to the structure of an object and its methods in order to successfully apply the presented approach. While it is theoretically possible for a reverse engineer to manually acquire this knowledge, the process would be extremely time-consuming. To overcome this challenge we propose a two-phase system consisting of fingerprint generation and fingerprint matching. The fingerprint generation phase involves automatic generation and analysis of appropriate binaries to extract relevant features of objects and methods and store them in fingerprints. This phase eliminates the need for manual inspection and significantly reduces the time and effort required to gather the knowledge. In the fingerprint matching phase, we implement the heuristics to recognize objects in a target binary by employing the fingerprints stored in the first phase.

The framework is implemented as a static analysis tool targeting the x86_64 architecture and is developed in Python 3.10. The phases are divided into steps that are designed to perform a specific sequence of tasks. Each step is associated with a module that encapsulates the functionality relevant to that step. This modular design ensures a clear separation and organization of the codebase, allows easy development and facilitates the addition of new features. In Figure 4.1 we show an overview of the system architecture.

The first three steps of the fingerprint generation phase are, in order of execution, the Parser, Code Generator and Compiler. Their respective modules are taken from the BINO framework [4]. The Parser obtains a list of all the available methods belonging to the class under consideration. The Code Generator generates the source code necessary to call the

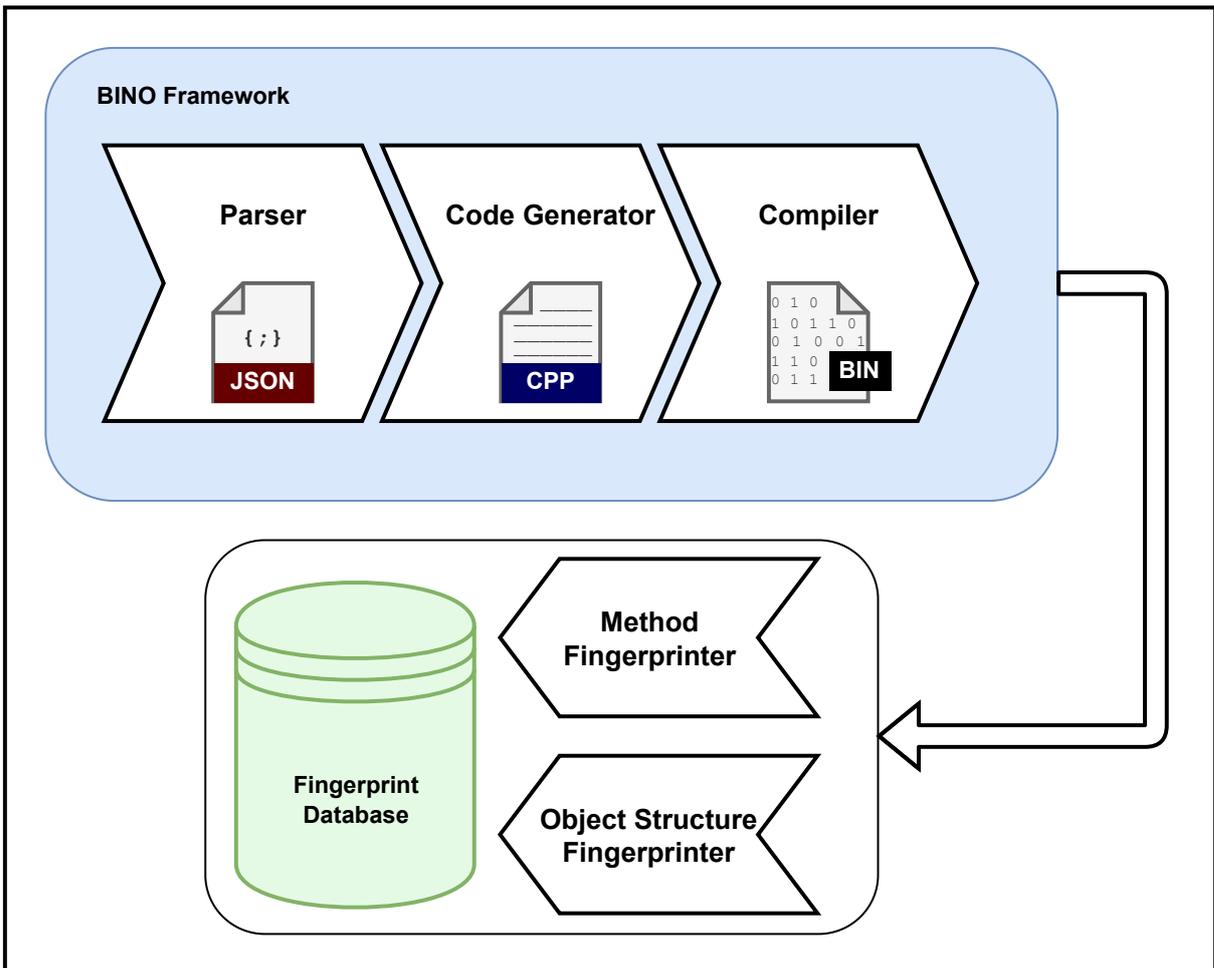
methods of the parsed class. Step three deals with the compilation of the sources into binaries with different levels of optimization.

The fourth step generates fingerprints for the methods belonging to the target class. To do this it statically analyzes the generated binaries with Angr [18, 19], then it examines the disassembled code to capture the memory access offsets and store them inside fingerprints.

Step five exploits the DWARF debug information [8] to extract the structure of an allocated object of the class, and stores it in a special fingerprint called *Object Structure Fingerprint*.

At the end of these steps, we have a database containing the features of the objects we are considering and their potential inline methods in the form of fingerprints. Thus, we are ready to move onto phase two that deals with object recognition and is implemented in the Recognizer module. First, it is necessary to statically analyze the target binary, and to do this we once again rely on Angr. Hence, we can analyze the individual functions within the binary and extract the offsets from all the instructions that perform memory access using the stack pointer. With those, we build our stack access model called Custom Stack Frame. Next, we apply the heuristic presented in Section 3.3 to guess the location of objects inside the Custom Stack Frame. Finally, we use the method fingerprints along with the second heuristic (Section 3.4) to recognize inline methods and refine the Object Candidates.

Phase 1 - Fingerprint Generation



Phase 2 - Fingerprint Matching

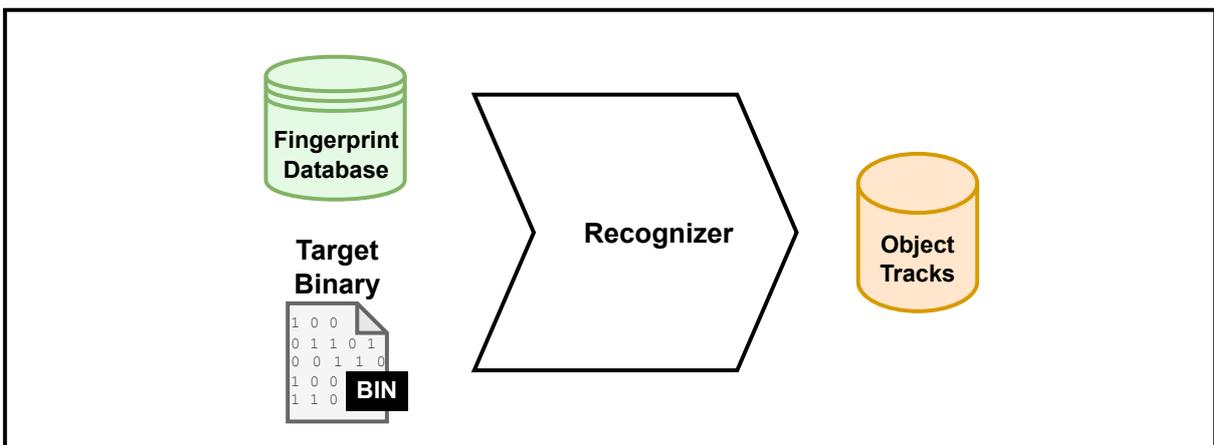


Figure 4.1: Overview of the system architecture.

4.2. System Details

In this section, we provide a detailed analysis of each module within our system. These modules work together to achieve our goal of identifying C++ objects. Notice that the modules Parser, Code Generator and Compiler are taken from the BINO framework [4], hence here we propose a summary of what they do.

4.2.1. Parser

This is the first step of the system. This module automates the process of source code generation by parsing the source code of a class based on a JSON file like the one depicted in Listing 4.1. The JSON file contains the necessary information to locate the class and build the source code. The module creates a temporary .cpp file and includes the specified library, such as `#include <string>`. Then, it utilizes the *clang-cindex* python package to expand the include, parse the C/C++ source code and build the Abstract Syntax Tree (AST). Next, it inspects the AST to extract information about the public methods of the class. This includes the name of the methods, parameter types and return value type. Finally, the module outputs an updated JSON file (Listing 4.2) with the retrieved additional information.

```

1 {
2   "standard_include": "string",
3   "optional_includes": [],
4   "class_name": "std::basic_string",
5   "optional_typedefs" :[
6     "typedef TYPE_1 char_type",
7     "typedef TYPE_1 _CharT"
8   ],
9   "tokens": [
10    {
11      "token_name" : "TYPE_1",
12      "token_values" :[
13        "char"
14      ]
15    }
16  ],
17  "template_values": [
18    {
19      "position": 0,
20      "type": "string",
21      "value": "char_type"

```

```
22     },
23     {
24         "position": 1,
25         "type": "none",
26         "value": ""
27     },
28     {
29         "position": 2,
30         "type": "none",
31         "value": ""
32     }
33 ]
34 }
```

Listing 4.1: Example of JSON input file.

```
1 {
2   ...(omissis)...
3   "namespace": "std",
4   "public": [
5     {
6       "method_name": "length",
7       "return_type": "std::basic_string<_CharT, _Traits, _Alloc>::
          size_type",
8       "const_qualified": true,
9       "operator_type": "",
10      "parameters": []
11    },
12    {...}
13    {
14      "method_name": "compare",
15      "return_type": "int",
16      "const_qualified": true,
17      "operator_type": "",
18      "parameters": [
19        "const std::basic_string<_CharT, _Traits, _Alloc> &"
20      ]
21    }
22  ]
23 }
```

Listing 4.2: Example of JSON output file.

4.2.2. Code Generator

The goal of this module is to generate source code that calls the methods of the parsed class, which will later be fingerprinted. To achieve this, it employs the C++ skeleton code in Listing 4.3 to make a generic function call.

The tokens enclosed by dollar symbols are placeholders for valid C++ code. They are replaced with the appropriate code snippets based on the information extracted from the JSON file generated by the previous module. In Listing 4.4 we show an example of the generated valid C++ code.

The wrapper function serves as a starting point for setting up the necessary objects and variables to invoke an object method. Instead of instantiating the objects and variables directly within the function, they are passed as parameters to the wrapper function. In this way, we fool the compiler which cannot infer the state of the objects or the variables and we prevent it from applying aggressive optimizations. Nevertheless, it should still inline the call to the method.

Furthermore, the two assembly snippets defined with the `asm volatile` keywords allow distinguishing the inline method assembly from the rest of the wrapper function code.

In addition, this module also supports the use of macros to generate multiple versions of the same method with different template parameters. However, we do not use them since we focus on non-template classes.

```
1 $CLASS_INCLUDE$
2 $OPTIONAL_INCLUDES$
3 $OPTIONAL_NAMESPACE$
4 $OPTIONAL_TYPEDEFS$
5
6 void wrapper($PARAMETER_LIST$)
7 {
8     $VARIABLE$;
9
10    asm volatile(
11        "or %rax, %rax;"
12        "or %rax, %rax;"
13        "or %rax, %rax;"
14    );
15
16    $FUNCTION_CALL$;
17
18    asm volatile(
19        "or %rbx, %rbx;"
```

```

20     "or %rbx, %rbx;"
21     "or %rbx, %rbx;"
22 );
23
24 $RETURN_UTILIZATION$;
25 }
26
27 int main(int argc, char const *argv[])
28 {
29     return 0;
30 }

```

Listing 4.3: Example of skeleton code with tokens.

```

1 #include <string>
2 #include <stdio.h>
3
4 using namespace std;
5
6 void wrapper(const std::basic_string<char> object_variable, const std::
   basic_string<char> & p_1)
7 {
8     asm volatile(
9         "or %rax, %rax;"
10        "or %rax, %rax;"
11        "or %rax, %rax;"
12    );
13
14    int ret_value = object_variable.compare(p_1);
15
16    asm volatile(
17        "or %rbx, %rbx;"
18        "or %rbx, %rbx;"
19        "or %rbx, %rbx;"
20    );
21
22    printf("%d\n", ret_value);
23 }
24
25 int main(int argc, char const *argv[])
26 {
27     return 0;
28 }

```

Listing 4.4: Example of source code generated for the method `std::string::compare`.

4.2.3. Compiler

This module compiles the previously generated source code files using three different optimization levels: -O2, -O3, and -Os. We select these optimization levels not only to obtain optimized binaries but also because they are the ones that enable method inlining. However, it is important to note that even if these options enable method inlining, the final decision of whether to inline methods lies with the compiler. Since it is not possible to predict the behavior of the compiler, in a later step we check that the methods have been successfully inlined.

To summarize, this module produces three optimized binaries (one per optimization level) for each source code file provided in input.

4.2.4. Method Fingerprinter

This module deals with the generation of fingerprints of the method from the binaries compiled in the previous steps. We also refer to fingerprints generated in this step as *Query Fingerprints*.

To achieve this, we disassemble and statically analyze the generated binaries with Angr [18, 19], an open-source binary analysis platform for Python. Then, for each binary, we examine the disassembled code of the wrapper function to capture the memory access offsets of the inline method.

Specifically, we know that the object on which the method is called is always passed as the first parameter of the wrapper function. However, we need to understand how this behavior is managed when we are dealing with disassembly. The architecture x86_64 adopts the System V ABI Calling Convention [15]. Hence, we know that the first parameter of a call to a function is always placed in the RDI register. Thus, at the beginning of the wrapper function, the address of our object is contained inside that register and all accesses to its attributes are performed with an offset from the RDI register. However, this is not sufficient to guarantee a correct extraction of all offsets related to the object under consideration. In fact, there is no guarantee that the address of the object will not be moved from the register RDI to other registers, so it would no longer be enough to track offsets applied only on the RDI register. We have therefore developed the algorithm in Algorithm 4.1 to analyze the instructions in the wrapper functions and follow the movement of the address of the object from the initial RDI register to other registers. During this process, we focus only on instructions that are commonly used to access and modify registers, discarding instructions that present a complex behavior or occur rarely. In this way,

we build a Track composed of assembly instructions that manipulate the object's address or its attributes, and since we are only interested in the memory accesses performed by the inline method, we only save inside the track the assembly instructions located between the two assembly snippets defined with the `asm volatile` keywords. Furthermore, the Track resembles the control flow graph (CFG) generated by Angr: the tracked instructions are contained inside nodes, which correspond to the basic blocks of the CFG. These nodes are interconnected to represent the flow of execution between them. Next, we extract all the offsets related to the object by traversing the Track in a depth-first manner, starting from the first node of the track. We identify a memory access by looking at the load/store operation performed by the instruction; we retrieve such information using the methods provided by Angr. For each memory access performed starting from the object address we extract the following parameters:

- the **mnemonic**: this indicates the type of operation that is executed by the instruction;
- the **value** of the offset: this integer indicates the attribute within the object the instruction refers to;
- the **type** of memory access: there are three types of memory access:
 - *Read*: the access is used to read the data inside the memory. For example, this happens during a MOV operation with the memory access at the source operand;
 - *Write*: the access is used to write the data inside the memory. For example, this happens during a MOV operation with the memory access at the destination operand;
 - *Read and Write*: the access is used both to read and write the data inside the memory. For example, this happens during an ADD operation with the memory access at the destination operand.

After that, we discard fingerprints that have less than 2 memory access offsets. This may happen in two cases: when a method is not inlined or when it is inlined and perform zero or one memory access. This is a limitation of the current approach and we discuss it in Chapter 6. Finally, we employ the pickle Python module to store the fingerprints in a serialized format, allowing us to save and retrieve the fingerprint data efficiently.

Algorithm 4.1 Track Registers Algorithm

Input: R set of tracked registers. I list of tracked instructions, $insn$: analyzed instruction.

Output: updated R , updated I .

```

1:  $n \leftarrow \text{numberOfOperands}(insn)$ 
2: if  $n = 2$  then
3:    $firstOp \leftarrow \text{firstOperand}(insn)$ 
4:    $secondOp \leftarrow \text{secondOperand}(insn)$ 
5:   if  $secondOp.reg \in R$  then
6:      $I.add(insn)$ 
7:     if  $firstOp.access = \text{WRITE}$  and  $secondOp.type = \text{REG}$  then
8:        $R.add(firstOp.reg)$ 
9:     end if
10:  else if  $firstOp.reg \in R$  then
11:    if  $firstOp.access = \text{READ}$  or ( $firstOp.access = \text{READ/WRITE}$  and
     $firstOp.type = \text{MEM}$ ) then
12:       $I.add(insn)$ 
13:    else if  $firstOp.access = \text{WRITE}$  then
14:       $R.remove(firstOp.reg)$ 
15:    end if
16:  end if
17: end if

```

4.2.5. Object Structure Fingerprinter

This module focuses on the generation of Object Structure Fingerprints. An Object Structure Fingerprint consists of a detailed description of the object including the size in bytes of the object, the location of its attributes and their respective sizes. This allows us to have a clear understanding of the actual internal structure of the object. We obtain these features by analyzing the DWARF debugging information [8].

To do so, we need to create a basic program that instantiates an object of the type of the class we want to recognize and calls a method of the object to prevent the compiler from optimizing away the instantiation of the object. In the previous modules, we have already created a similar program. The only difference is that we do not have an explicit instantiation of the object. However, for debugging information purposes, we are only interested in the class type of the referenced object. Indeed, this information is exactly the same for objects regardless of whether they are instantiated conventionally or used as arguments in a function.

Therefore, we compile one of the source code files generated in Section 4.2.2 with the flag `-gdwarf` to produce the DWARF debug information. With the help of the `pyelftools`

library we parse the DWARF debug information and retrieve the wrapper function's Debugging Information Entry (DIE). Then, we examine its children's DIEs to locate the first argument of the function identified by the tag `DW_TAG_formal_parameter`. Next, we examine its children DIEs to locate the first argument of the function. We do this by searching for DIEs with the tag `DW_TAG_class_type`, which represents the class type of the argument. Finally, the `DW_TAG_class_type` DIE contains everything we need: the size of the object and attributes with their own size and location inside the object. We save this information inside the Object Structure Fingerprint and utilize again the pickle Python module to store the fingerprints. In Figure 4.2 we illustrate an example of Object Structure Fingerprint.

```
Object full class name: std::basic_string<char, std::char_traits<char>,
    std::allocator<char>>
Object namespaces: ['std']
Object size: 32 bytes
Attributes:
Attribute name: _M_dataplus, location: 0, type: _Alloc_hider, size: 8
    bytes
Sub attribute name: _M_p, location: 0, type: N/A, size: 8 bytes
    Attribute name: _M_string_length, location: 8, type: long unsigned
        int, size: 8 bytes
    Attribute name: N/A, location: 16, type: N/A, size: 16 bytes
```

Figure 4.2: Example of Object Structure Fingerprint for the class `std::string`

4.2.6. Recognizer

This is the last module in the toolchain, and it is responsible for the recognition of objects inside a given binary. It requires a target binary and a fingerprint database with both Object Structure Fingerprints and Query Fingerprints. This module implements the heuristics proposed in the approach in Chapter 3.

First, we disassemble and statically analyze the target binary with Angr. With the disassembly available, we proceed to analyze every function in the binary, and we build the Custom Stack Frames as shown in Section 3.2. Since we are targeting the `x86_64` architecture, we consider `RSP` as the stack pointer register. Therefore, we extract the offsets from every assembly instruction that has `[RSP + offset]` as one of its operands.

Next, for each class we want to recognize we adopt the heuristic proposed in Section 3.3 to infer the objects' position on the stack frames. In Algorithm 4.2 we show the core of the implementation for this heuristic.

Algorithm 4.2 Guessing object position on the stack frame.

Input: O set of stack offsets values in a function. s object structure for a target class, f max stack frame size

Output: C set of Candidate Objects.

```

1: for  $o \in \text{sort}(O)$  do
2:   for  $loc \in s.\text{attribute\_locations}$  do
3:      $base \leftarrow o - loc$ 
4:     if  $base < 0$  then
5:       continue
6:     end if
7:     if  $f$  is set and  $base + s.\text{byte\_size} > f$  then
8:       continue
9:     end if
10:     $attr\_locs \leftarrow \text{getCandidateAttrLocations}(base, s)$ 
11:    if not  $\text{isAttrLocationValid}(O, base, s.\text{byte\_size}, attr\_locs)$  then
12:      continue
13:    end if
14:     $c \leftarrow \text{CandidateObject}(s.\text{class\_name}, s.\text{byte\_size}, base, attr\_locs, s.\text{namespaces})$ 
15:     $C.\text{add}(c)$ 
16:  end for
17: end for

```

After that, we create a Track starting from the Object Candidate base address with the same strategy proposed in Section 4.2.4. This track captures the complete lifecycle of the object, including its movement through function calls and contains all the offsets associated with the object.

In this case, during the track analysis, we also check the PLT calls made with the base object address passed as the first argument. If the class name of these PLT calls matches the class of the Candidate Object, it provides reliable information to accurately determine the class and the position of the object. PLT calls, or Procedure Linkage Table calls, are a mechanism used in dynamic linking to invoke functions. They are available even in stripped binaries because they are essential to resolve and link functions at runtime. However, it is important to note that PLT calls may not be used by all classes or functions hence, we cannot always rely on them.

Another aspect to consider is the overlapping of multiple tracks. For instance, the heuristic

may output two or more Candidate Objects that share a set of memory regions. Whenever this happens our algorithm generates tracks for different candidates that share one or more instructions. To address this issue, we keep only the candidate with the longest track among those and discard the other candidates.

Last, we refine the remaining candidate objects by matching the offsets of their respective track with the offsets of the inline methods contained in the Query Fingerprints. We consider two offsets to be the same if and only if they have the same value, access type and instruction mnemonic. We recognize an inline method when the track contains the exact sequence of offsets found in the Query Fingerprint. We discard candidates that do not have any matches with inline methods. In Algorithm 4.3 we show the core implementation for the offset comparison. Instead, in Figure 4.3 we provide an example of output for a successfully recognized object. Below, we explain the meaning of each item in the output:

- *Object class name*: name of the class of the recognized object;
- *Object start address*: address of the instruction where the object is first referred to inside the binary;
- *Object base location*: base location of the object inside its function's stack frame;
- *PLT call matches*: list of PLT calls encountered in the track of the object;
- *Number of function matches*: number of inline methods inside the whole track that have been matched by our heuristic;
- *Track insns*: list of assembly instructions that compose the track.

Algorithm 4.3 Matching inline methods through offset comparison

Input: Q list of offsets of a query fingerprint. T list of offsets of a target fingerprint**Output:** H list of list of offset sequences.

```

1:  $H \leftarrow$  initialize empty list
2: for  $i \leftarrow 0$  to  $(\text{length}(T) - \text{length}(Q) + 1)$  do
3:    $\text{local\_hits} \leftarrow$  initialize empty list
4:   for  $j \leftarrow 0$  to  $(\text{length}(Q) - 1)$  do
5:      $q \leftarrow Q[j]$ 
6:      $t \leftarrow T[i + j]$ 
7:     if  $t.\text{value} \neq q.\text{value}$  or  $t.\text{access} \neq q.\text{access}$  or  $t.\text{mnemonic} \neq q.\text{mnemonic}$  then
8:       break
9:     end if
10:     $\text{local\_hits.add}(T[i + j])$ 
11:  end for
12:  if  $\text{local\_hits}$  is not empty then
13:     $H.add(\text{local\_hits})$ 
14:  end if
15: end for
16: return  $H$ 

```

```

Object class name: std::basic_string<char, std::char_traits<char>, std::
    allocator<char>>
Object start address: 0x1012f1
Object base location: 0x10
PLT call matches:
    std::basic_string<...>::_M_create(unsigned long&, unsigned long)
    std::basic_string<...>::append(char const*)
    std::basic_string<...>::_M_dispose()
    std::basic_string<...>::find const(char const*, unsigned long,
        unsigned long)
Number of function matches: 42
Track insns:
    0x1012f1: lea rbp, [rsp + 0x10]
    0x1012f6: lea rax, [rsp + 0x20]
    0x101309: mov rdi, rbp
    0x10130c: mov qword ptr [rsp + 0x10], rax
    0x101322: mov qword ptr [rsp + 0x10], rax
    0x10132b: mov qword ptr [rsp + 0x20], rdx
...(omissis)...

```

Figure 4.3: Example of recognized object of the class `std::string` with its respective Track.

5 | Experimental Validation

In this chapter, we discuss the results obtained with our tool tested on a dataset of C++ projects. We also include a description of the experimental setup including the hardware adopted, and a discussion of the metrics used to evaluate the performance of our approach.

5.1. Goals

In the experiments, we want to prove the effectiveness of our hybrid approach to successfully detect C++ objects inside optimized binaries. We believe that being able to recognize an object and track its lifecycle in optimized binaries can make a significant contribution to the reverse engineering task and could be used as a starting point for new works in the field. However, object recognition and inline function recognition in binaries are still emerging areas of research and the lack of publicly available benchmark datasets makes it difficult to compare the performance of different techniques and to measure progress in the field.

Nevertheless, we have managed to collect a dataset of C++ projects from GitHub for our experiments. Although these projects are far from representing a complete benchmark dataset, they are sufficient to validate the effectiveness of our approach.

To evaluate the tool, we rely on the DWARF debugging information. This requires compiling the projects with the `-gdwarf` flag to generate the necessary debugging information. It is important to note that we use this information only to validate the results produced by our tool and not during the recognition process. Nevertheless, evaluating the performance of our tool poses a challenge in terms of obtaining reliable ground truth data. Extracting the necessary information from the DWARF debugging information proved to be a difficult task, as the DWARF format does not provide easily accessible and precise data about objects' location. Even gathering the actual number of instantiated objects is challenging due to the structure of DWARF and how debug information is emitted [16]. Furthermore, these problems intensify especially in the presence of heavy optimizations [14], which are the focus of our study.

The DWARF format does not provide a direct way to determine the total number of objects created in a program, especially when objects are used as attributes or templates of other objects. Therefore, we cannot extract from DWARF the exact number of instantiated objects per class. In addition, the precise location of objects is not always accessible. The format may not include explicit details about object locations in memory, making it challenging to accurately determine their exact positions.

In conclusion, these limitations in the DWARF format hinder our ability to accurately validate the results of our tool and compare them with ground truth.

5.2. Dataset

We collect a CSV file containing about 50000 projects from GitHub that matched the tags `C++` and `Makefile`. We download them all and discard the ones that fail to compile and those whose `CXXFLAGS` flags cannot be changed. After this step, the dataset consists of around 7000 remaining projects.

As target classes, we prioritize non-template classes over template classes due to the increased difficulty and potential ambiguity associated with the latter. Template classes introduce additional complexity due to their generic nature. The variability in their object structure and inline functions makes it more challenging to extract and analyze them accurately. In contrast, non-template classes provide a simpler representation, offering better consistency for our analysis.

Thus, we use CodeQL [11] to extract the most used non-template classes from the C++ Standard Library present in the dataset and we select the classes `std::string` and `std::thread`. For completeness, we also include in the dataset some projects of the class `std::vector` which is one of the most used template classes.

Therefore, we compile the projects in the dataset with the optimization levels `-O2`, `-O3` and `-Os`. We do not consider binaries with a size greater than 200 kilobytes for time constraints and projects that do not have the classes we want to test in our experiments.

To best cope with the DWARF limitations, we employ two different techniques to collect a good amount of ground truth data to be used for the evaluation.

The first technique focuses on simply extracting all the Debugging Information Entries (DIEs) with the `DW_TAG_variable` tag and collecting only those that match the target class type. Then, we look for the corresponding instruction addresses where these DIEs are used and compare them with the results given by our tool.

In the second technique, we examine every declared variable DIE and check if they have any attributes with a class type matching the target class type. In this case, we do not have precise information about the instruction addresses where the attributes are used. Instead, we refer to the attribute's position on the stack frame, which we retrieve through the stack unwinding process. Once again, it is important to note that this information may not always be available. We compare the attributes' stack position computed in this way with the objects' stack position found with our tool.

In the end, the dataset comprises 433 projects for `std::string`, 32 projects for `std::thread` and 165 projects for `std::vector`.

5.3. Experimental Setup

We run all the experiments within a Docker container running Ubuntu 22.04. The container is hosted on a PC powered by a Ryzen 5 3600 processor with 16 GB of RAM. The container is equipped with g++ version 11.3.0, and we use C++17 as language version (default on g++11). By using a Docker container, we eliminate potential compatibility issues and ensure that the experiments can be easily replicated across different systems.

In order to automatize the experimental process, we develop a dedicated module. This module is written in Python and is executed within the Docker container. It takes as inputs the project folders and the fingerprint DB of target classes. Then, it compiles each project, runs the recognizer module, extracts the ground truth, and compares the results.

5.4. Experiments

Firstly, we create the Fingerprint DB with the most used non-template classes from the C++ standard library found in the projects: `std::string` and `std::thread`. Regarding the experiments on `std::vector` we create a single Fingerprint DB with the most known template parameters from the C++ Standard Library such as `int`, `double`, `char` and `std::string`. This is necessary because `std::vector` is a container class; hence, its object structure remains unchanged regardless of the template parameters used (which is not the case for its methods). By fixing the templates, we can treat `std::vector` as a non-template class, which provides us with better usage of the class in dataset projects. Instead, distinguishing objects of the same class but with different templates is beyond the scope of this work.

We employ the indicators Precision, Recall, and F1-score to evaluate the performance of our approach.

Precision measures the accuracy of the correct predictions made by the system. It is the ratio of the correctly recognized objects (True Positives) to the total number of objects recognized by the system (True Positives + False Positives). A higher precision value indicates a lower rate of false positives.

Recall measures the ability of the system to identify all relevant objects. It is the ratio of the correctly recognized objects (True Positives) to the total number of objects that should have been recognized (True Positives + False Negatives). A higher recall value indicates a lower rate of false negatives.

F1-score is a combination of precision and recall and provides a balanced measure of the system's performance.

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}, \quad \text{F1-score} = 2 \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

For each experiment, we also include plots that demonstrate the scalability of our tool. These plots reveal that, across all the experiments conducted, the main algorithms exhibit short execution times.

5.4.1. Class `std::string`

This class is a non-template class implemented as a template class with a fixed template parameter `std::basic_string<char>`, therefore it suits our case. It is widely used in C++ programs for string manipulation and storage. In Table 5.1 we show the results for the `std::string` class. Each column in the table represents a different optimization level applied during the compilation process.

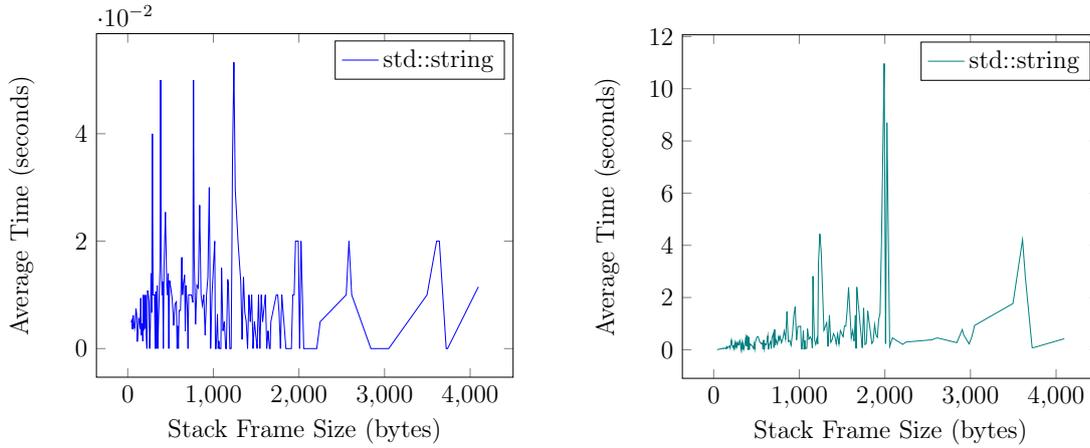
At optimization level `-O2`, there are 4323 false positives, indicating that 4323 objects were incorrectly identified as `std::string` instances. This high number of false positives can be attributed to the large number of candidate objects generated by our heuristic. However, we want to highlight again that not all the reported false positives and false negatives can truly be considered incorrect identifications. In some cases, these objects cannot be matched with the information provided by DWARF. This results in objects being mistakenly labeled as false positives and false negatives.

Similar results can be observed for optimization levels `-O3` and `-Os`. The overall performance decreases slightly as more aggressive optimizations are applied by the compiler.

Opt. lvl	-O2	-O3	-Os
True Positives	6477	6193	5065
False Positives	3982	3992	5749
False Negatives	3823	3904	3638
Precision	0.62	0.61	0.47
Recall	0.63	0.61	0.58
F1-score	0.62	0.61	0.52

Table 5.1: Results obtained for the `std::string` class in the respective dataset.

In Figure 5.1a we illustrate the average time taken by the heuristic to infer a Candidate Object inside a function with a given Stack Frame Size, while in Figure 5.1b we illustrate the average time taken by our main algorithm to generate a Track. We observe that as the Stack Frame Size increases, the average time taken by our tool to infer a Candidate Object and generate a Track also increases. However, there are some variations in the trend, with dips and peaks in the average time for certain Stack Frame Sizes. These variations may be influenced by various factors, such as the number of different memory access offsets present in the function and the complexity of the function. When there are numerous distinct memory access offsets, our heuristic has to evaluate a larger number of tentative objects' positions, leading to potentially longer execution times. Conversely, when the number of unique memory access offsets is limited, the heuristic can perform the analysis more efficiently. Instead, the complexity of the function refers to the number of instructions, branches and the level of nesting. These features impact the time required for our algorithm to create a Track for the object considered. As the function becomes more complex, our algorithm has to analyze a larger amount of instructions, resulting in longer execution times. Overall, our tool efficiently infers a candidate object of `std::string` with an average time of 8.4 milliseconds and generates an object's track with an average time of 646 milliseconds.



(a) Average time to generate a Candidate Object per size of stack frame.

(b) Average time to generate a Track per size of stack frame.

Figure 5.1: Average time depending on the function stack frame size for class `std::string`.

5.4.2. Class `std::thread`

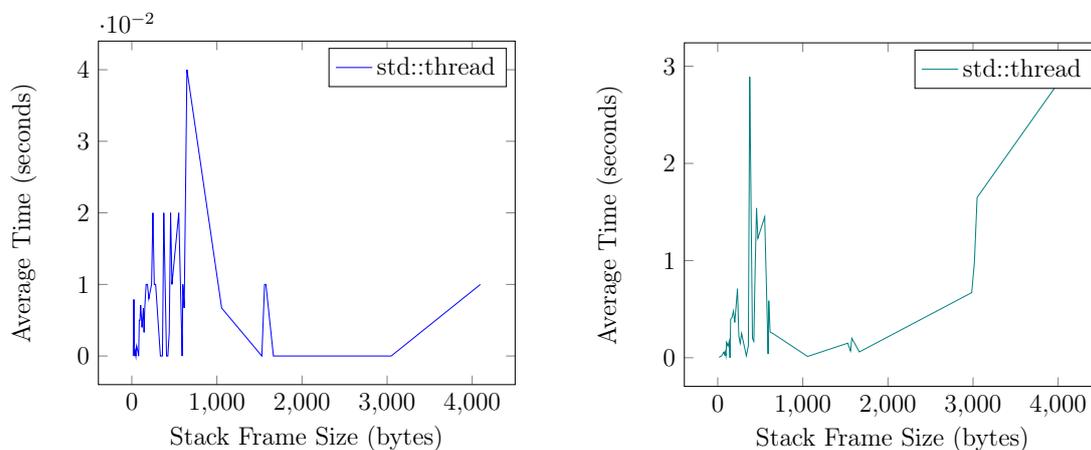
This class is another non-template class, however, we have only a few projects where it is used. We still report the results due to their relevance. Since the methods of this class are typically not inlined, we cannot generate a substantial number of query method fingerprints. Thus, for this class, we skip the inline method matching step.

Despite the small size of the class, only one attribute of 8 bytes, the performance of the recognizer for this class is great with high precision at each optimization level. The recall is slightly lower, indicating that some instances of the class are missed during the recognition process. Since we are not using inline methods to verify candidate objects, the presence of PLT calls becomes the only factor to confirm the recognition of an object. Thus, it is possible that sometimes the PLT calls are not used, and we miss some objects. However, when we find a PLT call, we are almost certain that we are dealing with an object of this class, resulting in high precision.

Opt. lvl	-O2	-O3	-Os
True Positives	82	68	74
False Positives	9	15	4
False Negatives	42	49	32
Precision	0.90	0.82	0.95
Recall	0.66	0.58	0.70
F1-score	0.76	0.68	0.80

Table 5.2: Results obtained for the `std::thread` class in the respective dataset.

Again, Figure 5.2a shows the average time taken by the heuristic to infer a Candidate Object inside a function with a given Stack Frame Size, and Figure 5.2b shows the average time taken by our main algorithm to generate a Track. The same reasoning made earlier in Section 5.4.1 applies here as well. Also in this scenario, our tool efficiently infers a candidate object of `std::thread` with an average time of 6.7 milliseconds and generates an object’s track with an average time of 458 milliseconds.



(a) Average time to generate a Candidate Object per size of stack frame.

(b) Average time to generate a Track per size of stack frame.

Figure 5.2: Average time depending on the function stack frame size for class `std::thread`.

5.4.3. Class `std::vector`

As stated before, `std::vector` is a template class, however we treat it as a non-template class by fixing the template parameter to `int`, `double`, `char` and `std::string`. By doing so, we consider all instances of `std::vector` with these fixed template parameters as a single class and evaluate their recognition jointly.

The lower precision observed for this class is due to a very high number of false positives and it can be attributed to three factors.

First, the `std::vector` object itself has a small size of only 24 bytes with three attributes of 8 bytes each. It is reasonable to imagine that such a simple structure can be found in other classes as well. This causes our heuristic to generate a high number of candidate objects that match such characteristics. For the same reasons, also the inline function matching shows poor results and with it, we can discard only a few false positives.

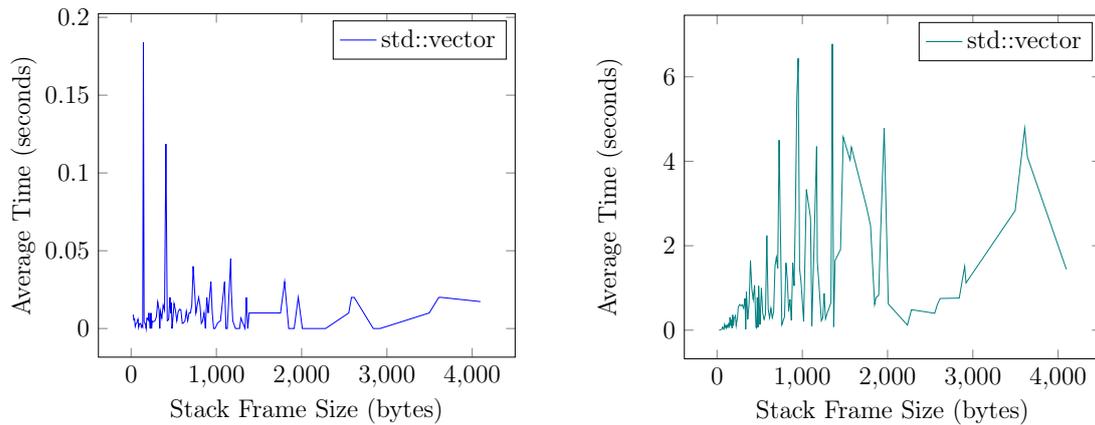
Second, the compiler almost never generates PLT calls for the base `std::vector` class when inlining methods. Therefore, we do not have such valuable information to distinguish objects, and this heavily affects the recognition process.

Last, the presence of other objects of `std::vector` with an untracked template parameter. We know that many projects in the dataset utilize custom classes as template parameters. During the tests of this class, we discard all false positives that match in the ground truth one of the vector objects with template parameter not tracked by us. This allows us to discard a good number of false positives. In fact, the presence of these objects affects the stack frame structure and thus the generation of memory regions and object candidates.

Opt. lvl	-O2	-O3	-Os
True Positives	636	545	617
False Positives	2328	2273	1558
False Negatives	622	642	605
Precision	0.21	0.19	0.28
Recall	0.51	0.46	0.50
F1-score	0.30	0.27	0.36

Table 5.3: Results obtained for the `std::vector` class in the respective dataset.

In Figure 5.3a we present the average time taken by the heuristic to infer a Candidate Object inside a function with a given Stack Frame Size, while in Figure 5.3b we show the average time taken by our main algorithm to generate a Track. The same reasoning made earlier in Section 5.4.1 applies here as well. Also in this case, our tool efficiently infers a candidate object of `std::vector` with an average time of 10.7 milliseconds and generates an object's track with an average time of 1.12 seconds.



(a) Average time to generate a Candidate Object per size of stack frame.

(b) Average time to generate a Track per size of stack frame.

Figure 5.3: Average time depending on the function stack frame size for class `std::vector`.

6 | Limitations

In this chapter, we discuss the main limitations of our approach. In addition, we provide suggestions on how each limitation could be overcome.

6.1. Simple Objects

The first limitation of our approach is related to the recognition of simple objects. Simple objects are characterized by low size and basic internal structure, typically consisting of a limited number of attributes with common sizes, such as pointers. This results in a limited number of distinct memory access offsets. Consequently, our heuristic is more likely to find more memory regions that align with the structure of these simple objects. Moreover, it is reasonable to assume that as objects become simpler, the probability of other objects sharing the same structure increases.

For these reasons, accurately identifying simple objects and distinguishing them from each other becomes more challenging. Our heuristic may generate a large number of candidate objects that exhibit similar characteristics, which introduces errors and leads to incorrect recognition.

Therefore, to address this challenge, it is necessary to supplement our approach with other information. Our suggestion is to perform an analysis of the type of content that is accessed and/or modified in the object's attributes. By examining the specific data associated with the attributes of simple objects, we could refine their recognition. This would potentially allow us to distinguish objects that correspond to the target class from those that, while sharing a similar structure, do not belong to the target class.

6.2. Methods with Low Offsets Count

Another limitation of our approach is related to matching inline methods with a low offset count. The heuristic we employ to match inline methods requires them to have at least two memory access offsets. However, it may happen that some methods, particularly the

smaller ones, perform less than two memory accesses in their code. This results in small inline methods not being identified and thus we may discard a valid candidate object when no other inline method is found.

Also in this case, the simpler the memory access patterns of a method, the greater the probability of finding its pattern inside an object's track. This can lead to incorrect object-method association and prevents the removal of a candidate object that should have been discarded.

To mitigate this limitation, different strategies shall be considered, such as mixing the heuristic with other techniques and enriching the comparison of methods by considering additional instructions that are closely related to the memory accesses performed by the functions.

7 | Future Works

In this chapter, we explore potential future directions and improvements for our tool and related research topics. As a first future work, we aim to address the limitations with the solutions discussed in Chapter 6 to enhance the accuracy and effectiveness of our approach. Furthermore, we can improve our approach by implementing the recognition of heap-allocated objects and global objects. By considering multiple object categories we expand the scope of our tool, thus performing a more in-depth analysis on optimized binaries.

In addition, we propose future works in other two areas: object recognition in non-optimized binaries and inline function recognition.

7.1. Object Recognition in Non-Optimized Binaries

The first research direction we propose involves performing object recognition in non-optimized binaries. Non-optimized binaries contain function calls, symbolic function names and other valuable information that simplify the identification of objects. By leveraging this information, we could successfully recognize C++ objects inside non-optimized binaries. This can further help reverse engineers during their analysis process.

7.2. Inline Functions Recognition

Improving the recognition of inline functions is a key area for improving our tool. In fact, this aspect has a dual relevance: our tool can be used as a starting point for inline functions recognition while at the same time, it requires better inline functions recognition to effectively distinguish objects in optimized binaries. To achieve this, we could combine the known features of a candidate object, namely the class type, tracked instructions, base address and memory accesses, with the approach adopted by the BINO framework [4].

8 | Conclusions

This work aims at providing a tool to automatically recognize C++ objects in optimized binary applications. Specifically, we select reliable features that persist across different optimization levels and store them in the form of fingerprints. Features include the size and internal structure of objects, and memory accesses offsets of inline methods.

We propose a hybrid approach composed of two heuristics to recognize C++ objects, along with generating a track that identifies their usage and lifecycle in optimized binaries. The first heuristic focuses on inferring an object's position in our stack access model. The second heuristic involves recognizing inline methods that are associated with the object, further enhancing the accuracy of the object recognition process.

We implement the tool as a two-phase system, capable of automatically generating appropriate binaries and fingerprints in the first phase and recognizing objects by employing the heuristics and the generated fingerprints in the second phase.

The results obtained from our study demonstrate the robustness of our hybrid approach for recognizing C++ objects in optimized binaries, even in the most unfavorable cases where symbols are unavailable. However, it has some limitations. Simple objects and inline methods with low offsets count are hard to identify and distinguish.

Future works could explore other techniques for improving the performance and accuracy of this approach. Others could utilize the results of this tool as a starting point for tackling different problems such as the recognition of inline functions.

In conclusion, we believe that our tool can assist reverse engineers in their complex tasks, and future integration with existing tools like IDA and Ghidra has the potential to further enhance the reverse engineering process.

Bibliography

- [1] T. Ahmed, P. Devanbu, and A. A. Sawant. Learning to find usages of library functions in optimized binaries. *IEEE Transactions on Software Engineering*, 48(10):3862–3876, 2022. doi: 10.1109/TSE.2021.3106572.
- [2] S. Bagarin. MemRec: automatic recognition of inlined binary functions from template classes. Master’s thesis, Politecnico di Milano, 2023. URL <http://hdl.handle.net/10589/203177>.
- [3] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI’07*, page 1–28, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540697350.
- [4] L. Binosi, M. Polino, M. Carminati, and S. Zanero. BINO: Automatic recognition of inline binary functions from template classes. *Computers & Security*, page 103312, 2023. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2023.103312>. URL <https://www.sciencedirect.com/science/article/pii/S0167404823002225>.
- [5] J. Caballero and Z. Lin. Type inference on executables. *ACM Comput. Surv.*, 48(4), may 2016. ISSN 0360-0300. doi: 10.1145/2896499. URL <https://doi.org/10.1145/2896499>.
- [6] S. H. H. Ding, B. C. M. Fung, and P. Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, 2019. doi: 10.1109/SP.2019.00003.
- [7] Y. Duan, X. Li, J. Wang, and H. Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and distributed system security symposium*, 2020.
- [8] *DWARF Debugging Information Format Version 5*. DWARF Debugging Information Format Committee, 2017. URL <https://dwarfstd.org/doc/DWARF5.pdf>. Published standard.

- [9] R. A. Erinfolami and A. Prakash. DeClassifier: Class-inheritance inference engine for optimized C++ binaries. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 28–40, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367523. doi: 10.1145/3321705.3329833. URL <https://doi.org/10.1145/3321705.3329833>.
- [10] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. SmartDec: Approaching C++ decompilation. In *2011 18th Working Conference on Reverse Engineering*, pages 347–356, Los Alamitos, CA, USA, oct 2011. IEEE Computer Society. doi: 10.1109/WCRE.2011.49. URL <https://doi.ieeecomputersociety.org/10.1109/WCRE.2011.49>.
- [11] GitHub Inc. CodeQL, 2023. URL <https://codeql.github.com/>.
- [12] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. pub-ISO, fifth edition, dec 2017. URL <https://www.iso.org/standard/68564.html>.
- [13] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326490. doi: 10.1145/2556464.2556465. URL <https://doi.org/10.1145/2556464.2556465>.
- [14] Y. Li, S. Ding, Q. Zhang, and D. Italiano. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1052–1065, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386020. URL <https://doi.org/10.1145/3385412.3386020>.
- [15] H. Lu, M. Matz, M. Girkar, J. Hubicka, A. Jaeger, and M. Mitchell. *System V Application Binary Interface*, 2023. URL <https://gitlab.com/x86-psABIs/x86-64-ABI>. Version 1.0.
- [16] C. Pang, T. Zhang, R. Yu, B. Mao, and J. Xu. Ground truth for binary disassembly is not easy. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2479–2495, Boston, MA, Aug. 2022. USENIX Association. ISBN 978-1-939133-31-1. URL <https://www.usenix.org/conference/usenixsecurity22/presentation/pang-chengbin>.
- [17] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines. Us-

- ing logic programming to recover C++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 426–441, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243793. URL <https://doi.org/10.1145/3243734.3243793>.
- [18] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016. doi: 10.1109/SP.2016.17.
- [19] F. Wang and Y. Shoshitaishvili. Angr - the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9, 2017. doi: 10.1109/SecDev.2017.14.
- [20] X. Wang, X. Xu, Q. Li, M. Yuan, and J. Xue. Recovering container class types in C++ binaries. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, page 131–143. IEEE Press, 2022. ISBN 9781665405843. doi: 10.1109/CGO53902.2022.9741274. URL <https://doi.org/10.1109/CGO53902.2022.9741274>.
- [21] K. Yoo and R. Barua. Recovery of object oriented features from C++ binaries. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 231–238, 2014. doi: 10.1109/APSEC.2014.44.

List of Figures

3.1	Stack frame for the function <code>stack_example</code>	12
3.2	Example of Custom Stack Frame.	13
3.3	Example of Inferred Object Position.	15
4.1	Overview of the system architecture.	19
4.2	Example of Object Structure Fingerprint for the class <code>std::string</code>	27
4.3	Example of recognized object of the class <code>std::string</code> with its respective Track.	30
5.1	Average time depending on the function stack frame size for class <code>std::string</code> . 36	
5.2	Average time depending on the function stack frame size for class <code>std::thread</code> . 37	
5.3	Average time depending on the function stack frame size for class <code>std::vector</code> . 39	

List of Tables

- 5.1 Results obtained for the `std::string` class in the respective dataset. . . . 35
- 5.2 Results obtained for the `std::thread` class in the respective dataset. . . . 37
- 5.3 Results obtained for the `std::vector` class in the respective dataset. . . . 38

Listings

3.1	Function with two local variables.	12
4.1	Example of JSON input file.	20
4.2	Example of JSON output file.	21
4.3	Example of skeleton code with tokens.	22
4.4	Example of source code generated for the method <code>std::string::compare</code>	23

Ringraziamenti

Ringrazio il Prof. Stefano Zanero per avermi dato l'opportunità di lavorare a questa tesi. L'ambiente stimolante e collaborativo del NECSTLab ha arricchito la mia esperienza di ricerca.

Grazie al Prof. Mario Polino per avermi introdotto nel mondo del reverse engineering e avermi ispirato con la sua competenza e il suo entusiasmo.

Ringrazio Lorenzo Binosi per i suoi preziosi consigli e per la sua disponibilità. Grazie per avermi fornito spunti fondamentali e per avermi indirizzato nei momenti di indecisione.

Un grazie va anche a tutti gli amici che mi hanno aiutato a “staccare la spina”. A quelli di sempre e a quelli più recenti. Ai fedelissimi della *Gentooneria* e agli *Amici di Romolo*.

Ringrazio di cuore i miei genitori per avermi sempre sostenuto e per avermi permesso di portare a termine gli studi universitari.

