EasyChair Preprint

№ 11183

Call Graph Discovery in Binary Programs from Unknown Instruction Set Architectures

Håvard Pettersen and Donn Morrison

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

October 27, 2023

Call graph discovery in binary programs from unknown instruction set architectures

Håvard Pettersen and Donn Morrison

[1] Norwegian University of Science and Technology
Trondheim, Norway
haavapet@stud.ntnu.no
[2] Norwegian University of Science and Technology
Trondheim, Norway
donn.morrison@ntnu.no

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. We focus on the tasks of detecting candidate call and return opcodes for automatic extraction of call graphs in order to simplify the reverse engineering process. Empirical testing on a small dataset of binary files from different architectures demonstrates that the approach can accurately detect specific opcodes under conditions of noisy data. The method lays the groundwork for a valuable tool for reverse engineering where the reverse engineer has minimal a priori knowledge of the underlying instruction set architecture.

Keywords: Reverse engineering · unknown CPU architecture · program call graph · binary analysis

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 process of reverse engineering. Reverse engineering in the context of software, is the practice of inspecting, deconstructing, and analysing 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 priori knowledge about the instruction set architectures of the binary being analysed, which poses limitations and challenges when dealing with unknown or undocumented 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 [7]. 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 [14,9].

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 [13]. 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 a call graph be heuristically deduced from binary programs of an unknown instruction set architecture?
RQ2 How effective is the heuristic approach and what are its limitations?

The central contribution of this study is the development and validation of a method to detect opcodes and generate call graphs from binaries with unknown instruction set architectures. Our method is evaluated in detail, revealing its capabilities and limitations. A secondary contribution is a metric called the Opcode Candidacy Probability Score (OCP-Score). This metric enables the ranking of opcodes based on likelihood, showing the reverse engineer the most probable call-return pairs.

The structure of the rest of the paper is as follows: Section 2 describes the background and related work. Section 3 describes the methodology and proposed algorithm. Section 4 evaluates the proposed approach on a small dataset of binary programs from different instruction set architectures. Section 5 offers a discussion of the results and Section 6 concludes and proposes potential avenues for further research.

2 Background and related work

In this section we briefly introduce the background and most relevant related work.

2.1 Background

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 (e.g., x86_64 architecture). Table 1 illustrates the fixed-width instruction format 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 represented as `0x1234` for big-endian, and `0x3412` for little-endian.

Table 1: Instruction format of the MIPS architecture, illustrating the arithmetic (R-type), immediate (I-type), and jump (J-type) instruction formats [12].

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address/immediate		
J-type	op	target address				
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Compiled programs and firmware are typically stored in a binary file format, for example the Executable and Linkable Format (ELF). This is of interest in our study because a binary file often contains more than just instructions; it also contains data and metadata. In the case of ELF, there are sections and segments of different types of data. In Figure 1, 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.

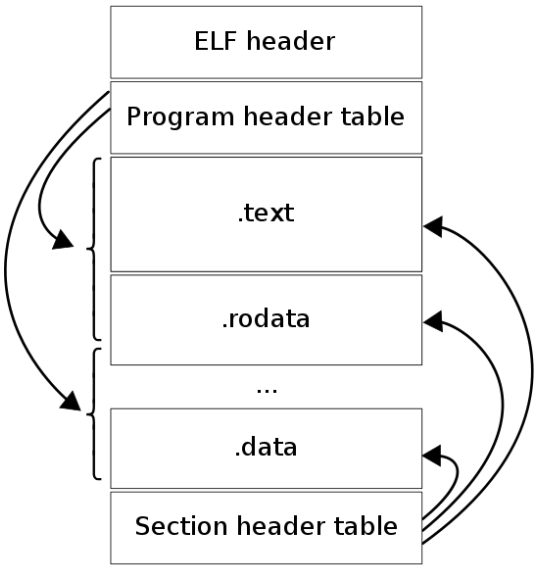
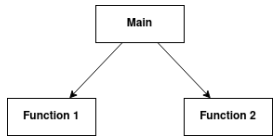


Figure 1: ELF file structure [6].

To detect call graphs in a binary with an unknown instruction set architecture, the most important task is detecting the function boundaries, namely the byte position at which a function starts and ends. It is typical that compiled programs from known architectures exhibit distinct function epilogues and prologues, in the form of 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 2. A more complex call

graph may have characteristics such as cycles, which can be the result of compiled recursion and loops.




```
int main()
{
    Function1();
    ...
    Function2();
}
void Function1() { ... }
void Function2() { ... }
```

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

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.

2.2 Related work

Reverse engineering is an active research field where applications range from malware detection [3] to vulnerability detection [11] to architecture classification [5]. However, nearly all software reverse engineering research assumes a priori knowledge of the instruction set architecture upon which it is executed. Software reverse engineering without knowledge of the instruction set architecture depends first on reverse engineering the hardware, and such research is scarce.

Clemens [5] 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.

Qiu et al. [10] introduce a function representation called the reverse extended control flow graph (RECFG) for function identification that doesn't rely on function prologues and epilogues. They address four key challenges in this approach: 1) difficulty in differentiating data from code when file formats are unknown; 2) sensitivity to disassembly starting points for variable-length instructions; 3) the risk of inaccurate disassembly due to candidate return instructions being part of another instruction; and 4) issues with relying on compiler-specific prologues and epilogues. Their method uses a multilayer perceptron trained with features based on the 32 bytes around a candidate opcode and debug symbols as groundtruth. However, the method requires detailed knowledge of the ISA (the candidate opcodes must be decoded from their instructions).

Sharif *et al.* [11] developed a system called Rotalume to reverse engineer binaries that have been obfuscated using programs such as VMprotect [3]. This approach was however dependent

[3] https://vmpsoft.com/

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.

In an unpublished work by Chernov *et al.* [4], 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 our heuristic approach, as well as dataset acquisition and analysis strategy.

3.1 Call graph extraction

At a high-level, our approach[4] takes a binary file and a set of parameters as input, and returns a list of potential call graphs along ranked by probability. Figure 3 depicts a context where a reverse engineer would use the method as part of the process of reverse engineering a binary towards a high-level representation.

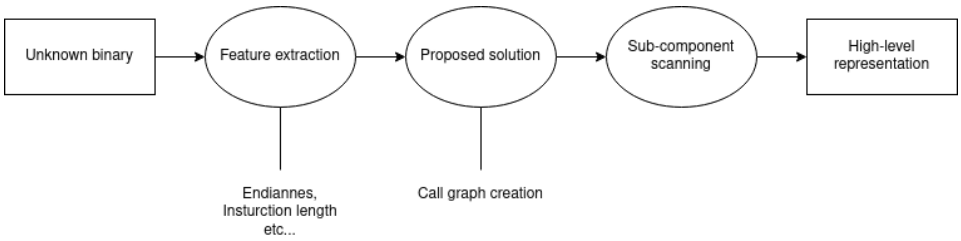


Figure 3: Context of the use of the proposed solution, occurring between architectural feature extraction and sub-component scanning.

A small binary from the Chip8 architecture was used for verification during development of the method due to its instruction format being well-suited for static analysis. This enabled testing under the assumption that properties of the Chip8 ISA were not known, and then checking results against the ISA as a groundtruth. In Sections 4 and 5 we will further analyse and discuss the method against a more common and comprehensive set of architectures.

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

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 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
```

Support for relative addressing was implemented as additional functionality and 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
```

We introduce a metric for computing a probability for a given call opcode and return opcode pair to rank candidates for visual inspection by a reverse engineer. The metric is listed as Equation 1, and is referred to as **Opcode Candidacy Probability Score** (**OCP-Score**).

$$\textbf{OCP-Score}_{op} = \frac{a \cdot (\textbf{valid edges}_{op}) + (\textbf{potential call edges}_{op})}{b \cdot (\textbf{call count}_{op})}, \tag{1}$$

where $a = 2$, $b = 3$, and $\textbf{call count}_{op}$ refers to the number of instructions with the given call opcode *op*. **potential edges** refers to the number of edges associated with the candidate call opcode *op*, in particular, those instances where the operand points to a valid address. **valid edges** refers to the number of edges where there is a function epilogue, specifically a candidate return instruction, located within a specified range before the address of the candidate call instruction containing the opcode *op*.

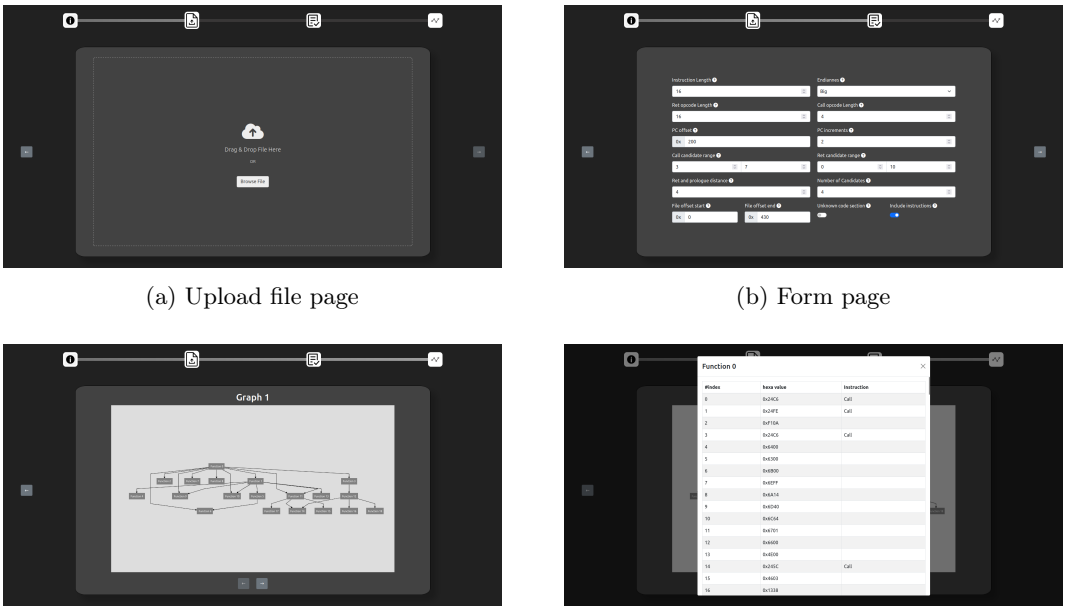
The OCP-Score is normalised within the range $[0, 1]$, explained by the constraint that **length valid edges** and **potential call edges** are always less than or equal to **call count** (numerator and denominator constants a and b). It is worth noting that **valid edges** is weighted more heavily than **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.

To reduce the large opcode search space, the analysis of the binary file requires a set of initial parameters provided alongside the file itself. The parameters, their type, and a description can be seen in Table 2. All parameters are currently required by the API, however, a potential modification with reasonable defaults and increased search space, could require only the first three parameters (instructionLength, retOpcodeLength, and callOpcodeLength) while keeping the rest optional, which would greatly increase usability. The reverse engineer would normally play a role in determining initial values for parameters.

Table 2: 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
returnToFunction-PrologueDistance	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 backend implementation in Python, a frontend written in React was also developed for ease of use. The frontend provides a simple graphical interface where the reverse engineer can upload the binary file, input the required parameters, and then display the created call graphs. Figure 4 shows the interface of the frontend.



(a) Upload file page

(b) Form page

(c) Display graph page

(d) Modal after clicking function 0

Figure 4: User interface of the frontend solution, showing the different pages for uploading a binary file, entering parameters, and displaying the generated call graph.

3.2 Analysis strategy and data generation

The results and analysis focus on three integral parts of the presented heuristic method. The first part is to input the binary 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 our method 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 method 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.

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 3, with the corresponding list of parameters found in Table 4.

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 method.

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 3: 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 4 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 [1][2].

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 4: API parameters used in the analysis.

Parameters \ Binaries	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 5, 6, 7, and 8 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 5 and 8, whereas the remaining two tables reveal contrasting outcomes.

Upon examining the binary in Table 6, 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 7. 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 5: 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 6: 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 7: 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 8: 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 9 and 10, respectively.

Table 9: 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 10: Top 5 most probable return and call opcodes from the cURL binary with x86_64 architecture.

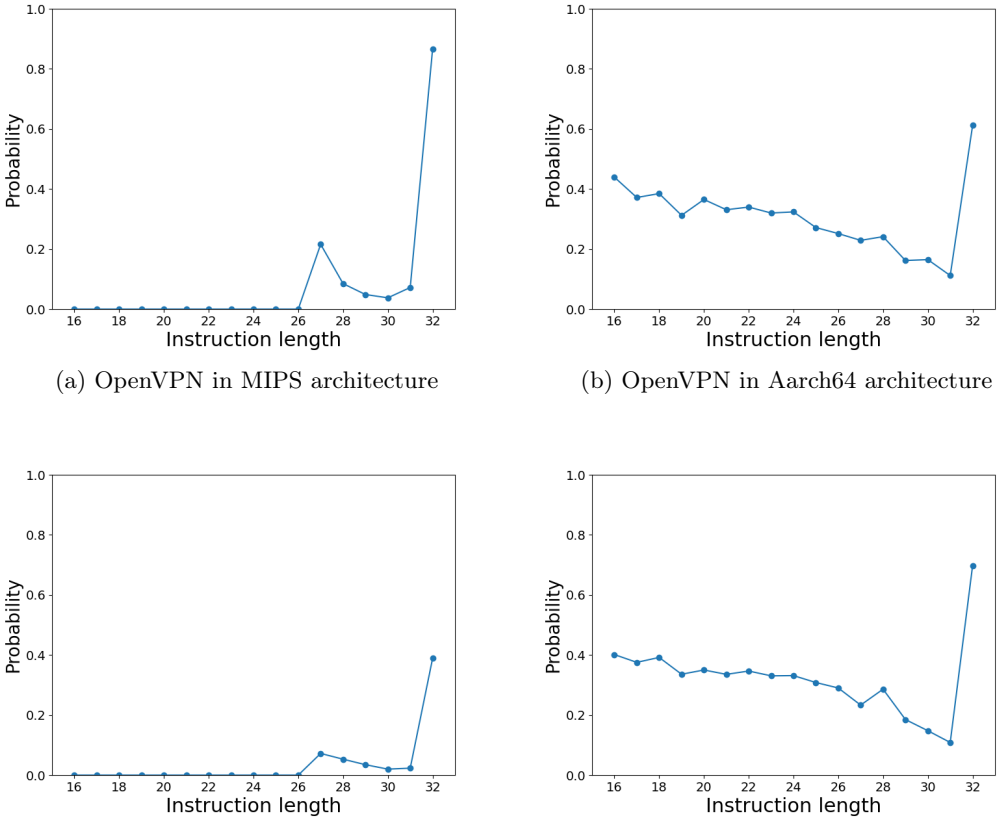
OCP-Score	Call opcode	Return opcode	Correct
0.001	0xF00000000	0x4800000000	
0.001	0xF00000000	0x8B00000000	
0.001	0xF00000000	0xFF00000000	
0.001	0xF00000000	0x2400000000	
0.001	0xF00000000	0x8900000000	

4.3 OCP-Score as a metric

Figure 5 depicts 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 be an address in the range between the first and 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

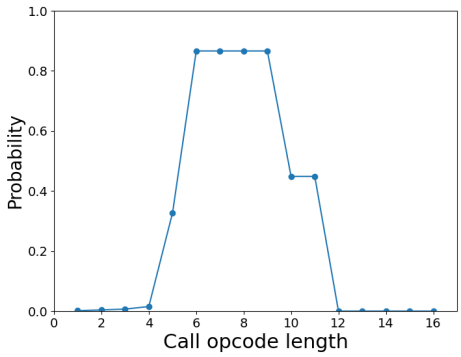
(d) cURL in Aarch64 architecture

Figure 5: OCP-Score for different inputs of the *instructionLength* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.

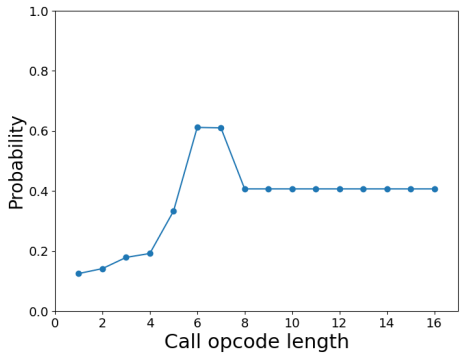
Figure 6 depicts the maximum OCP-Score corresponding to various values of call opcode length. The data suggests that multiple values close to the correct value give a high OCP-Score. The explanation for this is presented in Section 5.

Figure 7 depicts 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.

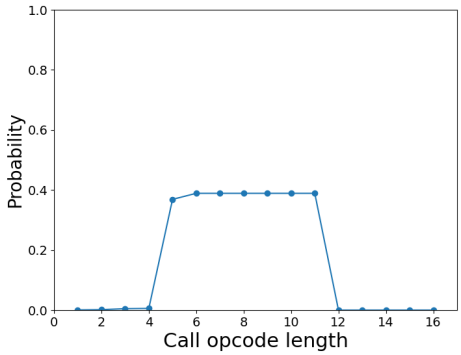
Figure 8 depicts the maximum OCP-Score corresponding to various values of PC offset. It is important to clarify that these values do not affect the particular instructions read from the binary file, but rather assign a specific address to each instruction. For example, with a PC offset value of `0x1000`, 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,



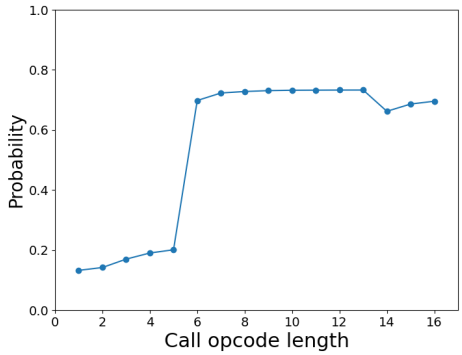
(a) OpenVPN in MIPS architecture



(b) OpenVPN in Aarch64 architecture

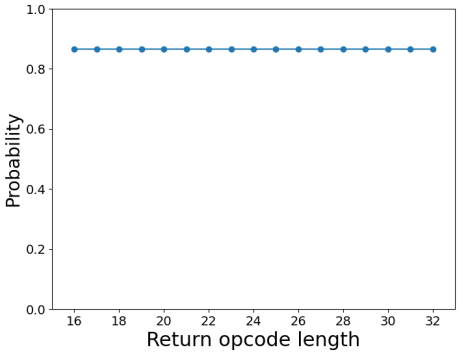


(c) cURL in MIPS architecture

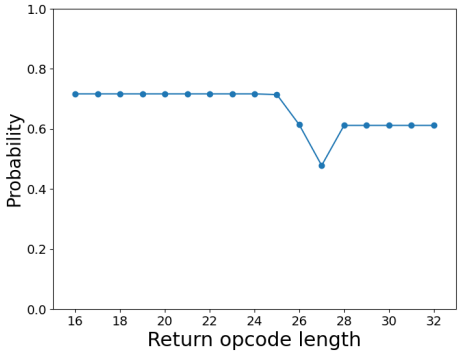


(d) cURL in Aarch64 architecture

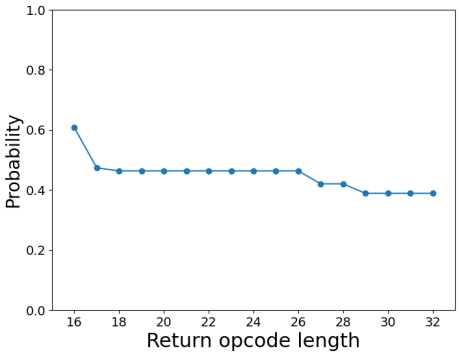
Figure 6: OCP-Score for different inputs of the *callOpcodeLength* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.



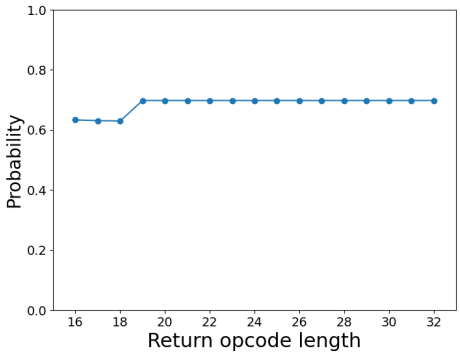
(a) OpenVPN in MIPS architecture



(b) OpenVPN in Aarch64 architecture



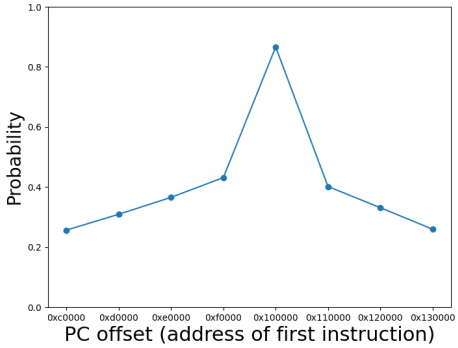
(c) cURL in MIPS architecture



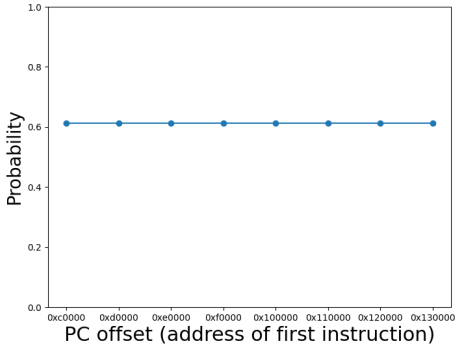
(d) cURL in Aarch64 architecture

Figure 7: OCP-Score for different inputs of the *retOpcodeLength* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.

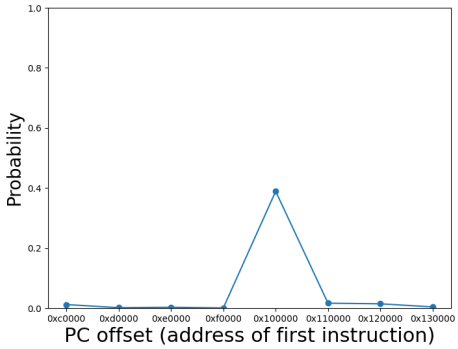
which aligns with expectations. However, in the context of absolute addressing in MIPS, the correct value gives a significantly higher OCP-Score (subfigures (a) and (c)).



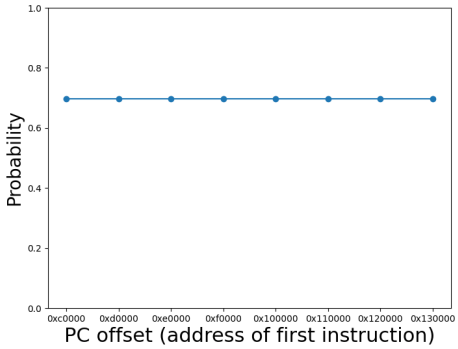
(a) OpenVPN in MIPS architecture



(b) OpenVPN in Aarch64 architecture



(c) cURL in MIPS architecture



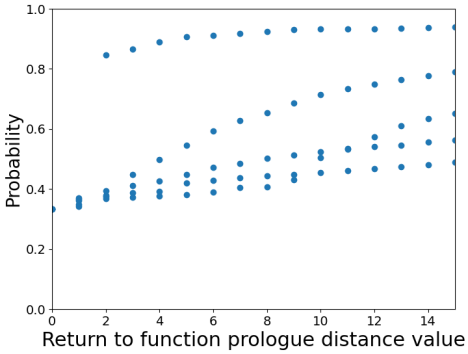
(d) cURL in Aarch64 architecture

Figure 8: OCP-Score for different inputs of the *pcOffset* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.

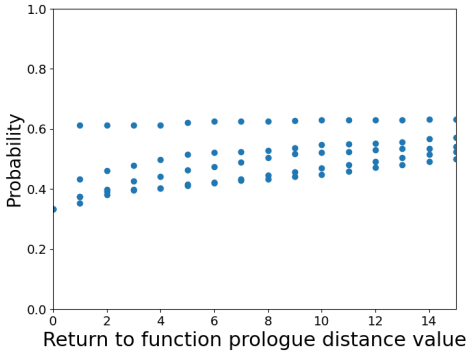
Figure 9 depicts the five highest OCP-Scores corresponding to various values of return to function prologue distance. This value determines how far above a function prologue one can search for a potential return instruction. From the data, we can see that a value of 2 is necessary to correctly detect 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.

4.4 Call graph creation

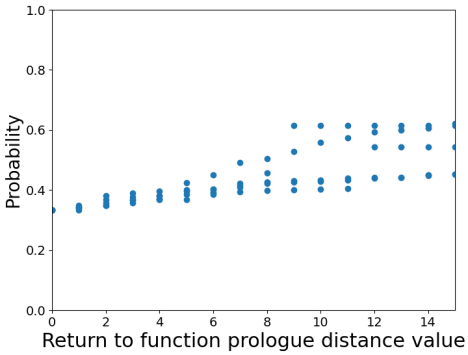
To illustrate the call graph functionality effectively, a small program is optimal as it allows clear visualization of the distinct function nodes and edges. In the ensuing figures, different versions of a call graph from the Chipquarium program are presented. Figure 10 depicts the call graph derived from inspecting the functions and function calls in the source code. Figure 11 represents the same graph, with the first five functions merged into one, and Figure 12 presents the call graph as generated by the developed program. Both Figure 11 and 12 showcase identical call graphs, albeit rendered via different graph engines. The rationale



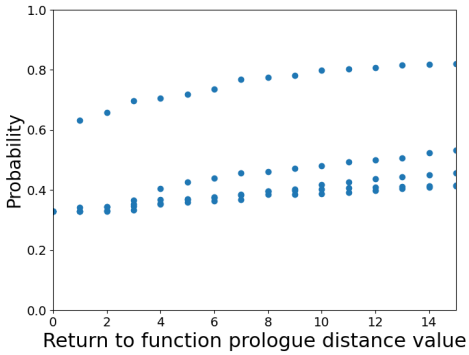
(a) OpenVPN in MIPS architecture



(b) OpenVPN in Aarch64 architecture



(c) cURL in MIPS architecture



(d) cURL in Aarch64 architecture

Figure 9: OCP-Score for different inputs of the *returnToFunctionPrologueDistance* parameter, shown for the cURL and OpenVPN binaries in the MIPS and Aarch64 architectures.

behind the merging is due to undetected functions, and will be discussed further in Section 5.

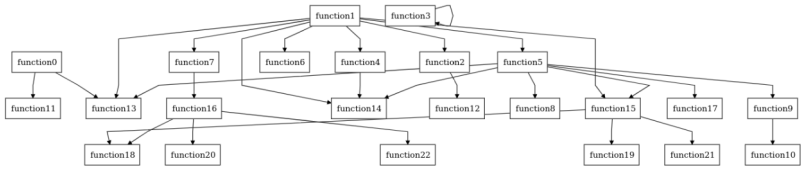


Figure 10: Call graph of the Chipquarium binary hand-crafted from the source code.

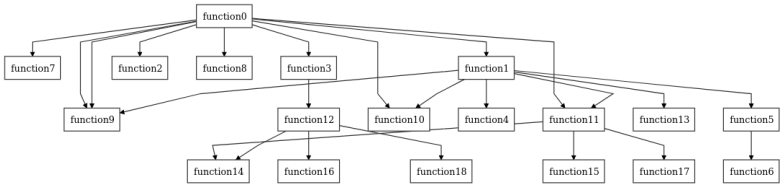


Figure 11: Call graph of the Chipquarium binary hand-crafted from the source code with the first five functions merged into one.

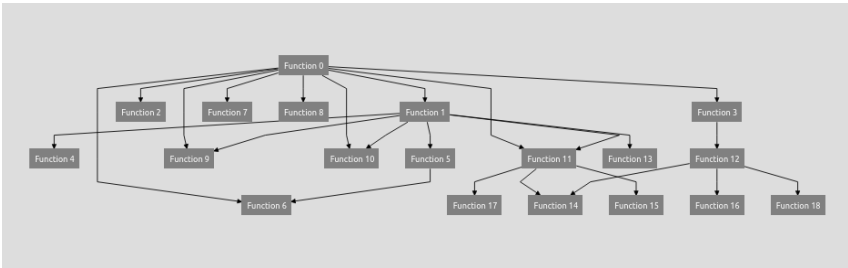


Figure 12: Call graph of the Chipquarium binary as generated by our approach.

5 Discussion

The results presented in the previous section illustrate the application 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 and how they address the research questions.

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 is expected, which can guide subsequent analysis.

An interesting observation from the Aarch64 OpenVPN binary in Table 6, 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 heuristic approach described in this paper could be to discard them as candidates for call and return instructions, reducing noise in the resulting ranking.

In order to address RQ2, an analysis of the OCP-Score was conducted to determine the effectiveness and limitations of the approach. This analysis iterated over a selection of parameters examining its sensitivity to noise and its impact on the output. Figure 5 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 approach combined with manual inspection would prove useful. An experienced reverse engineer could inspect the instructions and deduce 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 9, **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 high value would give the branch instruction an OCP-Score similar to the call instruction, since the likelihood of there being a return instruction in any of the in-scope 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: 1) to present an intuitive ranking of the most probable candidates, and 2) to have a simple scalar value that can be quickly referenced by the reverse engineer. 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 our approach has with generating call graphs: if a function never gets called, the heuristic apparoch 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 approach has accurately identified most of the function prologues, the remaining functions could potentially be identified using techniques such as machine learning or pattern matching.

The results and analysis over multiple architectures and binaries demonstrates the effectiveness of the presented heuristic approach. We are confident that it can serve as a useful tool to help reverse engineers in the process of analysing binary programs from unknown instruction set architectures, and fills a much-needed gap in the current research. Despite the effectiveness of the heuristic approach, 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 was aimed at reducing the effort of reverse engineering binaries from unknown instruction set architectures. The results and discussion focused on evaluating key properties of the heuristic approach including opcode detection and the OCP-Score.

The approach is effective 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.z

However, several limitations were also found and discussed, most notably that variable-length instructions are not supported as 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. First, support for variable-length instructions would enable the method to support a wider variety of unknown ISAs. Next, functionality to detect specific instructions such as NOPs could further reduce noise. Additionally, it was found that branch instructions were often detected as the second most probable call opcode, and a potential improvement would be to detect and include information on such branch instructions. Identification of uncalled functions via prologues/epilogues matching would improve binary code coverage. Lastly, evaluation on a larger dataset of binary programs would help generate a clearer picture of performance over a wider variety of ISAs.

References

1. Arm a-profile a64 instruction set architecture. https://developer.arm.com/documentation/ddi0602/2023-03/Base-Instructions/BL--Branch-with-Link-?lang=en
2. Mips reference sheet. https://uweb.engr.arizona.edu/~ece369/Resources/spim/MIPSReference.pdf

3. Bao, C., Forte, D., Srivastava, A.: On application of one-class svm to reverse engineering-based hardware trojan detection. In: Fifteenth International Symposium on Quality Electronic Design. pp. 47–54. IEEE (2014)
4. Chernov, A., Troshina, K.: Reverse engineering of binary programs for custom virtual machines. In: ReCon 2012 (2012), https://recon.cx/2012/schedule/attachments/40_Chernov-Troshina.pdf
5. Clemens, J.: Automatic classification of object code using machine learning. Digital Investigation **14**, S156–S162 (2015)
6. Commons, W.: Executable and linkable format. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, file: `ELF-layout-en.svg`
7. Fyrbiak, M., Strauss, S., Kison, C., Wallat, S., Elson, M., Rummel, N., Paar, C.: Hardware reverse engineering: Overview and open challenges. 2017 IEEE 2nd International Verification and Security Workshop (IVSW) (2017). https://doi.org/10.1109/ivsw.2017.8031550
8. Kairajärvi, S., Costin, A., Hämäläinen, T.: 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. Kinder, J.: Towards static analysis of virtualization-obfuscated binaries. In: 2012 19th Working Conference on Reverse Engineering. pp. 61–70. IEEE (2012)
10. Qiu, J., Su, X., Ma, P.: Identifying functions in binary code with reverse extended control flow graphs. Journal of Software: Evolution and Process **27**(10), 793–820 (2015)
11. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: 2009 30th IEEE Symposium on Security and Privacy. pp. 94–109. IEEE (2009)
12. Singh, K.P., Parmar, S.: Design of high performance MIPS cryptography processor based on T-DES algorithm. CoRR **abs/1503.03166** (2015), file: `MIPS-instruction-Type.png`
13. Votipka, D., Rabin, S., Micinski, K., Foster, J.S., Mazurek, M.L.: An observational investigation of reverse engineers' process and mental models. Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems (2019). https://doi.org/10.1145/3290607.3313040
14. Xu, D., Ming, J., Fu, Y., Wu, D.: VMHunt: A verifiable approach to partially-virtualized binary code simplification. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 442–458 (2018)