



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

EXECUTIVE SUMMARY OF THE THESIS

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

LAUREA MAGISTRALE IN COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

Author: SAMUELE NEGRINI

Advisor: PROF. MARIO POLINO

Co-advisor: PROF. STEFANO ZANERO

Academic year: 2022-2023

1. Introduction

Reverse engineering is the process of analyzing the binary code of a program to understand its behavior, particularly useful when the corresponding source code is unavailable. With the increasing popularity of C++, which supports object-oriented programming, identifying objects in a binary is of fundamental importance in the reverse engineering field.

In this study, we address the problem of recognizing C++ objects in optimized or stripped binaries, thus in situations where relevant information, such as symbols and function calls, is missing or unavailable due to compiler optimizations. We focus on *stack-allocated objects*, the most common type of allocated object. 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.

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 approach in a 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 of objects and methods 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 optimized binaries based on the features captured in the stored fingerprints. 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`.

2. State of the art

The recognition of objects and classes falls under a wider topic called type inference. Numerous studies explore the generic area of type inference [3], focusing on diverse aspects such as the identification of primitive types, complex data structures and class hierarchies.

Balakrishnan et al. [1] 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.

Other works try to identify C++ objects in optimized binaries. OOAnalyzer [4] employs symbolic execution and a Prolog-based reasoning system to recover C++ abstractions from executables. However, its performance may be affected by the computational complexity of symbolic execution and is tailored to analyze only Windows executables.

One recent study [5], 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.

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.

3. 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. Function inlining replaces a function call with the function's code, making it difficult to identify the boundaries between different functions. 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. Objects in C++ 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 replaced with an inline function, thus complicating their usage for object recognition. By recognizing an object, we can provide reverse engineers with insights into 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.

In conclusion, recognizing C++ objects in optimized binaries is a complex and challenging problem for reverse engineers, and addressing the discussed aspects is essential to enable effective reverse engineering.

4. Approach

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. Therefore, we present a stack access model and two heuristics to infer objects' positions and detect associated inline methods.

4.1. Building Custom Stack Frame

We focus on identifying stack-allocated objects, which are locally bound to the stack frame of a function. Thus, 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. It is composed of a set of offsets related to memory accesses made on the stack during the execution of the function. Data on the stack frame can be accessed using two memory access notations applied to the Stack Pointer (SP), $[SP + \text{offset}]$, or the Base Pointer (BP), $[BP + \text{offset}]$. However, in optimized binaries, compilers often omit the base pointer to reduce the code size and enhance the speed of execution of the program. Hence, we only consider the $[SP + \text{offset}]$ notation to access data within the stack frame and its respective 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 1 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.

4.2. Inferring Object Position

We present a heuristic that allows us to identify potential candidate objects within a Custom Stack Frame by employing features related to the type of object (class) we want to recog-

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

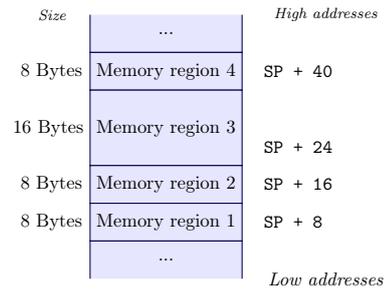


Figure 1: Example of Custom Stack Frame.

nize. We consider the size and the internal structure 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. The internal structure includes the location and size of each attribute within the object and, in case the attributes are not primitive types, any sub-attributes. These features allow 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 accesses 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.

4.3. Matching Inline Methods

The heuristic proposed in Section 4.2 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.

5. Implementation

We implement the tool as a two-phase system consisting of fingerprint generation and fingerprint matching (Figure 2). The first phase involves automatic generation and analysis of binaries to extract relevant features 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 second phase, we implement the heuristics to recognize objects in a target binary by employing the fingerprints stored in the first phase. The first three steps of the fin-

gerprint generation phase are, in order of execution, the Parser, Code Generator and Compiler. Their respective modules are taken from the BINO framework [2]. 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 (-O2, -O3 and -Os).

The *Method Fingerprinter* generates fingerprints for the methods belonging to the target class. To do this it statically analyzes the generated binaries with Angr, an open-source binary analysis platform for Python. Next, starting from the known object, we build a Track composed of assembly instructions that manipulate the object's address or its attributes. Then, we traverse the Track in a depth-first manner starting from the first node of the track, identify memory accesses by looking at the load/store operations performed by the instructions and store them inside fingerprints. Last, we discard fingerprints that have less than two memory access offsets as they are hard to distinguish.

The *Object Structure Fingerprinter* exploits the DWARF debug information 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 move onto phase two, *Recognizer* module, that deals with objects recognition. First, we statically analyze the target binary with Angr. Hence, we inspect 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 the Custom Stack Frame. Next, we apply the heuristic presented in Section 4.2 to guess the location of objects inside the Custom Stack Frame. Finally, we use the method fingerprints along with the second heuristic (Section 4.3) to recognize inline methods and refine the Object Candidates. During the track analysis, we also check the PLT calls made with the base object address passed as the first argument. When available, they provide reliable informa-

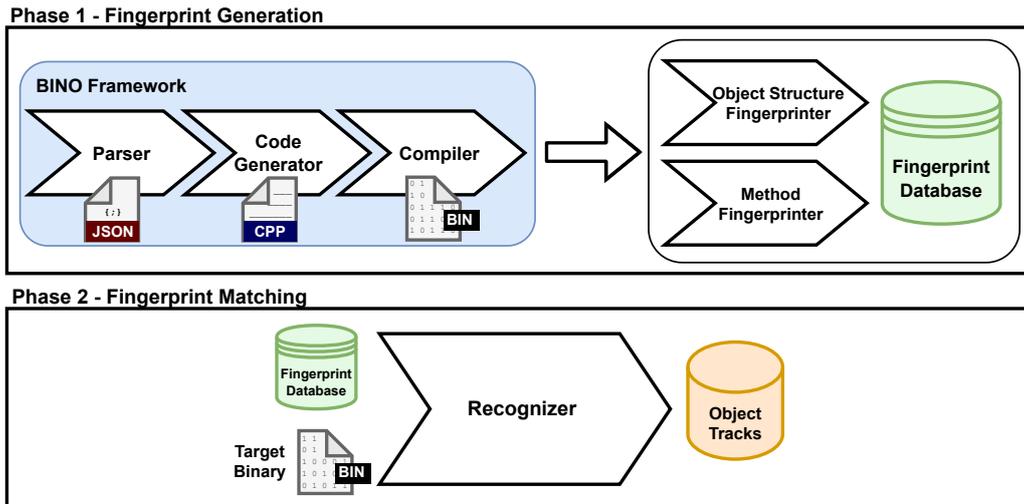


Figure 2: Overview of the system architecture.

tion to accurately determine the class and the position of the object.

6. Experiments

We evaluate the tool on a dataset of GitHub C++ projects compiled with different optimization levels and the `-gdwarf` flag to generate the debugging information. The dataset comprises 433 projects for `std::string`, 32 projects for `std::thread` and 165 projects for `std::vector`. We compare the results of our tool against the ground truth extracted from the DWARF debugging information. We use this information only to validate the results produced by our tool and not during the recognition process. Nevertheless, extracting the necessary information proved to be a difficult task, as the DWARF format does not provide easily accessible and precise data about objects' locations. Therefore, we cannot extract from DWARF the exact number of instantiated objects per class. Furthermore, these problems intensify especially in the presence of heavy optimizations, which are the focus of our study. In Table 1 we show the results of the experiments. Our tool performs well in recognizing objects of non-template classes `std::string` and `std::thread`, but struggles with the objects of template class `std::vector`, due to its simple and common structure and the absence of PLT calls. Overall, it efficiently infers a Candidate Object with an average time of 8.6 milliseconds and generates an object's Track with an average time of 740 milliseconds.

7. Limitations and Future Works

We identify two main limitations of our approach: recognition of simple objects and detection of inline methods with a low offset count. Simple objects, due to their low size and basic internal structure, have 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. In fact, as objects become simpler, the probability of other objects sharing the same structure increases. Inline methods with a low offset count are not identified by our approach, hence 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. Future works could explore object recognition in non-optimized binaries. Such binaries contain function calls and symbols that simplify the identification of objects. Other works could focus on the recognition of heap-allocated objects and global objects. Another future direction is related to the recognition of inline functions. 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 with the approach

std::string				std::thread				std::vector			
Lvl	-O2	-O3	-Os	Lvl	-O2	-O3	-Os	Lvl	-O2	-O3	-Os
TPs	6477	6193	5065	TPs	82	68	74	TPs	636	545	617
FPS	3982	3992	5749	FPS	9	15	4	FPS	2328	2273	1558
FNs	3823	3904	3638	FNs	42	49	32	FNs	622	642	605
Prec	0.62	0.61	0.47	Prec	0.90	0.82	0.95	Prec	0.21	0.19	0.28
Rec	0.63	0.61	0.58	Rec	0.66	0.58	0.70	Rec	0.51	0.46	0.50
F1	0.62	0.61	0.52	F1	0.76	0.68	0.80	F1	0.30	0.27	0.36

Table 1: Results obtained for the `std::string`, `std::thread` and `std::vector` classes.

adopted by the BINO framework [2].

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 inte-

gration with existing tools like IDA and Ghidra has the potential to further enhance the reverse engineering process.

References

- [1] Gogul Balakrishnan and Thomas 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.
- [2] Lorenzo Binosi, Mario Polino, Michele Carminati, and Stefano Zanero. BINO: Automatic recognition of inline binary functions from template classes. *Computers & Security*, page 103312, 2023.
- [3] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Comput. Surv.*, 48(4), may 2016.
- [4] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using 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.
- [5] Xudong Wang, Xuezheng Xu, Qingan Li, Mengting Yuan, and Jingling 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.