Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Håvard Pettersen

Discovering call graphs in binary programs from unknown instruction set architectures

Master's thesis in Master of Science in Informatics
Supervisor: Donn Morrison
June 2023

NTNU
Norwegian University of Science and Technology

Håvard Pettersen

Discovering call graphs in binary programs from unknown instruction set architectures

Master's thesis in Master of Science in Informatics
Supervisor: Donn Morrison
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Norwegian University of
Science and Technology

Sammendrag

Denne studien tar for seg utfordringen med *reverse engineering* av binærfiler fra ukjente instruksjonssett-arkitekturer, en kompleks oppgave med potensielle implikasjoner for programvarevedlikehold og cyber-sikkerhet. Den foreslåtte løsningen er et nyskapende program designet for å oppdage opkoder og lage *call* grafer, noe som potensielt kan lette og forenkle prosessen med *reverse engineering*. Empirisk testing på ulike binærfiler i forskjellige arkitekturer viser at programmet nøyaktig kan oppdage spesifikke opkoder og håndtere data med støy effektivt. Det bemerkes imidlertid at det krever at binærfilen har visse egenskaper, som fast lengde på instruksjoner. Til tross for disse begrensningene, kan programmet være et verdifullt verktøy for *reverse engineering*, hvor det er en klar mangel i nåværende forskning, samt legge grunnlaget for videre forskning.

Abstract

This study addresses the challenge of reverse engineering binaries from unknown instruction set architectures, a complex task with potential implications for software maintenance and cyber-security. The proposed solution is a novel program designed to detect opcodes and create call graphs, potentially facilitating and simplifying the reverse engineering process. Empirical testing on various binary files in different architectures shows that the program can accurately detect specific opcodes and handle noisy data effectively. However, it is noted that it requires the binary file to have certain properties, such as fixed-length instruction size. Despite these limitations, the program may provide a valuable tool for reverse engineering, offering a new tool where there is a clear research gap, while laying the groundwork for further research.

Table of Contents

Sammendrag i

Abstract ii

List of Figures iv

List of Tables v

1 Introduction 1

2 Background 3

2.1 Domain knowledge on reverse engineering 3

2.2 Related work 9

3 Methodology 11

3.1 Proposed solution 11

3.2 Analysis strategy and data generation 16

4 Results 17

4.1 Experimental setup 17

4.2 Return and call opcode detection 19

4.3 OCP-Score as a metric 22

4.4 Call graph creation 27

5 Discussion 28

6 Conclusion 30

7 References 31

List of Figures

1 Compilation steps in C 3
2 LLVM Compiler architecture 4
3 Instruction format of the MIPS architecture 5
4 Example C function 5
5 MIPS architecture - example program 5
6 Aarch64 architecture - example program 6
7 x86_64 architecture - example program 6
8 ELF file structure 7
9 Example call graph 8
10 Context of the proposed solution 11
11 Frontend user interface 15
12 Instruction length parameter and OCP-Score 22
13 Call opcode length parameter and OCP-Score 23
14 Return opcode length parameter and OCP-Score 24
15 PC offset parameter and OCP-Score 25
16 Return to function distance parameter and OCP-Score 26
17 Call graph - source code 27
18 Call graph - source code with merged functions 27
19 Call graph - generated by the program 27

List of Tables

1 Explanation of the API parameters 14

2 Binaries used in the analysis 17

3 API parameters used in the analysis 18

4 Top 5 most probable candidates - OpenVPN MIPS 19

5 Top 5 most probable candidates - OpenVPN Aarch64 20

6 Top 5 most probable candidates - cURL MIPS 20

7 Top 5 most probable candidates - cURL Aarch64 20

8 Top 5 most probable candidates - Cross-compiled cURL MIPS . . . 21

9 Top 5 most probable candidates - cURL x86_64 21

List of Algorithms

1 Detect call graph from binary 12

2 Get potential edges - relative adressing 12

1 Introduction

In an era defined by rapid technological advancements and a vast amount of different systems, further amplified by the rise of the Internet of Things, the importance of understanding and decoding the inner workings of software cannot be understated. At the heart of this is the concept of reverse engineering. Reverse engineering in the context of software, is the practice of inspecting, deconstructing, and analyzing the structure and operation of a binary file in order to understand its architecture, design, and functionality. This is often done without access to source code or design documentation, making it a painstaking, yet critical, part of software analysis and security.

The reverse engineering process is notably important in areas such as cyber-security, where detecting and understanding malware is key to developing and maintaining robust security. It also plays a vital role in maintaining and debugging legacy software and firmware, where the original documentation or developers may not be available. For these reasons, reverse engineering is a critical skill in the digital age and an important area in need of further research and development efforts.

In the broader context, several tools and methods have been developed over time to aid the reverse engineering process. Most of these tools require *a posteriori* knowledge about the instruction set architectures of the binary being analyzed, often explicitly needing the architecture in its dataset, which poses limitations and challenges when dealing with unknown instruction set architectures.

The current methods for reverse engineering binaries from unknown instruction set architectures are limited and often involve invasive procedures such as hardware decapsulation, which can be costly, slow, and potentially damaging to the hardware [1]. Additionally, obfuscation measures are often used to deliberately make the process even more challenging and time-consuming. Examples of such techniques are custom virtual machines used to execute the binary file.

When looking at the process of reverse engineering from a methodological perspective, a common practice is detecting important functions and focusing the reverse engineering efforts on them, so-called sub-routine scanning [2]. Hence, a tool capable of generating call graphs for binaries would alleviate much of the needed efforts in the current reverse engineering process.

There is a clear need for heuristic tools that can assist reverse engineers in extracting meaningful information from such binaries without prior knowledge of the instruction set architectures. With this in mind, the following research questions are formulated:

RQ1. Can call-graph information be heuristically deduced from binary programs of an unknown instruction set architecture?

RQ2. How effective is an approach such as this and what are its limitations?

The central contribution of this study is the development and validation of a novel program that can be used to detect opcodes and generate call graphs from binaries with unknown instruction set architectures. The program is evaluated in detail, revealing its capabilities and limitations. During the development of the program, an equation for attributing a probability value to opcodes has been formulated. This equation not only ranks the opcodes based on likelihood, showing the user only the most probable pairs but also provides a quick-reference value for an easy overview. In addition to this, a user-friendly web-based interface has been developed, enabling users to view the generated call graphs. The collection of software, tools, algorithms, and theory developed in this project is hereby referred to as a *framework*.

The structure of the rest of the report is as follows: The next chapter gives the necessary background needed to understand the report and reviews the current state-of-the-art in reverse engineering. The subsequent chapters describe the proposed solution, followed by a detailed analysis and discussion of the effectiveness and limitations of the program, validated through testing multiple binary files from different architectures. Lastly, the findings are summarized, and potential avenues for further research are proposed.

2 Background

This section will go through the basics of reverse engineering, as well as describe the current state-of-the-art. It is in large parts based on the work done in the preliminary project [3].

2.1 Domain knowledge on reverse engineering

The domain knowledge required for this project ranges from how programs are compiled to assembly language, binary file structure, and instruction set architecture. These topics are massive, and only a high-level overview of the most important aspects will be covered here.

Compilation

Compilation is the process of transforming code in a high-level language into machine-readable code. In the C programming language, the compilation of a program comprises several steps, as depicted in Figure 1.

The responsibility of each step is as follows: The pre-processor copies header files to the source code, expands macros, and removes comments; the compiler is responsible for transforming the source code to an assembly language targeting a specific architecture, as well as doing optimization; the assembler transforms the assembly code into machine code; the linker links additional libraries to the program.

Figure 1: The steps a C compiler goes through transform source code to machine code.

Compilers are typically divided into two parts: the frontend and the backend, as can be seen in Figure 2. The frontend is responsible for converting the source code to a low-level intermediate representation, similar to assembly language. Loop unrolling and other code optimizations are done in this intermediate representation. Finally, the backend is responsible for converting the intermediate representation to assembly or machine code, targeting a specific architecture. This separation is useful since one can reuse the same frontend for multiple architectures, and reuse the same backend for multiple programming languages.

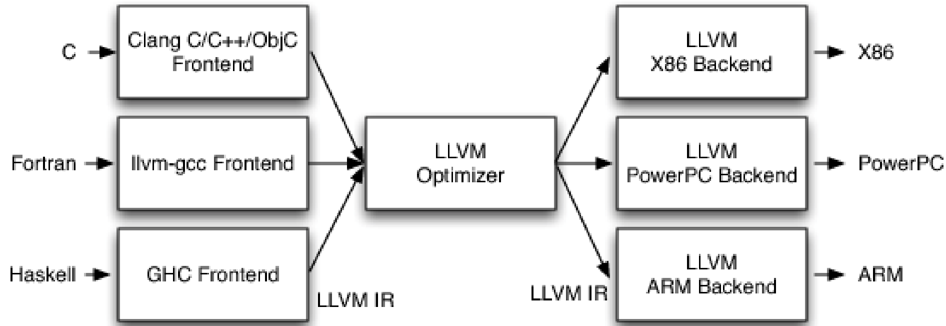


Figure 2: Architecture for the LLVM compiler, showcasing a single optimizer for different frontend languages and backend architectures [4].

Instruction set architecture

An instruction set architecture serves as an abstract model of the computer on which software runs, and when compiling a program, one must target a specific instruction set architecture. This instruction set architecture defines the supported instructions, data types, addressing modes, and other relevant aspects of the architecture. Consequently, a program compiled for eg. the x86_64 architecture will not execute on a computer with ARM architecture without the use of emulators.

Assembly code is a mnemonic of machine code, meaning there is a one-to-one mapping between them. For instance, an instruction `mov r1 #2` could be assembled into the following bytes: `0x5e83a2`. In much the same way, disassembly would mean translating the bytes back to the original assembly instructions. Typically, an instruction consists of an opcode, which specifies the operation, and operands, which determine the values to operate on. These operand values can include memory addresses, immediate values, or registers.

The instruction format demarcates the bits of an instruction representing the opcode, and the bits representing the operands. An instruction format can either be fixed length, where all instructions are the same length, or variable length, allowing instructions to be of different lengths, as is the case in the x86_64 architecture. Figure 3 illustrates an example of such an instruction format, in this case for the MIPS architecture. Additionally, the endianness of the instruction set architecture is an important consideration, indicating the order in which bytes are stored. An instruction stored as `0x1234` would be interpreted as `0x1234` for big-endian, and `0x3412` for little-endian.

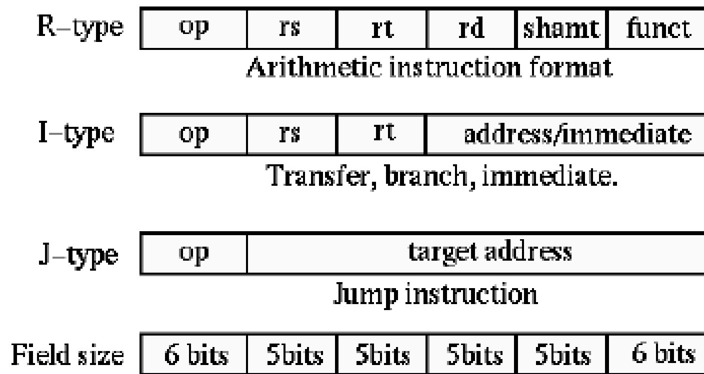


Figure 3: Instruction format of the MIPS architecture, showcasing the arithmetic, jump and I-type instruction formats [5].

To further illustrate the differences between different instruction set architectures, the function presented in Figure 4 has been compiled for different architectures using the online *Compiler Explorer* tool [1], which can be seen in Figures 5, 6, and 7.

```
int func(int a, int b) {
    return a + b;
}
```

Figure 4: Example C function which sums two integers.

```
addiu   sp,sp,-8
sw      s8,4(sp)
move    s8,sp
sw      a0,8(s8)
sw      a1,12(s8)
lw      v1,8(s8)
lw      v0,12(s8)
nop
addu    v0,v1,v0
move    sp,s8
lw      s8,4(sp)
addiu   sp,sp,8
jr      ra
```

(a) MIPS assembly code

```
27dbfff8
afbe0004
03a0f025
afc40008
afc5000c
8fc30008
8fc2000c
00000000
00621021
03c9e825
8fbe0004
27bd0008
03e00008
```

(b) MIPS machine code

Figure 5: Assembly and machine code for the example C program in figure 4 compiled for the MIPS architecture.

[1] https://godbolt.org/

```
sub     sp, sp, #0x10
str     w0, [sp, #12]
str     w1, [sp, #8]
ldr     w1, [sp, #12]
ldr     w0, [sp, #8]
add     w0, w1, w0
add     sp, sp, #0x10
ret
```

(a) Aarch64 assembly code

```
d10043ff
b9000fe0
b9000be1
b9400fe1
v9400be0
0b000020
910043ff
465f03c0
```

(b) Aarch64 machine code

Figure 6: Assembly and machine code for the example C program in figure 4 compiled for the Aarch64 architecture.

```
push    rbp
mov     rbp,rsp
mov     [rbp-0x4],edi
mov     [rbp-0x8],esi
mov     edx, [rbp-0x4]
mov     eax, [rbp-0x8]
add     eax,edx
pop     rbp
ret
nop     [rax+rax*1+0x0]
```

(a) x86_64 assembly code

```
55
48 89 e5
89 7d fc
89 75 f8
8b 55 fc
8b 45 f8
01 d0
5d
c3
0f if 44 00 00
```

(b) x86_64 machine code

Figure 7: Assembly and machine code for the example C program in figure 4 compiled for the x86_64 architecture.

Binary file structure

After compilation, the program is typically stored in a binary file format, with the Executable and Linkable Format (ELF) being the most common. The reason this is of interest is that a binary file often contains more than just instructions, it also contains data and metadata. In the case of ELF files, they consist of sections and segments of different types of data. In Figure 8, which shows the contents of an ELF file, we are specifically interested in the *.text* segment, as that is where the instructions are stored. When dealing with an unfamiliar file format, it is of interest to identify the start and end of the corresponding *.text* segment, to accurately isolate and extract the instructions.

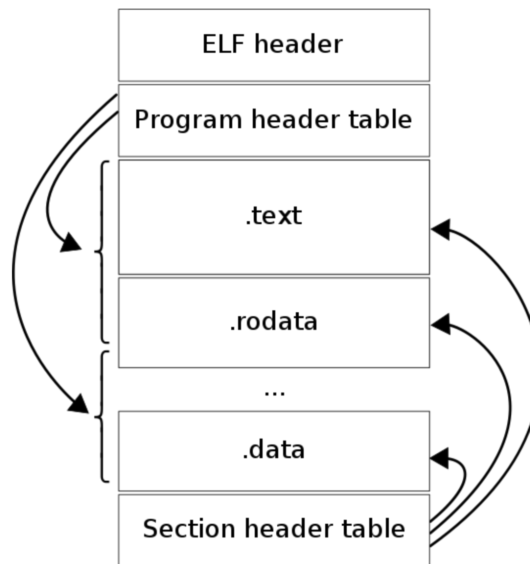


Figure 8: ELF file structure [6].

Call graphs

To detect call graphs in a binary with an unknown instruction set architecture, the most relevant task is detecting the function boundaries, namely the byte position at which a function starts, and where it ends. Notably, all the architectures depicted in Figures 5, 6, and 7 exhibit distinct function epilogues and prologues, through return instructions and stack operations, respectively. An example of a call graph for a simple program, consisting of a main function that calls two other functions, can be seen in Figure 9. A more complex call graph may have characteristics such as recursion and loops.

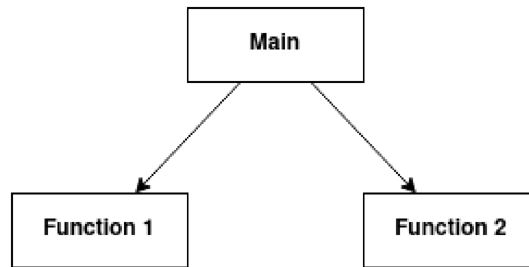


Figure 9: Call graph constructed from a program containing a main function which calls function 1 and function 2.

Addressing modes

Call instructions generally reference other functions in one of three ways: absolute addressing, where the operand of the instruction is the address we want to access; relative addressing, where the operand of the instruction contains the offset from the current address; and register addressing, where the address of the callee is stored and accessed through a register. In general, it is simpler to detect where a function points when it uses absolute and relative addressing, with register addressing being difficult without runtime knowledge.

Other challenges

Developing a static analysis method capable of disassembling any binary file for any architecture is an exceedingly difficult problem. However, it is feasible to create a tool that aids in the disassembly process for a subset of architectures. In interpreted languages like Python, it is possible to create an executable using tools like PyInstaller[2]. However, without prior knowledge of it being a Python executable, conducting static analysis on such files can prove challenging. This is due to both the interpreter being bundled with the executable, and the code itself being interpreted byte code. As a general rule of thumb, a program written in a lower-level language like C will be easier to extract meaningful information from than a program written in a higher-level language.

[2]https://pyinstaller.org/en/stable/

2.2 Related work

There is quite a bit of research within the field of reverse engineering, ranging from malware detection to architecture classification. However, most research are targeting a specific set of architectures, while research on the analysis of unknown instruction set architectures is scarce.

Clemens [7] uses a dataset of 16,000 binaries from 20 different architectures to detect endianness and instruction set architecture. The approach relied heavily on byte frequency distributions as features, suggesting that they retained sufficient opcode information for accurate instruction set architecture classification. The approach is similar to the approach of Kairajarvi *et al.* [8], and relies on the instruction set architecture being part of the training data.

Sharif *et al.* [9] developed a system called Rotalume to reverse engineer binaries that have been obfuscated using programs such as VMprotect [3]. This approach was however dependent on executing the binary in a protected environment, in order to extract runtime information, which makes the approach unfeasible for binaries with an unknown architecture.

On the methodology front, an observational study demonstrated the three-phased process of reverse engineering: overview, sub-component scanning, and focused experimentation [2]. The program discussed in this report falls primarily within the overview phase. It provides reverse engineers with a high-level visualization, in the form of a call graph, facilitating an informed decision in the sub-component scanning phase.

There also exist several tools that may assist in different parts of the reverse engineering process. `IDA Pro` [4], a widely used disassembly tool, allows interactive disassembly of binaries across popular architectures. Similarly, the Python library `angr` [5], assists in symbolic analysis of binary files, provided the architecture of the binary is known and supported. `Objdump` is a popular unix library that is useful in disassembling binaries, but it can only do so if the file contains appropriate headers, for example, an ELF file. `ILspy` [6] and `JD Project` [7] are tools that can be used to disassemble .NET and Java binaries respectively.

[3]https://vmpsoft.com/
[4]https://www.hex-rays.com/ida-pro/
[5]https://angr.io/
[6]https://github.com/icsharpcode/ILSpy
[7]http://java-decompiler.github.io/

In an unpublished work by Chernov *et al.* [10], a heuristic approach is presented, where they detect instruction set architectural features in binaries with unknown instruction set architecture. They present multiple assumptions of the binary file of an unknown architecture: Call opcodes usually have the absolute address of a function as an operand, a function prologue is closely spatially located to the previous functions epilogue, and call and return opcodes are amongst the most commonly used opcodes. Through the use of frequency distributions and address matching, they were able to detect subroutines and control flow in binaries, through only static analysis of the binary file. The work done in this report is based on the same assumptions made by Chernov *et al.* but differs in its implementation. It will also be the first *published* research on this specific topic.

Most studies discussed have necessitated prior knowledge of the instruction set architecture, with only the last paper presented by Chernov *et al.* focusing on unknown architectures. As such there is a clear research gap identified in this area, which this paper aims to contribute towards.

3 Methodology

This section introduces the proposed solution, as well as the data generation and analysis strategy.

3.1 Proposed solution

The proposed solution[8] consists of a set of services and algorithms to analyse the binary, hereby referred to as the *program*, and a web-based user interface, hereby referred to as the *frontend*. The program, developed in Python [9], takes as input a binary file and a list of parameters, and it gives as output a list of potential call graphs along with their corresponding probability. Figure 10 shows how a possible context where a reverse engineer might use the program as part of the process of reverse engineering a binary towards a high-level representation.

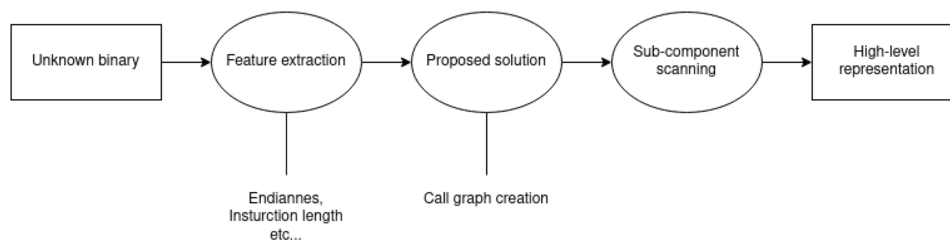


Figure 10: Context of the use of the proposed solution, occurring between architectural feature extraction and sub-component scanning.

The rationale for choosing Python as the programming language was primarily due to its simplicity, enabling quick iterations of the program and ease of development, with the most obvious trade-off being speed and efficiency. During development, the program was validated against a small binary from the Chip8 architecture, due to its instruction format being well-suited for static analysis. In Sections 4 and 5 we will further analyse and discuss the program against a more common and comprehensive set of architectures.

During the development of the program, modularity and modifiability were key non-functional requirements of the program, since future work on it may be done. To achieve this, the program was separated into functions with single responsibilities. A high-level pseudo-code of the main algorithm can be seen in Algorithm 1. The *extract_instruction* function separates the bytes of the binary into a list of instructions, based on the provided instruction length, and file offsets. The *get_potential_edges* function finds all instructions with the given call opcode where its operand points to

[8] https://github.com/haavapet/binary-analysis

[9] As of the writing of this report, a CLI version of this project is in development using the Rust programming language. This project should ensure much faster analysis of big binaries, as well as ease of use by virtue of being a CLI tool. The project can be viewed at https://github.com/haavapet/binary-analysis-rs, and will feature additional functionality such as optional parameters and multi-threading.

a valid instruction, with either relative or absolute addressing. The *filter_valid_edges* function validates edges by confirming that the given return opcode is one of the few instructions above the called instruction, to ensure there is a distinct function epilogue followed by a function prologue.

Algorithm 1 Detect call graph from binary

```
instructions = extract_instructions(...)          ▷ bytes → List[Instructions]
top_candidates = Heap(...)
for call candidates do
    potential_edges = get_potential_edges(...)
    for return candidates do
        valid_edges = filter_valid_edges(...)
        probability = get_probability(...)
        store_candidate_in_heap(...)
    end for
end for
for candidate in top_candidates do
    create_graph_for_candidate(...)
end for
return candidates_with_graph
```

As an illustration of the modularity and modifiability of the program; during the later stages of development, support for relative addressing was added as an optional functionality. This required only adding an if/else clause based on a new parameter to the API, and modifying the *get_potential_edges* function with a few lines of code. A simplified view of this functionality can be seen in Algorithm 2.

Algorithm 2 Get potential edges - relative adressing

```
potential_call_instructions = get_instructions_with_opcode(...)
for potential_call_instructions do
    signed_operand = int_to_signed_int(instruction.operand)
    if signed_operand hits relative instruction address then
        add_edge(...)
    end if
end for
return edges
```

It is highly recommended to inspect the documentation in the source code for a more thorough understanding of the algorithms.

As part of developing the program to detect call graphs in binary files, a formula for computing and associating a probability score to a given call opcode and return opcode has been created. The formula can be seen in Equation 1, and is hereby referred to as **Opcode Candidacy Probability Score** (**OCP-Score**).

$$\textbf{OCP-Score} = \frac{2 \cdot (\textbf{length valid edges}) + (\textbf{length potential edge})}{3 \cdot (\textbf{call count})} \tag{1}$$

call count refers to the number of instructions with the given call opcode. **potential call edges** refers to the number of edges associated with the candidate call opcode, in particular, those instances where the operand points to a valid address. **length valid edges** refers to the number of edges where there is a function epilogue, specifically a candidate return instruction, above the called instruction.

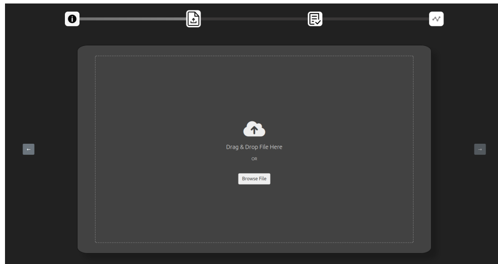
The OCP-Score is normalized to a value between 0 and 1, explained by the constraint that **length valid edges** and **length potential edges** are always less than or equal to **call count**. It is worth noting that **length valid edges** is weighted more heavily than **potential call edges**, due to being more strongly correlated with only call instructions as opposed to call and branch instructions. The OCP-Score will be evaluated and discussed further in Sections 4 and 5.

The analysis of the binary file requires a handful of parameters provided alongside the file itself. The parameters, their type, and a description can be seen in Table 1. All parameters are currently required by the API, however, a potential modification with sane defaults and increased search space, could require only the first three parameters while keeping the rest optional, which would greatly increase usability.

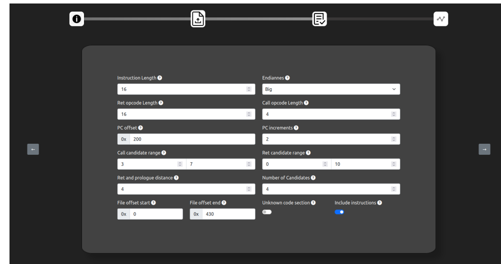
Table 1: Explanation of the API parameters.

Parameter	**Type**	**Description**
instructionLength	int	Length of an instruction in bits
retOpcodeLength	int	Length of instruction return opcode in bits
callOpcodeLength	int	Length of instruction call opcode in bits
fileOffset	int	Byte position of code section start in binary
fileOffsetEnd	int	Byte position of code section end in binary
pcOffset	int	Address of first instruction
pcIncPerInstr	int	Distance between the address of each instruction
endiannes	string	"big" or "little"
nrCandidates	int	How many graph candidates to return
callCandidateRange	int, int	Only search the [x:y] most popular instruction with a bitmask of callOpcodeLength as a potential call candidate
retCandidateRange	int, int	Only search the [x:y] most popular instruction with a bitmask of retOpcodeLength as a potential return candidate
returnToFunctionPrologueDistance	int	Distance from function epilogue (return instruction) to function prologue (call operand address)
unknownCodeEntry	bool	Search the binary for the most optimal fileOffset and fileOffsetEnd, drastically increases runtime
includeInstructions	bool	Include instructions in the result object. Recommended False for big binaries if rendering graph
isRelativeAddressing	bool	Relative or absolute addressing for call operands

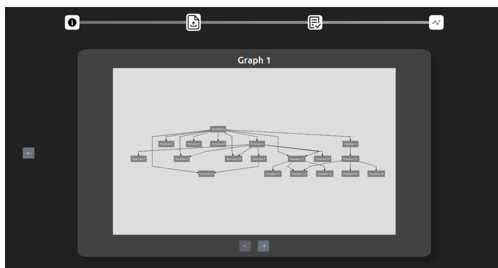
Alongside the aforementioned Python program, a frontend written in React is also included. Due to this inclusion, the FastAPI library [10] was chosen as the API for the program, enabling the integration between the program and the frontend through a RESTful interface. The frontend provides a simple graphical interface where the user can upload a file, input the required parameters, and then display the created call graphs. Figure 11 shows the interface of the frontend.



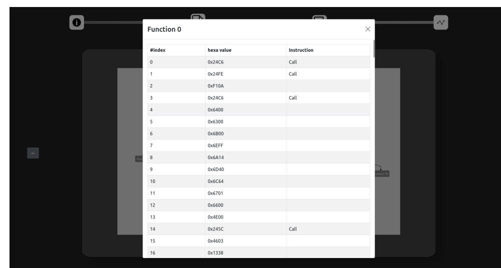
(a) Upload file page



(b) Form page



(c) Display graph page



(d) Modal after clicking function 0

Figure 11: User interface of the frontend solution, showing the different pages for uploading a binary file, entering parameters, and displaying the generated call graph.

The development of the program has incorporated multiple practices from the XP framework, such as test-driven development, coding standards, and continuous integration [11, p. 49]. Among the tests are end-to-end tests to validate the program as a whole, along with several tests for the different modular parts of the program, such as instruction extraction. Continuous integration through GitHub actions ensures that incremental updates to the program are validated, and pre-commit hooks ensure that faulty code is not pushed to the GitHub repository. In addition to tests, linters for both the program and the frontend, as well as static type checking for the program, are also included. This ensures that certain code standards are held, and enforces consistent code styles throughout the code.

The project can be run locally using either docker-compose [11], or the programming environments' respective package managers: poetry [12] for Python, and npm [13] for React. For detailed instructions, see the included README file.

[10] https://fastapi.tiangolo.com/
[11] https://www.docker.com/
[12] https://python-poetry.org/
[13] https://www.npmjs.com/

3.2 Analysis strategy and data generation

The forthcoming analysis will analyse and validate three integral parts of the program. The first part is to input the program with the correct parameters and ensure that the returned call opcode and return opcode are correct. The second part evaluates the assigned OCP-Score under different inputs, to detect how noisy and potentially faulty data affects the output. The third part will be looking at the created call graph of a small binary file, and comparing it to a call graph created by inspecting the source code.

There were multiple considerations taken into account when choosing programs and architectures for the opcode detection and OCP-Score evaluation analysis. Firstly, the architecture should conform to a fixed-length instruction format, as that is what the program expects and should be evaluated against. However, a reference binary with a variable-length instruction format has been included to provide insights into the behavior of the program under such conditions. Secondly, the binary should contain sufficient immediate or relative call and return instructions. Lastly, the programs used should be commonly used, complex, and written in a low-level language like C.

The most important characteristic of the binary used for the call graph creation was that the program is sufficiently small, this is to ensure the creation of a human-readable call graph, as well as reducing the manual labor required to create a call graph from inspecting the source code. In addition to this, it is important that the binary conforms to the same properties as mentioned in the previous paragraph.

The output of the program was fed to a Python script, which uses Matplotlib [14] to produce the graphs seen in the forthcoming section. The specific binaries used in this analysis, along with their associated parameters, are explained in detail in Section 4.1.

[14] https://matplotlib.org/

4 Results

This section will analyse three important parts of the proposed solution: opcode detection, call graph creation, and the OCP-Score. In addition to this, the experimental setup will be described such that the results can be reproduced.

4.1 Experimental setup

In order to reproduce the results in the following analysis, one can use the binaries in Table 2, with the corresponding list of parameters found in Table 3.

There are seven binaries in total, and they are all included in the accompanying GitHub repository. The binaries span three different programs: cURL, OpenVPN, and Chipquarium.

Four architectures are used in the analysis. The MIPS and Aarch64 architectures conform to a fixed-length instruction format, while the x86_64 architecture uses a variable-length instruction format. The Chipquarium binary, used in the call graph analysis, is compiled for the Chip8 architecture and is also the binary used during the development and testing of the program.

During the analysis it was found that the cURL MIPS binary had almost no occurrence of immediate call instructions, hence a new version of cURL MIPS was cross-compiled and included for reference. The binary was compiled with the `-no-pie` , `-fno-pie` , and `-mplt` compiler flags, causing more frequent use of immediate call instructions.

Table 2: Binaries used in the analysis.

Program	Architecture	Source	Version	Used for
cURL	MIPS	GitHub	Undisclosed	Opcode detection & OCP-Score evaluation
cURL	Aarch64	cURL website	8.0.1	Opcode detection & OCP-Score evaluation
cURL	MIPS	Cross-compiled from source	8.0.1	Opcode detection
cURL	x86_64	Compiled from source	8.0.1	Opcode detection
OpenVPN	MIPS	GitHub	Undisclosed	Opcode detection & OCP-Score evaluation
OpenVPN	Aarch64	Arch repository	2.6.4-1	Opcode detection & OCP-Score evaluation
Chipquarium	Chip8	GitHub	1.0	Call graph

The parameters found in Table 3 were obtained by analysing the binaries with command-line tools such as **readelf**, **size**, and **objdump**, and by reading the documentation of the architectures.

A specific modification was implemented for the MIPS and Aarch64 parameters in this process: the **pcOffset** and **pcIncPerInstr** parameters were divided by a value of 4 compared to what their architecture specified for them. This adjustment serves to emulate a left shift operation on the operand of the call instruction by a value of 2, as suggested by the architectural references [12][13].

As mentioned earlier, there is also a cross-compiled binary of cURL for the MIPS architecture, this binary has the same parameters as the cURL MIPS binary, with the exception of **fileOffsetEnd** which has a value of 567492 instead.

Table 3: API parameters used in the analysis.

Binaries / **Parameters**	**cURL MIPS**	**cURL Aarch64**	**cURL x86_64**	**OpenVPN MIPS**	**OpenVPN Aarch64**	**Chipquarium Chip8**
instructionLength	32	32	32	32	32	16
retOpcodeLength	32	32	8	32	32	16
callOpcodeLength	6	6	8	6	6	4
fileOffset	0	4096	0	0	68416	0
fileOffsetEnd	94560	2163136	501176	1782196	753456	1072
pcOffset	0x100000	ANY	0x100000	0x100000	ANY	0x200
pcIncPerInstr	1	1	1	1	1	2
endiannes	"big"	"little"	"little	"big"	"little"	"big"
nrCandidates	5	5	5	5	5	5
callCandidateRange	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20
retCandidateRange	0, 10	0, 10	0, 10	0, 10	0, 10	0, 10
returnToFunction-PrologueDistance	3	3	3	3	3	3
unknownCodeEntry	False	False	False	False	False	False
includeInstructions	False	False	False	False	False	False
isRelativeAddressing	False	True	False	False	True	False

4.2 Return and call opcode detection

Tables 4, 5, 6, and 7 present the top five probable candidates for call and return opcodes for the OpenVPN MIPS, OpenVPN Aarch64, cURL MIPS, and cURL Aarch64 binaries, respectively. The correctly identified opcodes emerge as most probable with a substantial margin in Tables 4 and 7, whereas the remaining two tables reveal contrasting outcomes.

Upon examining the binary in Table 5, it is observed that the call instruction appears approximately 1600 times. However, roughly 1200 of these instances are deemed invalid as they lack a preceding return instruction above the called function. The NOP instruction (`0xD503201F`) frequently precedes function prologues in this binary, which accounts for its higher OCP-Score as a potential return opcode.

The results also differ for the binary featured in Table 6. In this case, the call instruction and return instruction are encountered about 40 and 200 times, respectively. The return instruction does not rank within the top 20 instructions, and as a result, it falls outside the predefined search range defined by the **retCandidateRange** parameter. Despite this, the opcode associated with the branch instruction, `0x08`, is assigned a OCP-Score of roughly 0.4.

Table 4: Top 5 most probable return and call opcodes from the OpenVPN binary with MIPS architecture.

OCP-Score	Call opcode	Return opcode	Correct
0.866	`0x0C000000`	`0x03E00008`	✓
0.449	`0x08000000`	`0x0320F809`	
0.412	`0x08000000`	`0x8FBC0018`	
0.388	`0x08000000`	`0xAFA20010`	
0.373	`0x08000000`	`0x00001021`	

Table 5: Top 5 most probable return and call opcodes from the OpenVPN binary with Aarch64 architecture.

OCP-Score	Call opcode	Return opcode	Correct
0.612	0x94000000	0xD503201F	
0.478	0x94000000	0xD65F03C0	✓
0.426	0x94000000	0xD63F0060	
0.398	0x14000000	0xD63F0060	
0.396	0x14000000	0x72001C1F	

Table 6: Top 5 most probable return and call opcodes from the cURL binary with MIPS architecture.

OCP-Score	Call opcode	Return opcode	Correct
0.389	0x08000000	0x8FBC0010	
0.376	0x08000000	0x8FBC0020	
0.368	0x0C000000	0x8FBC0010	
0.365	0x08000000	0x8FBC0018	
0.357	0x08000000	0x0320F809	

Table 7: Top 5 most probable return and call opcodes from the cURL binary with Aarch64 architecture.

OCP-Score	Call opcode	Return opcode	Correct
0.698	0x94000000	0xD65F03C0	✓
0.367	0x94000000	0xA94153F3	
0.353	0x14000000	0xD65F03C0	
0.346	0x94000000	0x52800020	
0.334	0x94000000	0xAA1303E0	

As mentioned earlier, an additional binary for cURL MIPS was cross-compiled with additional compiler flags enabled, to ensure an appropriate frequency of immediate call instructions. The results for this binary, along with the x86_64 binary, which uses a variable-length instruction format, can be seen in Tables 8 and 9, respectively.

Table 8: Top 5 most probable return and call opcodes from the cross-compiled cURL binary with MIPS architecture.

OCP-Score	**Call opcode**	**Return opcode**	**Correct**
0.598	0x0C000000	0x03E00008	✓
0.378	0x0C000000	0x00001025	
0.345	0x0C000000	0x00002825	
0.342	0x0C000000	0x24020001	
0.340	0x0C000000	0x02002025	

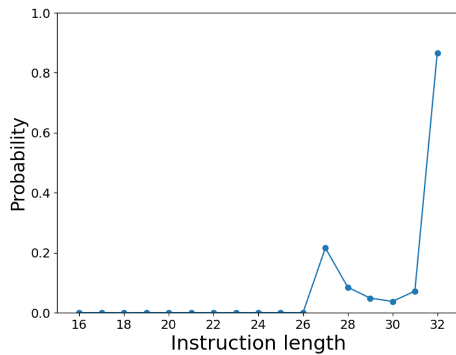
Table 9: Top 5 most probable return and call opcodes from the cURL binary with x86_64 architecture.

OCP-Score	**Call opcode**	**Return opcode**	**Correct**
0.001	0xF00000000	0x4800000000	
0.001	0xF00000000	0x8B00000000	
0.001	0xF00000000	0xFF00000000	
0.001	0xF00000000	0x2400000000	
0.001	0xF00000000	0x8900000000	

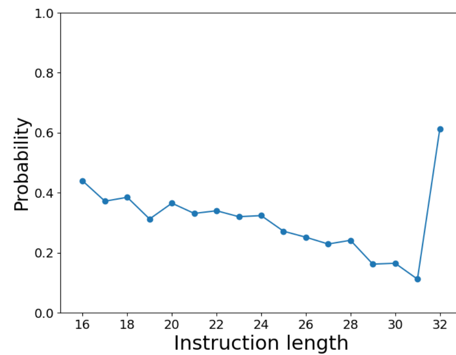
4.3 OCP-Score as a metric

Figure 12 displays the maximum OCP-Score corresponding to various values of the instruction length variable. The MIPS binaries exhibit a low OCP-Score for all values except the correct one. In contrast, the Aarch64 architecture binaries display greater variability, with higher OCP-Score for incorrect values.

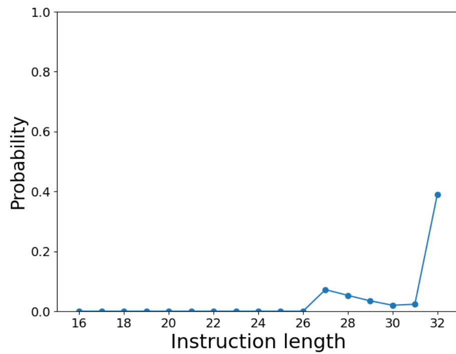
This discrepancy may arise due to the differing addressing modes employed in the call instructions. In the MIPS architecture with absolute addressing, a valid call operand must point towards an address located between the first and the address of the last instruction, for instance, within the range of `0x400160` and `0x5B3290`in the case of the OpenVPN MIPS binary. Conversely, a relative call instruction may involve lower values, which are arguably more common in noisy data. For example, an operand value of 4 would point toward the instruction preceding the call instruction itself.



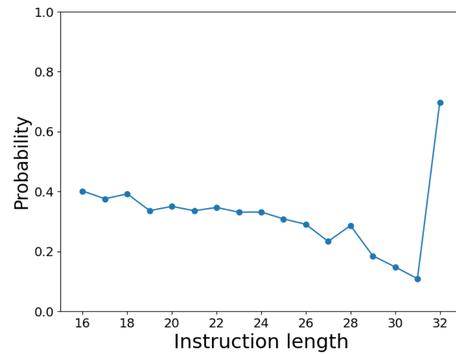
(a) OpenVPN in MIPS architecture



(b) OpenVPN in Aarch64 architecture



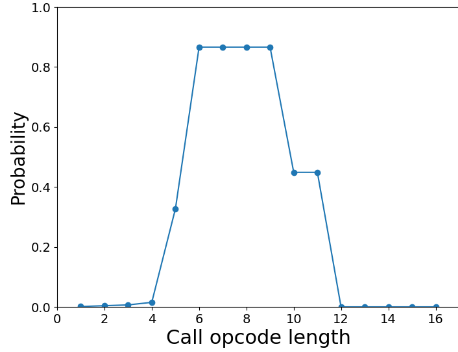
(c) cURL in MIPS architecture



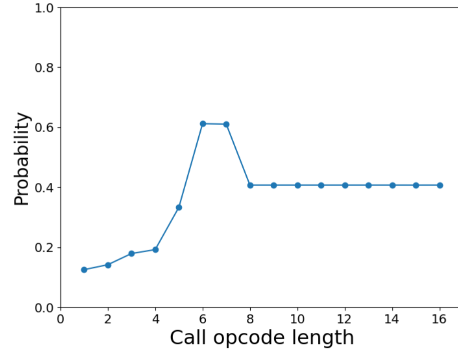
(d) cURL in Aarch64 architecture

Figure 12: OCP-Score for different inputs of the *instructionLength* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.

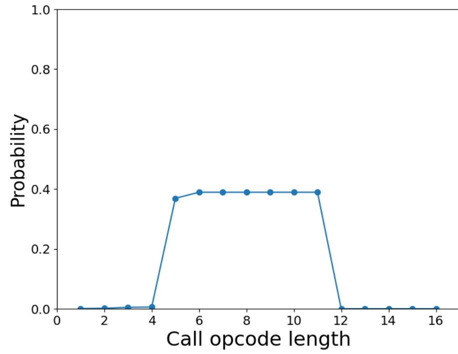
Figure 13 displays the maximum OCP-Score corresponding to various values of call opcode length. The data suggests that multiple values close to the correct value give a high OCP-Score. The explanation for this is presented in Section 5.



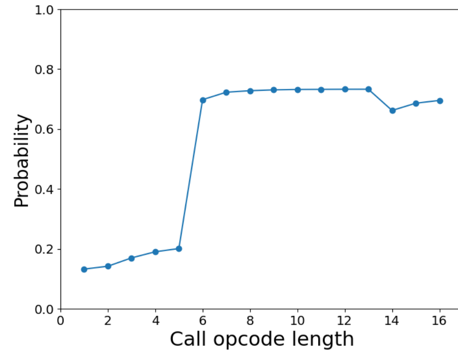
(a) OpenVPN in MIPS architecture



(b) OpenVPN in Aarch64 architecture



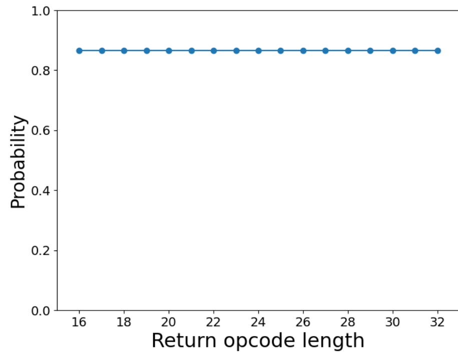
(c) cURL in MIPS architecture



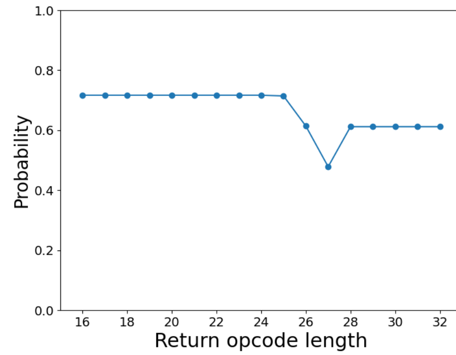
(d) cURL in Aarch64 architecture

Figure 13: OCP-Score for different inputs of the *callOpcodeLength* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.

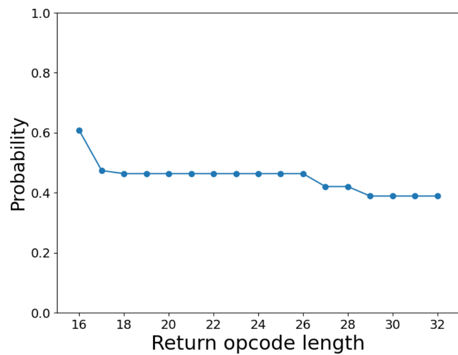
Figure 14 displays the maximum OCP-Score corresponding to various values of return opcode length. Looking at the data it seems that the change in value is not notably significant between different values. In general, when decreasing the return opcode length, we either see an increase in OCP-Score due to the set of instructions considered to be a return instruction increasing, or a decrease due to another incorrect but more frequent set pushing it out of the return search range.



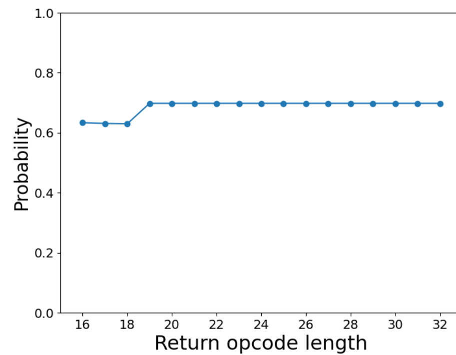
(a) OpenVPN in MIPS architecture



(b) OpenVPN in Aarch64 architecture



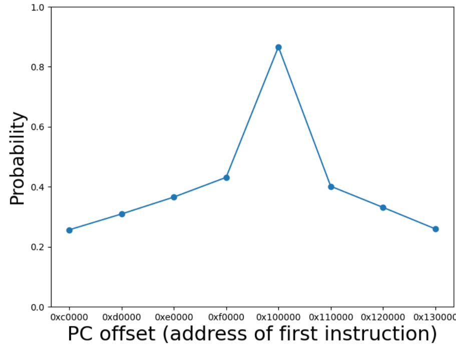
(c) cURL in MIPS architecture



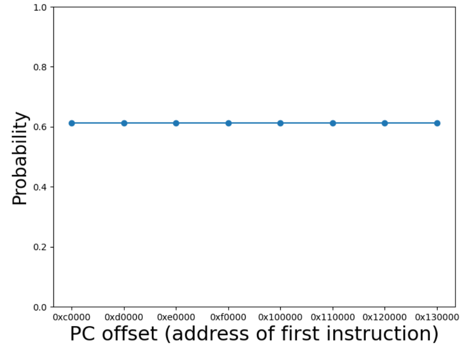
(d) cURL in Aarch64 architecture

Figure 14: OCP-Score for different inputs of the *retOpcodeLength* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.

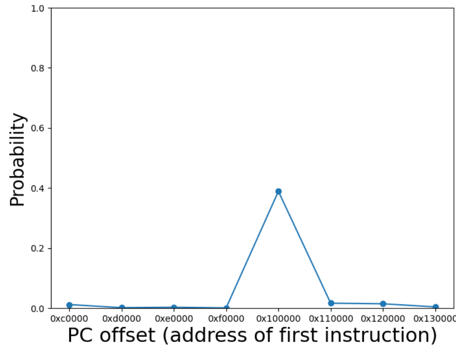
Figure 15 displays the maximum OCP-Score corresponding to various values of PC offset. It is important to clarify that these values do not affect the particular instructions read from the binary file, but rather assign a specific address to each instruction. For example, with a PC offset value of `0x1000`, the first instruction would be given an address of `0x1000`. From the results, it is evident that the PC offset value has no impact on relative addressing, which aligns with expectations. However, in the context of absolute addressing in MIPS, the correct value gives a significantly higher OCP-Score.



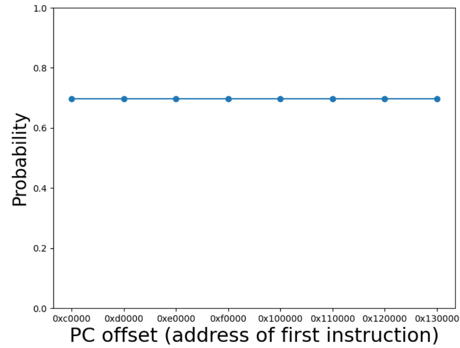
(a) OpenVPN in MIPS architecture



(b) OpenVPN in Aarch64 architecture



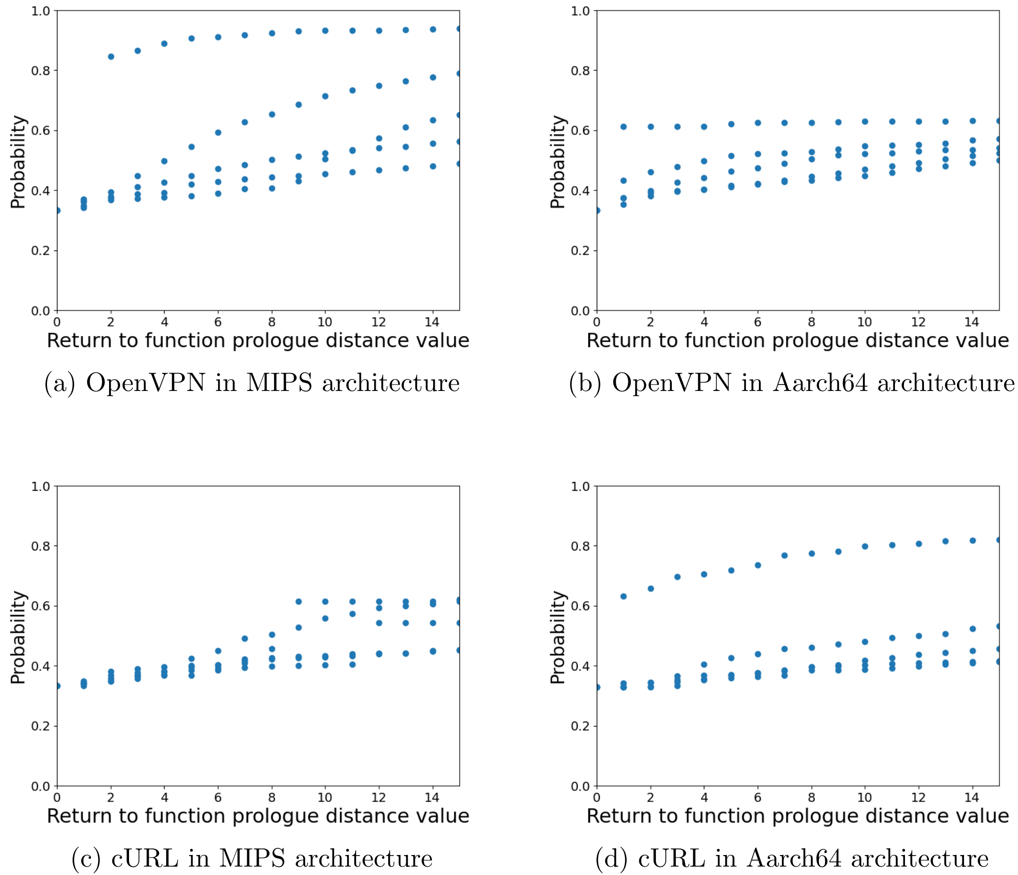
(c) cURL in MIPS architecture



(d) cURL in Aarch64 architecture

Figure 15: OCP-Score for different inputs of the *pcOffset* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.

Figure 16 displays the five highest OCP-Scores corresponding to various values of return to function prologue distance. This value determines how far above a function prologue one can search for a potential return instruction. From the data, we can see that a value of 2 is necessary to correctly detect functions in MIPS, and a value of 1 is sufficient in Aarch64. Values higher than this introduce additional noise in the data, by amplifying the OCP-Score of incorrect opcodes.



(a) OpenVPN in MIPS architecture

(b) OpenVPN in Aarch64 architecture

(c) cURL in MIPS architecture

(d) cURL in Aarch64 architecture

Figure 16: OCP-Score for different inputs of the *returnToFunctionPrologueDistance* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.

4.4 Call graph creation

To illustrate the call graph functionality effectively, a small program is optimal as it allows clear visualization of the distinct function nodes and edges. In the ensuing figures, different versions of a call graph from the Chipquarium program are presented. Figure 17 depicts the call graph derived from inspecting the functions and function calls in the source code. Figure 18 represents the same graph, with the first five functions merged into one, and Figure 19 presents the call graph as generated by the developed program. Both Figure 18 and 19 showcase identical call graphs, albeit rendered via different graph engines. The rationale behind the merging is due to undetected functions, and will be discussed further in Section 5.

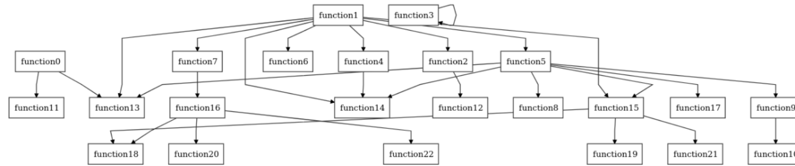


Figure 17: Call graph of the Chipquarium binary hand-crafted from the source code.

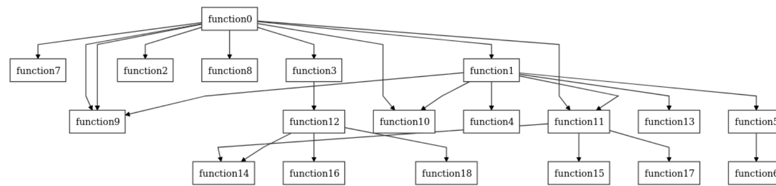


Figure 18: Call graph of the Chipquarium binary hand-crafted from the source code with the first five functions merged into one.

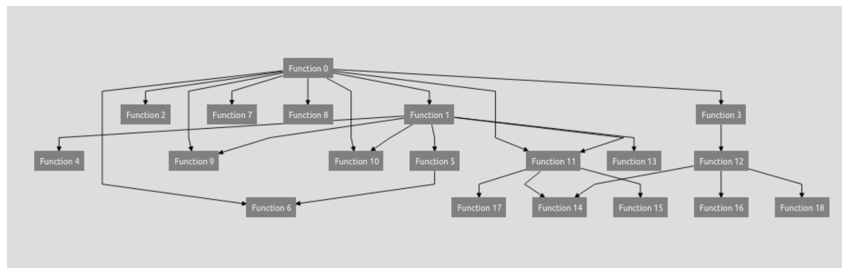


Figure 19: Call graph of the Chipquarium binary as generated by the program.

5 Discussion

Various aspects of the developed program were explored in the preceding analysis, including the viability of the OCP-Score, the accuracy of the opcode detection, and the correctness of the created call graph. In this section, we will discuss these results, seek to answer the research questions and shed light on potential assumptions and limitations.

Starting with the analysis of opcode detection, we can observe that given the binary file has certain properties, such as fixed length instruction format and a significant quantity of return and absolute or relative call instructions, one can effectively distinguish the return and call opcodes from the rest of the instructions. Conversely, a lack of absolute or relative call instructions or a non-fixed length instruction format causes the result to be inconclusive. Therefore, in response to RQ1: it is feasible – given certain properties and parameters – to identify the correct call and return instruction. If the results from the analysed binary are inconclusive, this may also provide valuable insight to the reverse engineer: either the provided parameters are incorrect, or the properties of the binary are not what the program expects, which can guide subsequent analysis.

An interesting observation from the Aarch64 OpenVPN binary in Table 5, was the low frequency of return instruction. However, the program contained a disproportionately high amount of NOP instructions, often found in function epilogues. These instructions have the unique property that they often occur successively, usually more than 3 times. This pattern should make them detectable, and a further improvement to the program could discard them as candidates for call and return instructions, which could further reduce noise.

In order to address RQ2, a thorough analysis of the OCP-Score was conducted to determine the effectiveness and limitations of the approach. This analysis iterated over one of the parameters, examining its sensitivity to noise and its impact on the output. Figure 12 presented how the highest OCP-Score differed with different values for the instruction length parameter. This parameter is unique in that changing its values changes how instructions are extracted, and each value gives a unique output. All values but the correct one generates a list of instructions that is essentially a pseudo-random combination of bits. Out of the 68 total iterations, the OCP-Score was dominant in the four cases where the correct value was chosen for the parameter. This result strengthens the viability and usability of the OCP-Score, indicating that it remains robust against random data.

When iterating over different call opcode lengths, we observe that multiple values resulted in a high OCP-Score. This can be attributed to the fact that the most significant bits of the operand rarely hold information. For instance, for absolute calls and positive relative calls, the most significant bits are usually 0, while for negative relative calls, the value is 1, due to it being a signed integer. An interesting consequence of this is that increasing the call opcode length to a value such as 8 would split the positive and negative relative call instructions into two distinct opcodes, where one of them could have a higher OCP-Score than the correct call opcode with

a length of 6. This is where the use of the program combined with manual inspection would prove useful. An experienced reverse engineer could inspect the instructions and figure out that the value of the operand is a signed integer, and identify the correct call opcode length.

Other parameters such as **returnToFunctionPrologueDistance** seen in figure 16, **callCandidateRange** and **retCandidateRange** require a minimal value to correctly identify the call and return opcodes, but increasing it further would only increase the noise in the resulting output. As an example of this, setting the **returnToFunctionPrologueDistance** parameter to a significantly high value would give the branch instruction as high of a OCP-Score as the call instruction, since the likelihood of there being a return instruction in any of the eg. 1000 instructions preceding the branch target is very high. Increasing the range of the other two parameters also increases the likelihood of noise in the data, due to increased search space.

The rationale for developing the OCP-Score was twofold: to have an ordering of the result and only output the most probable candidates, as well as having a value that can be quickly glanced at by a user. Nonetheless, it is important to be aware of the limitations of the value, and use it in conjunction with a manual inspection of the binary, the outputted call graph, and other analyses, for a better and more complete understanding. For instance, an arbitrary instruction that only occurs a few times, where the presumed operand would target an instruction with a return statement preceding it, would output a very high OCP-Score. However, an experienced user would notice that due to the infrequency of the instruction, it is either not likely to be a call instruction, or at the very least the lack of data points renders it inconclusive.

The final analysis examined the call graphs generated from the Chipquarium binary. The analysis revealed that the generated call graph was identical to the hand-crafted call graph, provided that the first 5 functions were merged into a single function. This illustrates the main limitation the program has with generating call graphs: if a function never gets called, the program will not identify it as a function. There are potential ways to remedy this, as most architectures have a distinct function prologue, often involving stack operations. Assuming the program has accurately identified most of the function prologues, the remaining functions could potentially be identified using techniques such as machine learning.

The analysis demonstrated the viability of the developed program, ranging over multiple architectures and binaries. The program can serve as a useful tool to help users in the process of reverse engineering binaries from unknown instruction set architectures, and fills a much-needed gap in the current research. Despite the effectiveness of the program, it is important to be aware of its limitations and to use it in combination with manual inspection and other techniques, for the best overall results.

6 Conclusion

The primary objective of this research project was the development and evaluation of the framework - consisting of the program, the frontend, and accompanying formulas and theory - aimed at assisting in the reverse engineering of binaries from unknown instruction set architectures. The analysis of the program was thorough, focusing on the key functionalities, including opcode detection and the OCP-Score. The results and discussion revealed promising capabilities of the program, validating its functionality, despite certain limitations.

The main contribution of this report is the developed program, which has shown a high degree of effectiveness when the binary files align with particular properties such as a fixed-length instruction format and the presence of return and absolute or relative call instructions. The accuracy of opcode detection and the robustness of the OCP-Score in dealing with noisy data were notable outcomes of this study.

However, several limitations were also found and discussed, most notably binary file properties that did not align with the program's expectations, as was seen with the x86_64 architecture. Furthermore, it was discussed that an integrated approach, incorporating both automatic processing and manual inspection, is both beneficial and necessary for an optimal result.

Regarding future work, several areas have been identified. Firstly, a method could be developed to detect specific instructions, such as the NOP instructions, which could further reduce noise in the output of the program. Secondly, a rewrite to a more efficient programming language such as Rust or C++ could be beneficial when analysing files of greater size. Another addition that would benefit users would be the option of using a CLI interface rather than the developed API interface. Additionally, it was found that branch instructions were often detected as the second most probable call opcode, and potential enhancement to the program could detect and include information on such branch instructions. The development of a method to identify uncalled functions by searching for distinct prologues and epilogues could enhance the capabilities of the program significantly. Lastly, providing sane defaults and optional parameters for the API, requiring only a subset of the currently required parameters, would greatly improve the usability of the program.

In conclusion, the developed framework has been proven to be a useful tool in reverse engineering binaries from unknown instruction set architectures. While there is room for improvement, the framework addresses a significant gap in the toolset available for such tasks. As research in this field continues, it is expected that more tools focusing on unknown instruction set architectures will be developed.

7 References

[1] M. Fyrbiak, S. Strauss, C. Kison, S. Wallat, M. Elson, N. Rummel, and C. Paar, "Hardware reverse engineering: Overview and open challenges," *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, 2017.

[2] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' process and mental models," *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[3] H. Pettersen, "Towards discovery of program control flow in binary programs from unknown instruction set architectures," 2022.

[4] *The architecture of Open source applications: Elegance, evolution, and a few fearless hacks*, vol. 1. Brown & Wilson, 2011. File: `RetargetableCompiler.png`.

[5] K. P. Singh and S. Parmar, "Design of high performance MIPS cryptography processor based on T-DES algorithm," *CoRR*, vol. abs/1503.03166, 2015. File: `MIPS-instruction-Type.png`.

[6] W. Commons, "Executable and linkable format." https://en.wikipedia.org/wiki/Executable_and_Linkable_Format. File: `ELF-layout--en.svg`.

[7] J. Clemens, "Automatic classification of object code using machine learning," *Digital Investigation*, vol. 14, pp. S156–S162, 2015.

[8] S. Kairajärvi, A. Costin, and T. Hämäläinen, "Isadetect: Usable automated detection of cpu architecture and endianness for executable binary files and object code," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pp. 376–380, 2020.

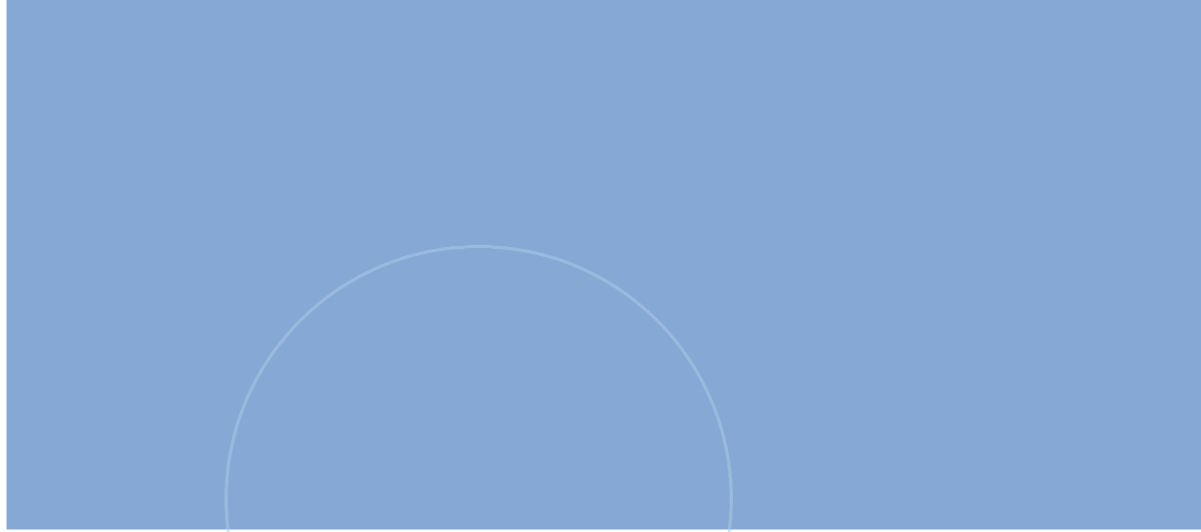
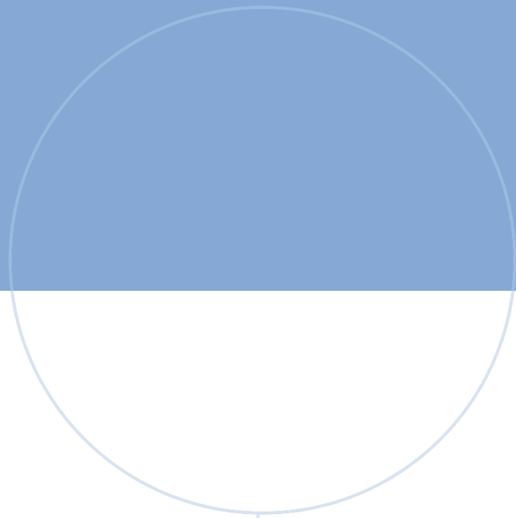
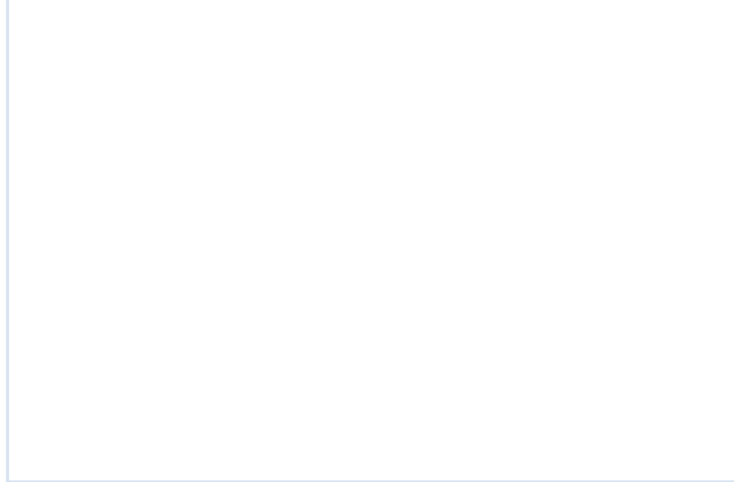
[9] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *2009 30th IEEE Symposium on Security and Privacy*, pp. 94–109, IEEE, 2009.

[10] A. Chernov and K. Troshina, "Reverse engineering of binary programs for custom virtual machines," in *ReCon 2012*, 2012.

[11] K. Beck, *Extreme programming explained : embrace change*. Boston, Mass.: Addison-Wesley, 2004.

[12] "Arm a-profile a64 instruction set architecture." https://developer.arm.com/documentation/ddi0602/2023-03/Base-Instructions/BL--Branch-with-Link-?lang=en.

[13] "Mips reference sheet." https://uweb.engr.arizona.edu/~ece369/Resources/spim/MIPSReference.pdf.

NTNU

Norwegian University of
Science and Technology