Detecting malware capabilities in superblocks using static binary analysis

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Information and Communication Technology
June 2025
Yacine Elhamer

Supervisors:
Antti Hakkala
Tahir Mohammad

UNIVERSITY OF TURKU
Department of Computing

YACINE ELHAMER: Detecting malware capabilities in superblocks using static binary analysis

Master of Science (Tech) Thesis, 69 p.
Information and Communication Technology
June 2025

Malware, or malicious software, is a central piece to the success of most cyber attacks since often times it is the element that is programmed and used to carry out the objectives of the threat actors once they gain access to a system. This has made it vital for cybersecurity analysts to study malware and accurately categorize it, so that they remain a step ahead of the attackers, and are able to respond quickly in the case of time-sensitive cyber breach situations. The large volume of malware that has created a necessity for tools that automate the process of analyzing new malware, and especially tools aimed at extracting the behavior and capabilities of the encountered malware. One of the most widely used tools for this purpose is the *capa* tool by Mandiant's FLARE team.

This thesis aims to improve FLARE's capa malware capability extraction tool by increasing the accuracy of its capability extractions and categorization. Currently, it extract features from instructions, basic blocks, and functions, and then matches them with the rules that apply to each individual scope. This extraction method presents an issue of false negatives if a rule is written to be applied on basic blocks, but the features to-be-matched end up being split into two or more basic blocks by the compiler (in the case of the malware author adding error-checking code between two related malicious API calls for instance); as well as potential false positives if the previous rule's matching scope is set to function instead (to circumvent the stated limitation), which might lead to completely unrelated features being paired up together in the case of large functions. The identified limitation is relevant because usage of if-else statements for error checking are common in software in general and malware specifically, and those are a major reason why semantically-related code would end up being segmented into multiple basic blocks.

The work adds a new "superblock" matching scope to capa that makes it possible to extract malware capabilities whose constituent features are spread across a number of basic blocks in direct sequence, therefore reducing the possibility of false positives and false negatives when matching some rules, as previously described.

Using the newly-introduced scope, we were able to rewrite an existing capa rule intended for detecting RC4 encryption logic, and have it match the relevant logic on the superblock scope as opposed to the old function scope. By doing so, we were able to retain the tool's true positive rate, while eliminating some observed false positive cases such as the *wmemcmp()* buffer comparison method from the C runtime being incorrectly identified as performing RC4 encryption.

Keywords: malware analysis, static binary analysis, security automation, capa

Contents

List of Figures

List of acronyms

API Application Programming Interface

ATT&CK Adversarial Tactics, Techniques, and Common Knowledge

C2 Command & Control

CIL Common Intermediate Language

CKC Cyber Kill Chain

CPU Central Processing Unit

CSP Call Signatures Plugin

ELF Executable and Linkable format

F.L.I.R.T Fast Library Identification and Recognition Technology

FP False Positive

GUI Graphical User Interface

IC3 Internet Crime Complaint Center

IDA Interactive Disassembler

JSON JavaScript Object Notation

Mach − O Mach Object

MBC Malware Behavior Catalog

$MD5$ Message Digest 5

PE Portable Executable

$PRGA$ Pseudo-Random Generation Algorithm

$RC4$ Rivest Cipher 4

$SBox$ Substitution Box

TP True Positive

TTP Techniques, Tactics, and Procedures

$YAML$ Yet Another Markup Language

$YARA$ Yet Another Recursive Accronym

1 Introduction

The advent of computers and complex communication networks, such as the Internet, has caused a large shift toward digitalization by people, organizations, and governments alike in different sectors. This shift is largely influenced by the increased efficiency and effectiveness that digital systems provide as opposed to purely manual ones. Unfortunately, however, these benefits come at the cost of exposing more potentially vulnerable aspects of systems to criminals, who were also among the groups that adopted the cyberspace as a stage of operation.

According to the 2023 Internet Crime Report published by the Internet Crime Complaint Center (IC3) [1], which is a division of the Federal Bureau of Investigation of the United States, exactly 880,418 Internet-related complaints were received from US citizens and nationals of other countries, with more than 12.5 billion dollars lost to criminals in that year alone. This figure is 80,000 higher than the previous year and continues a general rising trend for the frequency of cyber crime incidents.

As a consequence of the increased risk of falling victim to a cyber attack, organizations and individuals alike increased their investments in security countermeasures and products, with the aim of minimizing the threats facing their now-digital systems. This has caused many frameworks and models to emerge that attempt to help defenders stay ahead of attackers. One of these frameworks that will prove helpful in determining a key recurring aspect of cyber attacks (and the topic of this thesis) is Lockheed Martin's Cyber Kill Chain (CKC).

1.1 Cyber Kill Chain

The Cyber Kill Chain is a framework developed by the American defense and aerospace manufacturing company Lockheed Martin, and its goal is to assist defenders in identifying and preventing digital compromise [2]. The model is made up of seven essential steps that Lockheed Martin asserts must all be completed in sequence by an attacker in order for them to fulfill their objectives. These steps are as follows:

1. **Reconnaissance:** The threat actors conduct research to identify their targets as well as potential points of interest.

2. **Weaponization:** The threat actors combine first stage malware, exploits, as well as potential decoy files (such as Microsoft Office documents in the case of Phishing attacks) into a deliverable payload.

3. **Delivery:** The attackers deliver the previously crafted payload to the target using physical methods (such as malicious USB sticks) or digital methods (such as phishing emails).

4. **Exploitation:** At this stage, attackers must exploit a component of the system to gain access. This component could either be human or digital (such as a software vulnerability or misconfiguration).

5. **Installation:** The attackers installs malware on the target, and then attempts to establish persistence, backdoors, as well as attempting to hide their traces by wiping out the logs or changing the creation dates for files that they created (also known as a "timestomping attack").

6. **Command & Control (C2):** The malware that was installed in the previous stage connects back to the attacker's command and control server in order to exfiltrate files or receive further commands.

7. **Actions on Objectives:** Once the attackers achieve direct access to the target's system, they then proceed to carry out their objectives

By examining the kill chain detailed above, we can identify and extract elements that are shared by all cyber attacks. Then, by focusing on the least varying of these elements we can devise strategies and create security controls that could then be shared to and implemented by different organizations without requiring any major tweaking or modification these controls. One of the best candidates for such an element is malware.

1.2 Malware and Malware Analysis

Malware, which is an abbreviated term for Malicious Software, can be defined as any software that performs malicious actions on a user, computer, or network, including viruses, Trojan horses, worms, rootkits, ransomware, spyware, among others [3]. Typically, malware gets shared and reused a lot among threat actors, who often modify it slightly to fit the needs of the ongoing attack, and then deploy it. Malware samples with similar origins and traits are classified into families, with examples of common malware families including: Cobalt Strike, LockBit, and Metasploit [4].

According to the document published by Lockheed Martin titled "Gaining The Advantage: Applying Cyber Kill Chain Methodology to Network Defense" [2], which is aimed at defenders aiming to thwart cyber attacks, it is sufficient to prevent only one step of the kill chain's seven steps in order to stop the entire attack from reaching its objects. Additionally, the guide highlights malware analysis as a key task in order to detect and prevent the second stage (weaponization) and fifth stage (installation) of an attack. This is because effective analysis of the malware used can help defenders create effective mitigations for future encounters with the same novel piece of malware, as well as gain an idea about the actors behind the attack

and what their motives or attribution might be. However, malware analysis can be a difficult and complex task as we will see later.

1.3 Automating Malware Analysis

According to the 2024 M-Trends report published by Mandiant [4], which is an American cybersecurity firm and a subsidiary of Google, the company was able to track 626 new malware families in that year alone, with 128 of them having been observed by the firm's incident response team during engagements with clients. This figure represents an increase from 588 newly-tracked malware families in the year prior, and continues a general upwards trend with this regard. As a consequence, the rising number of unique malware families in the wild, as well as the large number of samples that can result from each individual family, conveys a need for developing and improving tools for automating and making the malware analysis process less time consuming and more straightforward to analysts in order to keep up with threat actors. One category of malware analysis automation tools that is seeing a rise in usage and efficiency is malware capability extractors such as capa [5] and Forecast [6].

Malware capability extraction tools typically operate on a simple shared principle: the tool receives a malicious sample either in binary form or as the output of a sandbox run, and tries to look for malware capabilities that are described by specific rules that are either written out (in the case of capa for instance) or learned (in the case of AI-based tools). One of the most widely adopted tools among malware analysts is *capa*, which was created by Mandiant's FLARE team, and has the ability to extract varying types of binary capabilities and alert the analyst to their location in the executable or library.

The way capa operates is that it takes in a binary and then extracts all of its constituent functions using a static analysis backend such as Mandiant FLARE's

vivisect framework, then, extract all of the basic blocks that make up each function, and finally, the instructions that make up each basic block. After that, the tool relies on a set of rules crafted by other security researchers that define malware capabilities in terms of smaller constituent instruction, basic block, or function features. Each rule has a scope at which it is meant to be applied, and at the matching stage capa reads each rule matching conditions as well as that rule's specified scope (i.e. function, basic block, or instruction), and then try to see if any of the binary's components of that scope's type satisfy the listed conditions. If there is a match, then the malware capability is deemed as present in the binary.

1.4 Research Questions

Although capa's current architecture functions well for extracting some kinds of malware capabilities, a room for improvement has been identified in order to minimize some of the tool's false positives and false negatives. This is due to a present flaw in the way capa currently performs capability matching, which is that rules made to match at the basic block scope might not trigger since a control flow statement might break what would have been a single basic block into two or more blocks (such as in the case of if-else statements being used for error checking) thereby triggering a false negative, and also because rules made to circumvent this issue by matching at the function scope instead could trigger false positives in cases where a function could be too large and thus unrelated binary artifacts (instructions, API calls, etc.) could be falsely correlated together.

As a result, we identify the following research questions for our thesis:

1. RQ1: *What are the most adopted tools for extracting malware capabilities? And does any of them solve the accuracy issue that capa suffers from?*

2. RQ2: *Can adding a new analysis scope that provides analysts with the ability*

to match malware capability logic at the level of basic blocks in close sequence improve the accuracy of capa's results? And will it increase the expressibility of capa rules?

3. RQ3: *Are there any existing capa rules whose associated accuracy could be immediately improved by changing their scope from function or basic block to superblock?*

1.5 Research Objectives

This thesis enhances capa's current capability extraction logic by introducing a new rule-matching scope named the "superblock" scope, which sits between the function and the basic block scope. This scope correlates basic blocks in succession with features that could result in a rule matches, thereby helping to reduce the rate of false negatives and false positives in the tool's output that are due to error checking logic or other regular control-flow constructs common among malicious and non-malicious software developers.

Our main objectives for this work are:

1. RO1: *Examine the main malware capability extraction tools, and whether any of them solves the identified issue that capa suffers from.*

2. RO2: *Design and implement a superblock scope for the capa tool, and showcase how it increases the expressibility of malware capabilities for security analysts authoring capa rules.*

3. RO3: *Find an example of an existing capa rule with the function or basic block scope, and show how its accuracy can be improved by using the newly-introduced superblock scope.*

Obfuscated malware samples are out of the scope of the thesis, since static binary analysis is usually not effective on them due to the obscured logic that is typically harder to decipher using static analysis tools, with sandboxes or manual deobfuscation being used first instead. Additionally, most of the malware capability extraction tools we examined were not able to handle such samples, with the capa tool for instance having specific hard-coded logic in it that determines at first whether a sample is packed or obfuscated at first, and halts the analysis entirely if that is the case.

1.6 Thesis Structure

The rest of the thesis is organized as follows: Chapter 2 examines some common malware capability extraction tools, the advantages and disadvantages of each one, and why capa was picked for this thesis project. Chapter 3 is a semi-structured literary review of static binary analysis of superblocks. Chapter 4 gives some background into the architecture of the target system which will be improved in this thesis, which is the capa tool. Chapter 5 details the conceptual design and specifications for the new scope that will be contributed to capa. Chapter 6 details the implementation of the scope into the tool, as well as the verification and analysis of results. Chapter 7 is the conclusion to the thesis, which recaps the contributions made as well as possible future work.

2 Malware Capability Extraction Tools

In this chapter, we will go over a set of existing tools and frameworks that are commonly used by analysts for malware capability extraction, while mentioning the advantages and disadvantages of each tool, and motivating in the process our choice for the target system that we will be integrating our contribution in this thesis into.

We will begin by first describing the YARA rule matcher in the first section, while providing example rules as well as highlighting the tool's advantaged and disadvantages. Then, we will detail the capa malware extraction capability and what its rules look like in the second section, as well as the tool's advantages and disadvantages. After that, we will go through the Call Signatures Plugin (CSP) tool and how it works, as well as its advantages and disadvantages. Finally, we will briefly compare the different tools and motivate our choice for the target system of this thesis in the fourth section of this chapter.

2.1 YARA

YARA (Yet Another Recursive Acronym) is a signature-based malware detection and classification tool developed by VirusTotal [7]. Its operating principle is that malware analysts write YARA rules describing different signatures for previously analyzed malware, then, YARA takes those rules and scans a set of file samples for

potential signature matches.

Unlike simple byte and sequence matching tools (such as Linux's "grep" utility, for example), YARA offers analysts greater flexibility by supporting boolean logic expressions for defining signatures instead of just simple constant byte sequences. Additionally, YARA also gives analysts the ability to extend its functionality by integrating it with custom written modules, with YARA already having built-in support for a handful of modules offering functionalities such as hash functions, mathematical functions, and file-format parsing functions, among others [8].

YARA rules are composed of 3 main sections:

- **meta:** contains author-defined key and value pairs that denote metadata about the rule. Examples of such metadata could be: the author's name, threat classification, malware family's first appearance date.

- **strings:** contains author-defined strings and byte sequences that are to be referenced later by the "condition" section described below. This section can be viewed as the "variables declaration section" for YARA rules.

- **condition:** contains the boolean algebra expression that denotes the condition that must be met by the input file in order for the rule to be matched.

An example YARA rule that showcases its syntax can be seen in Listing 1.

Listing 1 Example YARA rule for deteciting the Zeus malware family

```
import "pe"

rule Windows_Malware_Zeus : Zeus_1134

{

  meta:

    author = "Xylitol xylitol@malwareint.com"

    date = "2014-03-03"

    description = "Match first two bytes, protocol and string present

    in Zeus 1.1.3.4"

    reference = "http://www.xylibox.com/2014/03/zeus-1134.html"

  strings:

    $mz = {4D 5A}

    $protocol1 = "X_ID: "

    $protocol2 = "X_OS: "

    $protocol3 = "X_BV: "

    $stringR1 = "InitializeSecurityDescriptor"

    $stringR2 = "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SV1)"

  condition:

      ($mz at 0 and all of ($protocol*) and ($stringR1 or $stringR2))

}
```

The rule shown in Listing 1 was taken from the Yara-Rules project, which is a GitHub repository that contains numerous YARA rules under the GNU-GPLv2 license for malware analysts to share and contribute to [9].

Although YARA was initially developed to help identify variants of previously analyzed malware and malware families, the general design of the tool also allows one to extract and identify previously observed malware tactics, techniques, and procedures (TTP). This is particularly useful for detecting novel malware that has

not been previously encountered but employs capabilities that are known to be common in malware.

For example, cryptographic functions often rely on specific constants in their functioning such as Rijndael's SBox [10], MD5's initial state [11], and Blowfish's usage of the binary representation of $\pi - 3$ to initialize its key schedule [12]. By treating these constants as signatures, malware authors can detect the usage of such cryptographic functions in malware.

Additionally, many other malware techniques rely on constants that could be used as detection signatures. For example, presence of the following registry path is indicative of the binary potentially employing registry-based persistence, and can therefore be used as a YARA signature [13]:
"HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run".

Another avenue for extracting capabilities from malware with YARA is converting the CPU instructions composing a capability into their binary representation, and then setting that representation as our signature. For instance, the YARA rule depicted in Listing 2 does just that in order to detect usage of the "Heaven's Gate" technique, which allows malware authors to run 64-bit code inside 32-bit processes, thereby evading some security monitoring software [14].

Listing 2 Example YARA rule for detecting Heaven's Gate

```
rule HeavensGate
{
  meta:
    author = "Andrew Williams (@recvfrom) @ Cisco Talos"
    description = "Look for instructions associated with Heaven's Gate"
    date = "2020-10-27"
  strings:
    /*
    *  6A 33             push    33h ; '3'
    *  E8 00 00 00 00    call    $+5
    *  83 04 24 05       add     dword ptr [esp+0], 5
    *  CB                retf
    */
    $retf_to_64bit_mode = {6a33e80000000083042405cb}
    /*
    *  E8 00 00 00 00             call    $+5
    *  C7 44 24 04 23 00 00 00    mov     dword ptr [rsp+4], 23h ; '#'
    *  83 04 24 0D                add     dword ptr [rsp], 0Dh
    *  CB                         retf
    */
    $retf_to_32bit_mode = {e800000000c7442404230000008304240dcb}
  condition:
    (uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550) and
    $retf_to_64bit_mode or $retf_to_32bit_mode
}
```

While YARA's rudimentary nature brings forth many benefits to malware analysts such as the speed of execution and signature matching, its lack of support for many higher-level constructs and abstractions such as functions, basic blocks, and instructions makes it harder for analysts to write and maintain rules for specific malware capabilities.

If a rule author wants to write a YARA rule describing a specific binary capability, they would need to lookup the individual opcodes for each CPU instruction that makes up such a capability, and then construct their YARA rule with that. Additionally, they would need to also be wary of the difference in opcodes between x86 and x64 architectures, as well as the fact that the same capability could have slightly different machine code depending on the compiler used. Furthermore, rule authors would also need to make sure that their rules are being matched in the right section of the binary (i.e., the .text section), and not in the data or overlay section of the binary; failure to do so could result in false positives if the rule's signature bytes happen to occur inside an image stored in the executable's resources section.

The conditions mentioned above can theoretically be handled by malware analysts, but in reality however, they would end up consuming a lot of the analysts' time. Perhaps a fitting comparison might be that YARA to malware capability extraction is like assembly to computer programming, with the former of the two being somewhat more dire and demanding given the frequency at which new malware is encountered by analysts [4].

2.2 Capa

capa is a more advanced malware capability extraction tool and framework developed by the FLARE team from the Mandiant cybersecurity firm (now a subsidiary of Google).

The tool's general idea is similar to that of YARA in the sense that analyst-written rules are used to find matches in input samples. However, unlike YARA which was made with the goal of detecting previously-seen samples and malware families, capa's goal is to detect previously-seen malware capabilities and TTPs. This makes capa's rules more general and more useful to malware analysts given that old and novel malware often times reuses the same TTPs and capabilities, making detection using capa much more effective than YARA.

What distinguishes capa from YARA and makes it more suitable for the task of capability extraction is that it offers malware analysts higher abstraction layers that they could reason at, such as mnemonic instructions, basic blocks, and functions. As a result, instead of analysts concerning themselves with the mnemonic-to-opcode translations of different CPU instructions, functions-extraction from executables, or file format parsing, they would focus solely on the overall picture and leave all of those details to be handled by capa under the hood.

Additionally, it should be noted as well that capa supports all of the features offered by YARA such as bytes and string matching, at different abstraction levels (function, basic block, and instruction). However, this comes at a reduced execution time since capa and its dependencies are written purely in Python, while YARA is written in C.

An example capa rule for detecting memory region allocation or modification can be seen in Listing 3. The rule was taken from the official capa rules GitHub repository [15].

Listing 3 Example capa rule for detecting RW memory allocation or changing

```
rule:
  meta:
    name: allocate or change RW memory
    authors:
      - 0x534a@mailbox.org
      - "@mr-tz"
    lib: true
    scopes:
      static: basic block
      dynamic: call
    mbc:
      - Memory::Allocate Memory [C0007]
    examples:
      - Practical Malware Analysis Lab 17-02.dll_:0x1000D10D
  features:
    - and:
      - or:
        - match: allocate memory
        - match: change memory protection
      - or:
        - number: 0x4 = PAGE_READWRITE
        # lea     r9d, [rcx+4]  ; flProtect
        # call    cs:VirtualAlloc
        - instruction:
          - mnemonic: lea
          - offset: 0x4 = PAGE_READWRITE
```

As can be seen in Listing 3, capa rules are based on the YAML (Yet Another Markup Language) data serialization language [16], and they are made of two main sections: meta and features.

The meta section defines metadata for the rule such as its name, its static and dynamic matching scopes, as well as other fields such as the rule's authors and examples from the capa-testfiles GitHub repository [17] where the capability can be observed.

As for the features section, it contains a boolean algebra expression that is used by capa for matching the rule on sample files. Non-leaf nodes represent boolean algebra and capa-specific operators such as subscope specifiers (like "instruction" in this example), while leaf nodes specify the features to be matched.

When capa is executed with that rule and a sample malware file passed onto it, the tool first performs a deep analysis of the malware and extracts all of its constituting functions, as well as each function's constituting basic blocks, and the instructions that make up each basic block. Then, capa parses that rule and generates an internal binary expression tree for it. Finally, capa's matching engine keeps only the leaf nodes whose representative feature was found in the malware sample, and does not consider any leaf nodes that were not found in the malware sample; then, the binary expression tree is evaluated, and if is found to be True, then the capability is marked as "found" and is returned to the user, otherwise it is not.

One important nuance that should be mentioned is the subscope specifiers (such as "instruction" in the previously mentioned rule). These are special capa operators that mean that all of its child nodes must be in the same logical unit of the specified subscope. In the previously mentioned rule for example, usage of the "instruction" operator means that the "lea" CPU instruction and the presence of the "0x4" offset must be within the same operator (i.e., "lea" is the mnemonic and "0x4" is the instruction operand). Other subscope operators include the "basic block" and

"function" operators, and similar logic applies to them. The general scope of the rule (specified in the meta section) specifies the general scope of the rule.

The advantages of capa organizing rule writing and matching into scopes (i.e., instruction, basic block, and function) makes it easier for malware analysts to write and maintain rules for describing capabilities, it introduces several disadvantages that have to do with that particular scoping. For example, if a rule is set to match at the function scope and the malware sample contains a large enough function in it, then capa could end up correlating features (i.e., leaf nodes) that are not necessarily related to each other, thereby resulting in a false negative. Furthermore, if rule authors try to circumvent this issue by attempting to match at the narrower basic block scope instead, then capa could end up overlooking features (i.e., leaf nodes) that are related to each other but separated into different basic blocks due to a jump instruction caused by some error checking code for instance.

2.3 Call Signatures Plugin (CSP)

This tool was proposed by Joren Vrancken as part of their Master's thesis titled "Detecting Capabilities in Malware Binaries by Searching for Function Calls" [18]. Its works by using Hex-Rays' Interactive Disassembler (IDA) as a binary analysis backend that provides the needed information about the analyzed binary, which is then used by the tool alongside a pre-written capability-describing set of rules to determine what capabilities a malware sample has.

First, and once the binary has been loaded and analyzed in IDA, the CSP tool (which is implemented as an IDA plugin) uses IDA to get a list of all functions present in the binary using IDA's Python API. Then, the tool uses IDA's Fast Library Identification and Recognition Technology (F.L.I.R.T) to determine which functions are library functions and which ones were authored by the malware sample's author. The internals of discovered library functions are not analyzed.

After the malware-specific functions have been determined, CSP gets a list of all of the function calls being made by each malware-specific function. This is done with the help of IDA's built-in commercial decompiler, which traces back in the code before each "call" instruction to determine its arguments according the determined calling convention.

Finally, and once CSP has obtained the list of function calls (along with their arguments) using IDA's decompiler, and after parsing the rules that were passed onto it, the tool proceeds to analyze the IDA-extraction function calls and arguments to determine if there exists any matches with the pre-written rules. If a match is found, then it is reported to the analyst.

While the tool has some disadvantages relative to capa such as its lack of support for instruction mnemonic and bytes matching, it still manages to outperform capa in certain areas. Namely, in the identified weakness discussed in the previous section related to capa. CSP had a slightly higher detection rate using a dataset of malware samples compiled by the tool's author, as well as beating some of the cases where capa had false positives due to function-scope capa rules incorrectly correlating irrelevant features together, which is an issue that has already been identified in the previous section.

2.4 Discussion

In summary, we have showcased three state-of-the-art static malware capability extraction tools, as well as the advantages and disadvantages of each one of them.

From the review, it can be seen that capa is the most advanced and sophisticated of the bunch for this task, given its support for both the same low primitives supported by YARA, as well as its rich and existing code infrastructure which allows for high level reasoning and abstractions, thereby making the tool more extensible.

Additionally, we also showcased the weaknesses of capa that had been discovered

by contemporary research, as well as their relevance to the thesis' topic.

As a result, capa will be the tool that will be utilized in this thesis for implementing superblocks malware capability extraction.

3 Superblock Static Binary Analysis

In this chapter, we aim to perform a semi-structured literary review for the research conducted into superblock formation. Specifically, in the context of static binary analysis and malware analysis. We also speculate on how the analyzed results could help us in answering the research problem of this thesis.

We will begin the chapter by describing the methodology we followed and the results we obtained in the first section. Then, we will analyze the given results and provide an overall view of the related literature in the following section. Finally, we will theorize on how previous work could help guide our contribution in this thesis in the final section of this chapter.

3.1 Methodology

In order to find academic work that is relevant to the topic of superblocks formation and malware capability extraction, we perform a search on 5 major computer science academic journals with the general term: "superblock". When applicable, we use the journal's own field filtering options to select only topics related to Computer Science.

Results can be seen in Figure 3.1.

Figure 3.1: Literature Review Academic Journals Research Results

We also note that we tried using other search terms in the aforementioned journals, such as "superblock formation" and "superblock static binary analysis", but the general term "superblock" seemed to provide the most results given its general nature.

3.2 Analysis

Given the narrow scope of the topic at hand, it was expected that academic literature on it would be scarce, especially considering that most malware analysis field advancements tend to happen in the commercial sector. This was indeed the case with our search, and no articles were found that were directly related to the topic of the thesis.

The majority of the articles encountered throughout our search were coming from the compiler optimization field. The scope and goal of those articles was

the development of heuristic or probabilistic models that could predict with decent certainty the likelihood of different execution paths being taken during the program's execution.

The utility of such models is that they could be incorporated into different compilers that could utilize them to determine the paths that a program is most likely to take, and therefore generate machine code in such a way that those specific execution paths would be given priority.

Another observed use case for the papers we went through was in the field of CPU caching and optimization. Since accurate prediction of which chain of basic blocks (i.e., a superblock) are likely to be taken at a specific point of the program's execution can help the CPU cache the right code segments, and therefore speed up program's execution by preventing cache misses.

3.3 The Utility of Previous Research

While the encountered research does not prove directly useful to our research problem, several portions of it can be utilized and incorporated into our contribution in a constructive manner.

One instance for this can be extracted from the research paper published in 1993 by Hank et al. titled "Superblock Formation Using Static Program Analysis" [19], wherein the authors define many foundational terms and concepts that will prove useful to our thesis.

For example, the authors of that paper define a superblock as a "block of instructions in which control may only enter at the top but may leave at one or more exit points", and that "when the execution stream enters a superblock, it is likely that all basic blocks in that superblock will be executed". This definition seems to be agreed upon in other literature that followed this paper (albeit in CPU cache algorithms and compiler optimization), and provides us with legitimacy for naming

the scope we propose in this thesis the "superblock" scope.

Additionally, the definition given above —in addition to illustrations from that paper— also provide us with guidelines for how superblocks should be formed. For instance, if at a certain point of the superblock's execution it loops back to itself (i.e., a basic block in that superblock), then that superblock should ideally be divided into two superblocks that are split at the basic block which was branched into. However, this suggestion might not be very fruitful in our specific application (malware analysis), and we might choose to relax this assumption a bit).

Additionally, the superblock formation strategies illustrated by diagrams in the aforementioned paper also suggest that when a basic block branches into two other basic blocks that are not part of the current superblock, then if we wish to extend the current superblock to include those two newly-encountered basic blocks, then we must split the current superblock into two superblocks, one for each branch.

4 Target System: Capa

The contribution we make in this thesis to capa spans a large set of the tool's components, such as its rules' parsing and matching logic, its static binary analysis process, and the organization and display of discovered malware capabilities. As a result, it is vital that we provide an adequate description of the components and functioning of the target system in order to better describe the contribution we make to the tool.

In this chapter, we will describe the general architecture of capa and its operational model. Due to the large size of the tool, We will not give extensive details about elements of the tool that are not relevant to the implementation of our contribution in this thesis.

4.1 Overview

Capa takes in as input analyst-written rules and attempts to match them across the sample provided to it by the user. In order to do this, it relies on three major components: a rules parser, a binary analyzer (named internally as a Feature Extractor), and a logic matching engine that tries to see which rules apply on the output provided by the binary analyzer.

Additionally, normal execution of capa (as opposed to it being invoked as a Python module) utilizes two additional modules: a "capabilities" module that is responsible for orchestration between the aforementioned components, as well as a

results rendering component for displaying neat output to the user (including the matched rules and where exactly in the binary did they match).

In the following subsections, we will provide the necessary details and structure behind each of these components.

4.2 Feature Extractors

Capa utilizes several sample analysis tools as backends for extracting features and artifacts from the provided samples. Each of the backends that capa supports are wrapped internally by classes named "Feature Extractors", which abstract away the usage details of the specific backend at hand, and provide a common programming interface that makes it easier for capa developers to maintain the tool and add support for other analysis backends as needed. Two types of feature extractors exist, static, and dynamic.

Static feature extractors operate on the original provided binary sample, and directly instrument the binary via static analysis in order to extract features and artifacts from it such as functions, strings, and control flow information.

Dynamic feature extractors on the other hand operate on a sandbox analysis report (in JSON or LOG format), and they extract artifacts and sample features from the information contained in that sandbox analysis report.

Both static and dynamic feature extractors operate in terms of "scopes", which can be defined as logical units or containers within which sample features are organized and grouped. These scopes are later used by the capa rule matching engine to see which capabilities exist at each of these logical grouping units.

The static scopes are:

- **instruction:** this logical unit contains the features extracted at each instruction of the program, such as the instruction's mnemonic and operands.

- **basic block:** this scope entails features identified across a group of successive instructions with no branch-related instructions (such as *jmp* and *ret*) in it, other than the final instruction of this block. Features contained in this scope include all of the features extracted at the constituent instructions, in addition to basic block-exclusive features such as the detection of stack strings for instance.

- **function:** this scope provides logical grouping for the sample's methods. Capa rules that have the function scope assigned to them are evaluated on the set of features extracted from each function in the sample. This logical unit contains all of the features identified at the constituent basic blocks (and therefore instructions), as well as function-related features such as identified loops and recursive calls.

- **file:** this scope contains all of the features identified across all of the functions contained in the sample, in addition to information about the sample's sections, read-only data, and imports for instance.

- **global:** this logical grouping contains features that are not extracted from an instruction of a basic block or a function per se, but are still made available and shared across all of the static scopes. Examples of this are the "architecture" and "os" features which are often extracted from the sample's binary metadata, and then made available at each scope of the sample to aide to with writing rules' logic.

Dynamic scopes on the other hand are:

- **calls:** this logical unit groups features identified at a single API call that was logged by the sandbox. Examples of such features include function names and the provided arguments.

- **span-of-calls:** made up of a constant number of calls that are in succession. Evaluation at this scope is performed in a sliding-window fashion, and this scope helps prevent false positives in cases where a thread might contain a large volume or calls.
- **thread:** a logical unit for grouping features on a per-thread basis.
- **process:** a logical unit for grouping features on a per-process basis.
- **file:** similar to the static "file" scope.
- **global:** similar to the static "global" scope.

One downside for the usage of static backends as opposed to dynamic ones is that they do not work well on packed or obfuscated samples, because they have no meaningful way of obtaining the hidden contents of the binary at hand. However, they are often preferred over dynamic extractors whenever the sample is unpacked, because they give analysts the ability to navigate to the location of a capa rule match, and then further analyze the function or basic block where the match occurred. In this work, we will focus only on static feature extractors, with the dynamic ones being out of the scope and relevance of this thesis' contribution.

As previously mentioned, all static feature extractors implement a common interface. The methods we are are most interested in for the scope of this thesis are the following:

- **get_functions(self):** this object method retrieves all of non-library functions identified in the sample.
- **extract_function_features(self, fh):** this method takes in as argument a function handle *fh*, which contains an address (alongside possible metadata), and returns all of the function-exclusive features identified in that function.

- **get_basic_blocks(self, fh):** this method takes in a function address (contained in the variable *fh* of type "Function Handle"), and returns a list of all the basic blocks extracted from that function.

- **extract_basic_block_features(self, fh, bh):** this method takes in a function identifier, as well as a basic block identifier, and extracts all of the basic block-exclusive features identified in that block.

- **get_instructions(self, fh, bh):** this method takes in a function address (stored in the *fh* function handle) and a basic block address (stored in the *bh* basic block handle), and returns all of the instructions that compose the basic block identified by the *fh* and *bh* handles.

- **extract_instruction_features(self, fh, bh, ih):** this method takes in a function handle, basic block handle, and an instruction handle, and returns a list of all features identified in the instruction referenced by the input function, basic block, and instruction handles.

These methods were listed because they are are the core of how capa extracts features within their right scope from the provided binary sample. We will give more details on these methods later in this chapter.

Regarding the static binary analysis frameworks that capa supports, we note the following:

Vivisect: a free, open-source, and well-documented binary analysis framework that supports multiple file formats such as Linux's *ELF*, Windows' *PE*, and MacOS' *Mach-O*. The contribution made in this thesis will be based primarily on the interface provided by this framework given its free-to-use nature.

dncil: this is an open-source tool for disassembling Microsoft *.NET*'s Common Intermediate Language (*CIL*) instructions. We will not be handling .NET binaries

in this thesis, since the basic block scope is not supported for them as of the writing of this thesis.

Binary Ninja: an interactive disassembler, decompiler, and debugger developed by Vector35. It has support for Python-based scripting as well as headless execution (no GUI) which cape makes use of in order to extract features. However, the free version of this software does not support come with access to the tool's API, and therefore no Python scripting ability.

4.3 Capa's Rules Parser

After having discussed the portion of capa that handles instrumentation of the malware sample, we now turn our attention to the component responsible for parsing and processing the capa rules authored by security analysts.

Capa relies on the YAML human-readable data serialization language as a medium for describing malware capabilities. This not only makes it easier for analysts who are already familiar with YAML to write new rules, but also has the benefit of making processing and storing rules during capa's execution and easier task since prebuilt libraries could be used in this matter.

Each capa rule is stored in a "Rule" object. These objects are initialized from the parsed YAML code, and are checked for semantic and syntactic correctness upon initialization. If a rule is determined to be invalid, capa raises an exception which if left unhandled (in the case of command line execution for instance) then capa simply displays the error and exists.

Internally, the logic of capa rules is represented using a tree data structure. The leaf nodes of the tree are objects of the class "Feature", which hold the type of the feature (API name, instruction mnemonic, extracted string, etc.) as well as the feature's value extracted from the binary; while the non-leaf nodes represent the boolean logic operators (AND, OR, NOT, etc.) or subscopes (Basic Block,

instruction, etc.), and are stored in an object of the class "Statement" that contains the operators name as well as a list of its children, be it other statements of features. Additionally, each type of feature or statement implements an "evaluate()" method that is used by capa's rule matching engine to evaluate whether a node is true or false in the context of the current scope unit (function, basic block, instruction, etc.). We will cover this more in depth in the next subsection.

This rule's logic tree, in addition to other details such as the rule's metadata, is all stored inside the "Rule" object which provides an interface for easy evaluation and manipulation of capa rules by the developers. Some of the most relevant methods that the Rule object provides are the following:

- **from_dict(cls, d, definition):** this method takes in a capa rule in the form of a hashmap (Python dictionary), and uses the "build_statements()" helper method to construct a logic statement tree while making sure the tree is both syntactically and semantically correct. Then, the "from_dict()" method initializes a Rule object from that statement (in addition to the rule's metadata) and returns it to the caller.

- **from_yaml_file(cls, path):** this method reads a YAML file, parses it using a YAML parser, and passes the resulting dictionary to the previously-mentioned "from_dict()" method and returns the resulting object.

- **get_dependencies(self, namespaces):** this method returns a list of names of all of the rules that the current rule relies on for a successful match. All that this method does is that it goes through the rule's logic tree and searches for features of type "match" and returns that rule's name. This method will be useful later on for ordering rules and rule evaluation in a way such that dependency rules are always evaluated before the rules that depend on them.

- **extract_subscope_rules(self):** this method extracts subscope statements

in a rule into independent rules with a unique name, that will be placed before the current rule in the order of evaluation.

- **evaluate(self, features):** this method takes in a set of features and addresses and recursively evaluates the boolean value of the current rule depending on whether the input features satisfy the logic tree contained in the rule.

After all capa rules are converted from YAML format into a list of Rule objects, that list is then used to construct another object of type "RuleSet" that provides an interfaces of several methods that operate on the whole of the rules that it contains. Examples of such methods include a "match()" method that takes in a set of features and iterates through rules to see which ones match for that given set of features, as well as getter methods for retrieving rules of a certain scope (i.e., only function or basic block or instruction scope rules).

While the RuleSet object implements many methods vital to capa's operation, most of them are not relevant to the scope of this thesis (such as RuleSet optimization methods), and will therefore not be covered in this work.

4.4 Matching Engine

Thus far, we have covered the capa component responsible for instrumenting the input malware sample, as well as the component that responsible for handling the capability-describing rules. Additionally, we have also noted that feature extractors retrieve artifacts from the sample in the form of Feature objects of different types (api name, instruction mnemonic, etc.), and that Rule objects hold representations of the rule's evaluation logic in the form of a binary tree whose leaf nodes are Feature objects as well. Now, we will discuss the rule matching engine that sits between these two components, and handles matching features retrieved from the sample with features stored in a Rule object's statement tree in order to generate a

capa rule match.

Critical for the functioning of capa's rule matching engine is two components, the Statement class and its subclasses, as well as the *match()* method. The Statement class has been briefly mentioned while describing the RuleSet component, and that it is used for representing non-leaf nodes in a rule's evaluation logic tree. More specifically, each supported logic operator in capa is implemented in the form of a subclass of the Statement class, with the evaluation logic (defined in the inherited *evaluate()* method) being implemented differently depending on the operator being represented.

For example, the operator "and" is represented by the class "And" which inherits from the Statement class. When the *evaluate()* method is invoked on an "And" object, all of the immediate children of the object in the statement tree are evaluated, and the method returns True only if all of the child nodes evaluate to True. For the "or" operator however (represented by the "Or" subclass of the parent Statement class), only one child of the node needs to be true in order for the "or" statement to evaluate to true.

As for subscope statements (nodes), then these cannot be evaluated directly, and they must first be extracted into independent temporary rules and then linked to the parent rule as a dependency (using a "match" feature with the temporarily-created subscope rule's name as value for the match feature).

The second component for the engine is the *match()* function, which takes in as input a RuleSet, a set of features, and an address (this will be used to mark where the matching is being tested, be it at the function, basic block, or instruction levels).

Upon invocation, the *match()* function goes through all rules in the input rule set, and then attempts to evaluate it against the inputted set of features to see whether a match exists or not. If a match is found, it is added to the set of discovered matches, and also to the inputted set of features (in the form of a "match" feature)

in order for it to be considered by later rule evaluations that might depend on this recently matched rule.

While the *match()* function originally belongs to the capa engine, it should be noted that a more performant version of it is implemented by the RuleSet object. This more performant version relies on the fact that some features or rules are more likely to occur or match than others, and therefore can be checked first in order to short circuit the entire evaluation process. However, the more performant *RuleSet.match()* method is outside of the scope of this thesis.

4.5 Results Rendering

Once the features have been extracted, rules have been parsed, and capabilities have been matched using the capa matching engine, the results are then stored by capa in a dataclass object of type "ResultDocument" implemented as a Pydantic model. This object contains the processed sample's metadata, as well as a list of all matched rules and where they (and their constituent features) were found and matched. Then, capa passes this result document object to one of four result rendering components in order to display the matches to the user.

4.5.1 Display Modes

Capa supports the following display modes: default (show only metadata and rules), JSON, verbose, and vverbose (very verbose). The specific display mode that will be used is determine based on the arguments that capa was invoked with.

Default

The default (simple) capa display mode shows metadata, extracted ATT&CK TTPs, Malware Behavior Catalog (MBC) objectives [20], as well as the rules names. An

example can be seen in Figure 4.1.

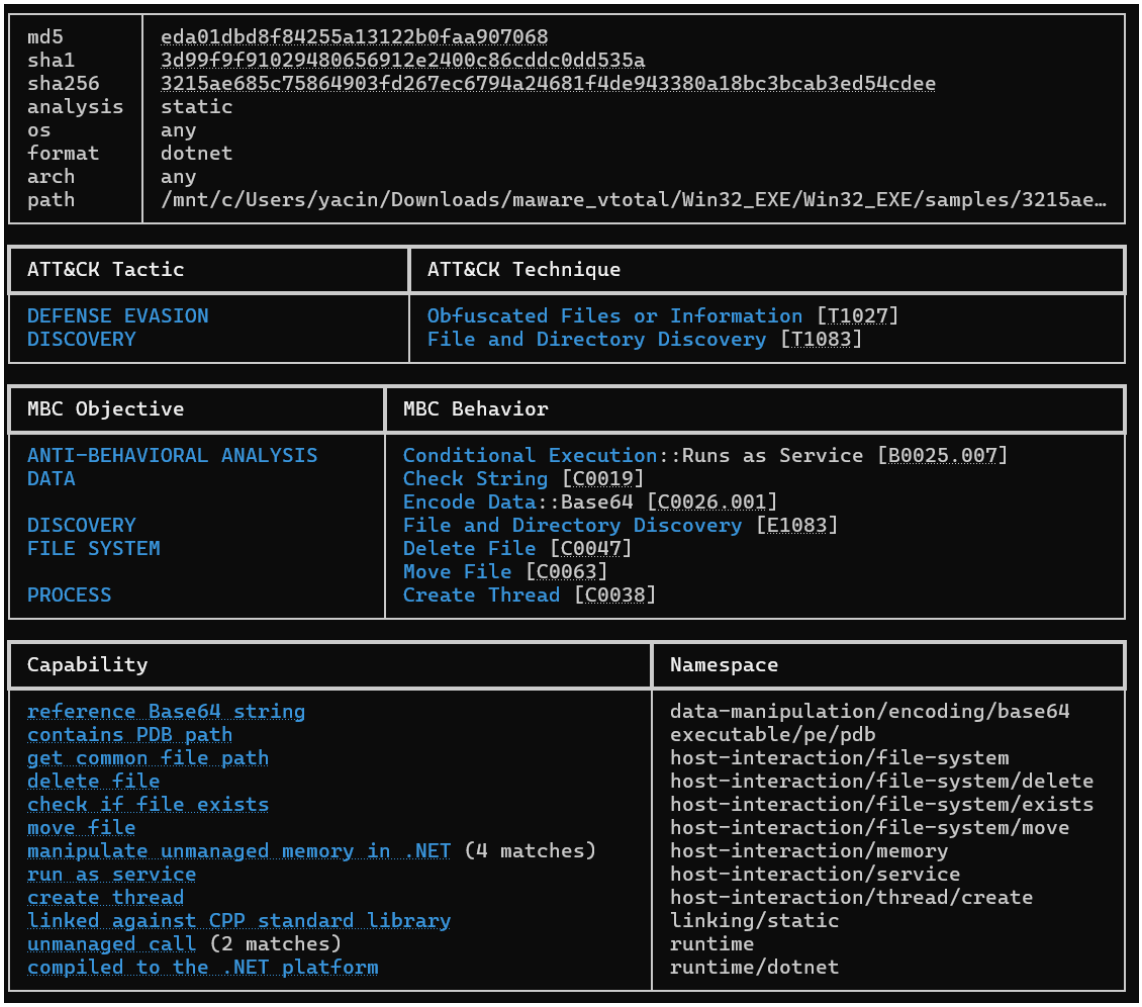


Figure 4.1: default display mode for capa

Verbose

As for the verbose mode of display, then it does not display the ATT&CK and MBC details, but instead shows where each capability was extracted. An example can be seen in Figure 4.2.

```
md5                     eda01dbd8f84255a13122b0faa907068
sha1                    3d99f9f91029480656912e2400c86cddc0dd535a
sha256                  3215ae685c75864903fd267ec6794a24681f4de943380a18bc3bcab3ed54cdee
path                    /mnt/c/Users/yacin/Downloads/maware_vtotal/Win32_EXE/Win32_EXE/sampl…
timestamp               2025-05-15 17:57:56.344198
capa version            9.1.0
os                      any
format                  dotnet
arch                    any
analysis                static
extractor               DnfileFeatureExtractor
base address            global
rules                   /mnt/c/Users/yacin/Downloads/maware_vtotal/Win32_EXE/Win32_EXE/capa-…
function count          39
library function count  0
total feature count     15268

reference Base64 string
namespace  data-manipulation/encoding/base64
scope      file

contains PDB path
namespace  executable/pe/pdb
scope      file

get common file path
namespace  host-interaction/file-system
scope      function
matches    token(0x6000009)

delete file
namespace  host-interaction/file-system/delete
scope      function
matches    token(0x6000004)

check if file exists
namespace  host-interaction/file-system/exists
scope      function
matches    token(0x6000004)

move file
namespace  host-interaction/file-system/move
scope      function
matches    token(0x6000004)
```

Figure 4.2: verbose display mode for capa

Very Verbose

As for the very verbose mode, it provides the utmost level of details by showing the ATT&CK and MBC related information, as well as the rules authors and where each feature in the rule was matched in the input sample. An example can be seen in Figure 4.3.

```
md5                     eda01dbd8f84255a13122b0faa907068
sha1                    3d99f9f91029480656912e2400c86cddc0dd535a
sha256                  3215ae685c75864903fd267ec6794a24681f4de943380a18bc3bcab3ed54cdee
path                    /mnt/c/Users/yacin/Downloads/mawware_vtotal/Win32_EXE/Win32_EXE/sampl…
timestamp               2025-05-15 17:59:33.305683
capa version            9.1.0
os                      any
format                  dotnet
arch                    any
analysis                static
extractor               DnfileFeatureExtractor
base address            global
rules                   /mnt/c/Users/yacin/Downloads/mawware_vtotal/Win32_EXE/Win32_EXE/capa-…
function count          39
library function count  0
total feature count     15268

reference Base64 string
namespace  data-manipulation/encoding/base64
author     moritz.raabe@mandiant.com
scope      file
att&ck     Defense Evasion::Obfuscated Files or Information [T1027]
mbc        Data::Encode Data::Base64 [C0026.001], Data::Check String [C0019]
regex: /ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
  - "%63[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789]%n:%d%n" @ file+0x22B5E2
  - "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/" @ file+0x22C302, file+0x3310BC
  - "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_" @ file+0x22B84A

contains PDB path
namespace  executable/pe/pdb
author     moritz.raabe@mandiant.com
scope      file
regex: /:\\.*\.pdb/
  - "F:\\CB\\ARM\\BuildResults\\bin\\Win32\\Release\\AdobeARMHelper.pdb" @ file+0x335E44
  - "e:\\fx19rel\\WINNT_5.2_Depend\\mozilla\\obj-fx-trunk\\toolkit\\crashreporter\\client\\crashr
eporter.pdb" @ file+0x12D194
  - "e:\\fx19rel\\WINNT_5.2_Depend\\mozilla\\obj-fx-trunk\\toolkit\\mozapps\\update\\src\\updater
\\updater.pdb" @ file+0x275588
  - "g:\\Acro_root_at\\Acrobat\\Viewer\\Win\\output\\acrobat\\AcroRd32Exe.pdb" @ file+0x15111B, file+0x299E1F
  - "g:\\acro_root_at\\acrobat\\systemsynchronizer\\synchronizerapp\\build\\win\\release\\AdobeCo
llabSync.pdb" @ file+0x236A9A
```

Figure 4.3: vverbose display mode for capa

4.6 Orchestration

The last capa component we need to cover is the capabilities module, which is responsible for orchestrating all of the modules we have covered thus far.

While the "main" and "loader" modules which are responsible to parsing command line arguments and instantiating different components are crucial to orchestration, we will not be focusing on them much in this section. Instead, we will be focusing more on the "capabilities" module which is responsible for extracting features and matching capa rules against these features, and then returning these matches to the main module in order for them to be stored in a ResultDocument object and later displayed by the main module.

The reason we give more attention to the capabilities module is that a large portion of our contribution in this thesis will be made in this module, since it plays a large part in implementing the scoping mechanism that capa has.

The capabilities module has two main components, a static and a dynamic capability extractor. The static component will be the focus of this thesis, while the dynamic one is out of scope.

The static capability extractor is composed of four different functions defined in a similar fashion. Each function corresponds to a scope, and performs feature extraction on that scope as well as iterate through subscopes and match rules on them. For example, the function corresponding to the basic block scope takes in as argument a basic block, and then iterates through all instructions in it, passes each one to the instruction scope-related function, and then extracts features from the current basic block and attempts to match capabilities at that scope. The full list of function is as follows:

- *find_static_capabilities():* this function corresponds to the file scope, and it iterates through all functions in the sample and passes them to the function scope-related method for capability extraction from it. Then, capabilities (i.e., rule matches) extracted at lower subscopes are merged with file features, and then this function attempts to match file-scope rules using this combined feature set.

- *find_code_capabilities():* this method relates to the function scope, and it work similar to the previously described function. The difference is that it takes as argument a malware sample function handle, and iterates through all basic blocks in it and passes each one to the basic block-related capability extraction function. Then, after combining function-level features and extracted basic block capabilities, rule matching is performed on the current function.

- *find_basic_block_capabilities():* this method relates to the basic block scope. It functions similar to the function-related method, except that it takes as argument a basic block and iterates through its constituent instructions, passing each one of them to the instruction scope-related function, and then matching basic block capabilities at the current block after all features and lower instruction-scope matches have been merged.

- *find_instruction_capabilities():* this function relates to the instruction scope. This method differs in that it is the lowest possible scope and therefore does not iterate through lower components. Instead, it extracts the features at the provided instruction, matches rules based on those extracted features, and returns that as output.

When the static capabilities module is invoked, the *find_static_capabilities()* method is called first, and then the remaining functions get called automatically by it. Output of the capabilities module is a set of extracted rule matches.

5 Design and Specifications

In this chapter, we will detail the objectives that our work aims to achieve, as well as the proposed design and our suggestion for incorporating it into capa.

We will begin by defining the objectives of our contribution in the first section. Then, we will detail the methodology we propose for forming superblocks in the context of static binary analysis. Finally, we will describe how our method could be incorporated into the target system.

5.1 Design Requirements

The primary objective behind our contribution is to provide capa rule authors with a way that they could describe logic that must exist on a set of basic blocks in sequence with no interruptions. Such addition would give capa rule authors more expressive powers and would help reduce the rate of false positives stemming for spurious correlations in the case of function scope rules, as well as false negatives stemming form overconstrained basic block scope rules.

In order to achieve our objective, we will first detail the general requirements and design principles behind our contribution in a concrete manner. This will help guide our design process and make testing our solution clear and straightforward.

In the following subsections, we will detail the main tenets behind our approach, and the motivation behind each one. Then, we will proceed to describe our proposed design in the following section.

5.1.1 Superblocks must not contain irrelevant basic blocks

The first requirement for our design is that the formed superblocks that capa rules would match on must not contain any redundant basic blocks in them that do not contribute to a superblock rule's match. Put differently, if a capa rule matches on a superblock, then all of the features that constitute that match must exist on a set of basic blocks that are directly chained together with no gaps in between.

This requirement is important because it is one of the mechanisms that prevent unrelated basic blocks from being incorrectly correlated by the capa engine, such as the case with function scope rules that do not consider the relationships between the constituent basic blocks.

It should also be noted that successful application of this design requirement necessitates that it would be applied on a rule basis. This means that superblock formation would depend on the capa rule being evaluated, and that a set of basic blocks in succession might form a superblock under one capa rule, but not in another.

5.1.2 Superblocks must contain only one execution path

The second design principle is that superblocks must not contain basic blocks from divergent execution paths inside the same superblock.

Such scenario might occur if a basic block at the tail of a superblock contains a branch to an earlier parallel execution path, which would result both the current execution path and the parallel one being wrongfully included in the same superblock.

This condition is set in order to conform to the definition of a superblock as seen in the reviewed literature, as well as to prevent features extracted from basic blocks in different parallel execution paths from being falsely correlated, thereby resulting in a false positive.

5.1.3 Superblocks must not contain loops

The third design principle we propose is for superblocks to be linear with no loops therein, with the only exception being basic blocks that loop over themselves (otherwise known as "tight loops").

The reason behind the introduction of this principle is to make writing rule logic easier and simpler for analysts, and to instead user superblocks as subscopes inside function scope rules whenever a rule author wants to describe a capa rule with complex logic involving loops.

5.1.4 Superblocks can be composed of just one basic block

The fourth design principle we propose is to consider basic blocks as superblocks of length one. The reason we consider a basic block as a superblock is primarily due to the convenience of not having to duplicate similar rules twice, since sometimes malware authors might skip error checking in their code for instance, which would result in what would have been a large superblock instead being collected into a singular large basic block, thereby necessitating a separate and similar basic block rule in order to detect such cases. Instead, we opt for recognizing basic blocks a superblocks of length one so that rule authors can avoid such redundancy and duplication.

5.1.5 Superblock matches cannot be nested

The fifth and final design principle we propose is for superblock rule matches to not be nestable. This means that a superblock rule cannot have as part of its required rule matching logic another superblock rule as a dependency, thereby reducing rule logic complexity.

Instead, if rule authors need to describe logic in terms of predefined superblock rules, they would instead need to set the current rule's scope to the function scope,

and then define superblock subscopes as needed and manage their dependencies as part of the parent function-scope rule's logic.

5.2 Design Description

After having listed the main conditions and requirements that our design must fulfill, we now proceed to describe our proposed design in this section as well as highlight how it satisfies the given requirements.

We will begin by providing an overview of our proposed design and the different stages of matching capa rules on superblocks. Then, we will proceed to detail each stage in its own subsection, while motivating the reasoning behind our approach and how it helps achieve our design goals.

The details of integrating the proposed solution into the target system will be discussed in the final section of this chapter.

5.2.1 General approach

Our proposed approach for matching malware capabilities on superblocks is composed of four main steps. First, a flow graph is constructed for each of the malware's functions. Second, all of the features contained in the function are grouped together as if they were part of the same basic block, and are then searched for all possible matches for superblock-scoped capa rules. Third, each resulting capability match is parsed, and the addresses at which its constituent features were found are extracted and their corresponding basic blocks are determined. Fourth and finally, we go through each of the discovered matches, and determine whether its constituent basic blocks (extracted from step three) form a non-cyclical path on the flow graph generated in step one; If the basic blocks form a non-cyclical path, the superblock match is determine to be valid and is therefore retained, otherwise it is discarded.

These steps will be executed in succession for each of the malware sample's functions in order to extract the superblock malware capabilities in that function. These steps will next be explained in more detail in the following subsections.

5.2.2 Generate the current function's flow graph

In this step, we will construct four data structures that will be required and used by the remaining steps. These data structures are constructed on a per-function basis, and they are as follows: a directed flow graph, a dictionary mapping basic block addresses to nodes of the flow graph, an interval tree that maps any instruction's address to the address of the basic block that houses it, and a global function-wide set of features.

These structures will be constructed in a gradual manner with each encountered basic block in the function being appended into these structures until they are complete. Once all of the basic blocks have been visited and the structures' construction has concluded, we then pass to the following steps.

Flow graph

The first structure of interest is a graph that models the function's control flow in terms of basic blocks. The graph will be a directed rooted graph composed of the function's constituent basic blocks as nodes, with branches between basic blocks representing the edges between these said nodes.

Each node of the graph will contain the address of its associated basic block, as well as up to two pointers to other nodes in the graph. The number of outgoing edges from a given node depends on the type of branching that the associated basic block performs, with possible cases being as follows:

- **0 outgoing edges:** If a given node contains no outgoing edges (i.e., leaf node) then its associated basic block ends with a return instruction, and is therefore

an exit point for the function.

- **1 outgoing edges:** If a given node has exactly one outgoing edge, then its associated basic block ends with an unconditional branch, such as the case with "jmp" (jump) assembly instructions for instance.

- **2 outgoing edges:** If a given node has exactly two outgoing edges, then its associated basic block ends with a conditional branch, such as the case with the "jne" (jump-if-not-equal) assembly instruction for instance.

In order to initiate a graph node object from a given basic block and then include it in the graph, we must first retrieve the addresses that it branches to. For this, it is sufficient to look at the final instruction of a given basic block, then retrieve the offset by which the possible jump is performed, then add it to the final instruction's address, thereby giving us address that the basic block branches to. However, the static binary analysis frameworks that capa works on already implement an interface method that allows us to retrieve the addresses that a conditional instruction branches to, and we will be using that instead for the sake of simplicity.

One thing that remains is how these gradually constructed graph nodes objects will be constructed together, since we said that the graph will be formed one basic block at a time, and for this, we will rely on the second data structure in this initial step.

Hash set of graph nodes

The second data structure that will be constructed in this step will be a dictionary (hash set) that maps basic block starting addresses to their corresponding flow graph node objects, thereby allowing us to find specific nodes in the flow graph in constant time without needing to search the graph each time.

One of the main benefits behind this structure is that it makes construction of

the function's flow graph more straight forward. This is because when initiating the node object for an encountered basic block, the branches of that node could be set to the target addresses for that branch, which can be immediately computed at any given basic block. Then when going through the flow graph, the code would rely on the address-to-node dictionary to translate the branches' addresses into their associated nodes. This approach is more straight forward than having to directly link node objects together by reference.

Another reason for this dictionary structure is that it will later make it easier for us to filter discovered capa matches in the final step of our approach, since given a set of basic blocks that house the features necessary for a match, we can index into their corresponding graph nodes in constant time, and then check whether they exist in sequence or not.

Interval tree of in-function addresses

The third data structure we will be constructing in this step will be an interval tree that maps the address range of each basic block to the starting address of that basic block.

This structure will later be used in the penultimate stage of our approach to retrieve the constituent basic blocks for a capa rule match. Addresses of the features that caused the match will be extracted from that match and looked up in this interval tree, and the corresponding basic blocks will be determined and passed from there to the final filtering step.

Global set of features

This is the final and simplest data structure that will be constructed in our approach. It is a set of all unique features that are contained in the function. It is gradually constructed by adding each of the features extracted from a basic block to it at a

time, until all of the function's blocks had been parsed. This structure will be useful in the second and following step of our approach.

5.2.3 Blind matching on the entire set of rules

In the second step of our approach, and once we have collected a list of all features present in a function across all of its basic blocks, we proceed to match superblock-scope rules on this large features set without taking into consideration that these features might exist on distinct (and possibly distant) basic blocks.

As a result of this step, for any combination of features from the previously constructed features set we will obtain a rule match if that said combination does indeed match a described malware capability. This matching action is performed using capa's regular rule matching engine, which already extracts matches on all possible combinations of features.

Once this set of superblock matches performed on an ambiguous set of features is obtained, our next step is to go through them all and filter them to determine which ones exist on a an actual sequence of uninterrupted basic blocks (i.e., superblocks). But before that, we must first convert the addresses associated with each feature present in a match, and convert it to the starting address of the basic block. This address can then be used with the previously-constructed flow graph to determine whether the match is a valid superblock match or not.

5.2.4 Retrieve the basic blocks used for each match

Given a certain match, and tasked with the goal of finding the set of basic blocks that the match spans across, we must first iterate through the given match to extract all of its constituent features. After that, we can simple utilize the previously-constructed interval tree that maps basic block address ranges to the starting address of that basic block in order to determine what basic block that specific feature at hand was

found in.

In order to list all of the given capability's constituent features, we must traverse the logic tree for the statement contained in the given result's "match" object, and extract leaf tree nodes of type "feature" while continuing to traverse intermediate statement nodes (such as AND, OR, etc.).

Once a list of "feature" objects is obtained, we can iterate through it and search the previously-constructed address interval tree with the current feature object's address value in order to determine the basic block that it is part of. After doing this for all of the features in the extracted set, we obtain the set of all basic blocks that were relevant to the match, which can then input into the following and final step.

5.2.5 Filter the results

After we have obtained a list of all possible matches identified on all possible combinations of features in a function, as well as the list of basic block addresses that pertain to each match, we now proceed to filtering these matches and only retaining the ones that exist on superblocks only.

To begin, we first go through all of the basic blocks pertaining to a given match, and if we find that one of them branches to two other basic blocks relevant to that match at the same time, then we exclude that match since it was discovered on divergent superblocks (i.e, features from two parallel execution paths might have been wrongfully correlated together). We exclude however the edge case where a basic block would branch to itself (i.e., tight loop) and to another basic block relevant for that match at the same time.

Once we have filtered out matches whose constituent basic blocks resided on divergent superblocks, we now proceed to filtering the rest of the remaining matches. To do so, we use the following algorithm:

1. To start off, we pick a random basic block from the list of basic blocks related to a match, then fetch the graph node associated with it using our previously-constructed address-to-node dictionary (from step one of the approach). We consider this node as the head of a basic block cycle of unknown length (at the time of picking this node), and we add it to a set of encountered cycles.

2. Once an initial graph node has been selected, we traverse the graph starting from that node in order to find the longest path composed of only unvisited basic blocks that are relevant for our match. We do so by keeping a pointer to the current node, which is initialized to the node picked in step 1, and then moving that pointer along the branch whose related basic block is relevant for the current match. We do this until we reach a graph node that is not in the list of previously unseen match-related basic blocks, at which point we move to the next step of the algorithm.

3. Once we have come to the node at the end of the current cycle, we check its nature. If that node happens to be the start of another cycle (i.e., is contained in the set of encountered cycles), then we merge the two cycles together and keep only the current more-precedent cycle head in the maintained set of encountered cycles. Otherwise, we deduce that we are either at the end of the function (i.e., exit node), or that the basic block is irrelevant to the match, and we therefore determine that the current cycle is finished, and we move back to step 1 of this algorithm and pick another never-before-seen node to repeat the process with until all nodes has been visited.

4. Once all of the basic blocks relevant to the match have been covered, and all possible disjoint cycles related to them have been constructed and added to the set of encountered cycles, we then proceed to check the length of that set. If the set has only one cycle remaining, that means that the match's

given basic blocks exist in sequence and are therefore a superblock, otherwise, we determine that the capability was matched on disjoint basic blocks of the function, and the match is therefore discarded.

After applying the algorithm listed above, we are left with only the matches that exist strictly on superblocks as previously defined in the next chapter, and these matches are then what will be returned to the user.

5.3 Integrating With Capa

Now that we have outlined our proposed approach for extracting superblock malware capabilities from binary samples, we proceed into discussing how our contribution will be integrated into capa in this chapter. More specifically, we outline the capa components that should be modified for our contribution to work, and how those components should be modified.

5.3.1 Rules Parser

The first component that should be modified is the capa rules parser, which needs to be modified to accept the keyword "superblock" as a valid capa rules scope and subscope, as well as define the scope internally in the module's static scopes enumeration class which will be used by other rule constructing and matching logic.

Additionally, the supported features for the superblock scope need to be correctly assigned to it. In this case, these would be all of the basic block features from the lower scope, with the possibility of adding superblock-exclusive features in the future.

Furthermore, the rules parser module needs to be updated with the new correct static scopes hierarchy, which puts the newly created superblock scope in between the function and basic block scopes.

5.3.2 Feature Extractor

The next component that requires modification is the features extractors module. The "StaticFeatureExtractor" interface needs to have two methods added to it, *get_next_basic_blocks()* which will be used to retrieve the branches of a given basic block, and the *get_basic_block_size()* which will return the size of a given basic block in bytes, which in turn will be used when constructed the previously-mentioned inner-function addresses interval tree.

Additionally, the feature extractors for static analysis backends that we wish to support (in our case Vivisect since it is free and open-source) need to be updated to support the newly introduced methods, with other feature extractors (such as the .NET extractor or Hex-Rays' IDA) being adjusted to provide a blank implementation of the newly introduced abstract methods until support is added.

5.3.3 Capabilities Module

This component will house the majority of our contribution including flow graph construction, capability extraction, and match filtering. The code for these tasks and data structures will all be housed in a class named *SuperblockMatcher*, which would internally house the data structures described in section 5.2, in addition to the necessary methods required to operate on these structures and carry out all of the tasks previously described.

A new *SuperblockMatcher* object will be instantiated each time a function is encountered, with basic blocks being gradually added to it using a method it exposes named *add_basic_block()*. This method will be responsible for generating a graph node for that given basic block as well as appending it to the flow graph and its address range to the addresses interval tree.

Once all basic blocks have been added to the matcher object, the *match()* method that the SuperblockMatcher object exposes will be invoked, which will in turn search

for capabilities on all of the detected function features, and then proceed to filtering the found matches using a private *_prune()* method. Correct matches are then added to the list of extracted capabilities, and are then returned for them to be displayed to the user.

To make matters easier for storing matches and distinguishing superblocks apart, we store the list of all constituent basic blocks for a superblock alongside the associated match, which will later simply the process of displaying where exactly a superblock capability was extracted.

5.3.4 Results Rendering

The final component we need to adjust is the results rendering component, which needs to be formatted and refactored to support displaying rule matches extracted at the new superblock scope.

Most of the capa code revolving around results rendering is dynamic and extensible, meaning that minimal refactoring is needed in this component in order to display simple superblock matches. All necessary information will be automatically extracted by the relevant display component.

One exception however is that verbose and very verbose display modes need to be slightly adjusted to display the addresses at which superblocks were matched. This is because these components were originally designed to display single addresses pertaining to an instruction, basic block, or function, whereas superblock matches can span a number of basic blocks in sequence. To display this accurately, we rely on the list of addresses that is stored alongside each superblock match (as explained towards the end of the last subsection), and we simple format this list of addresses into a presentable string for it to be displayed to the user.

6 Implementation

After having outlined the proposed design for our contribution, as well as how we plan to integrate it into the target system (capa), we now proceed into implementing our solution and testing its functionality, viability, and efficacy with regards to addressing the problems outlined at the start of the thesis.

First, we will briefly go over general implementation details such as external libraries used, as well as interfacing with the API's of the static analysis backends. Then, we will verify that our solution works in the following section, and showcase the output across the different display modes. Finally, we will provide a real life use case for our proposed solution, and motivate through it how our contribution improves the target system.

6.1 Implementation

The written code concerning our contribution has been submitted in a Pull Request to the capa GitHub repository. As of the writing of the thesis, it is awaiting approval from the maintainers of the target system, followed by addition into the upcoming version of the tool. The entirety of the code can be viewed using the following link: `https://github.com/mandiant/capa/pull/2665/`

6.2 Verification

In order to verify that our code can indeed extract malware capabilities that exist across superblocks, and to also showcase how it displays such matched superblock rules, we will be doing a test run of the the updated target system with a custom sample and rule to verify that our code functions.

6.2.1 Verification sample

The sample that we will be utilizing for our verification is taken from the following GitHub repository belonging to the thesis's authors: `https://github.com/yelhamer/light-C-server/`.

The test sample is a lightweight file-sharing web server written entirely in C, whose function is to listen for incoming connections, accept them when they arrive, get the filename that the remote user wants to download, and send the contents of that file back.

The superblock capability we will be aiming to match in our verification process will be the setting up of a connections listener. This superblock can be found in the *init_server()* method, and its disassembly using Hex-Rays' IDA can be seen in Figure 6.1.

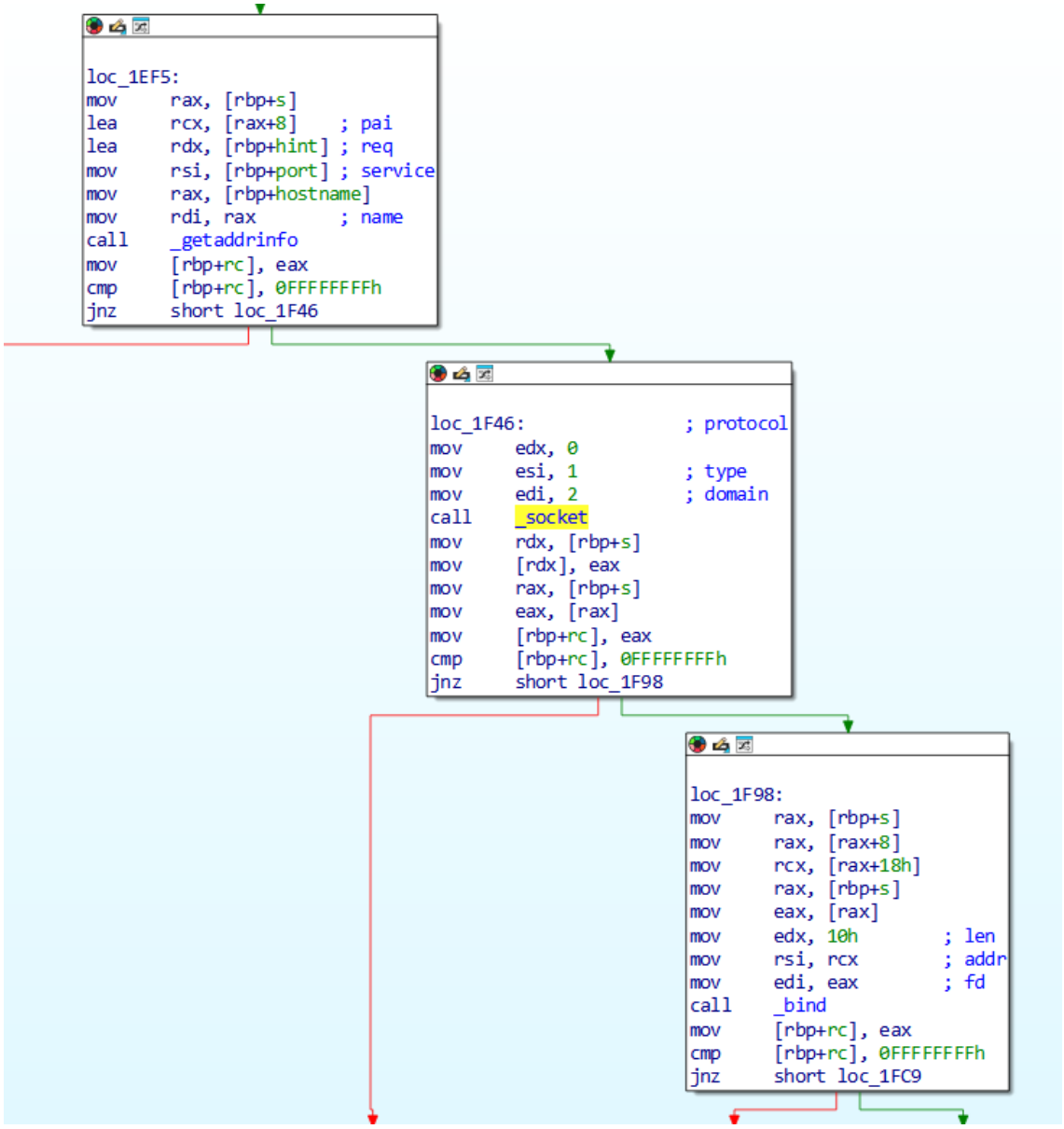


Figure 6.1: Disassembly View of the Target Match Superblock

As we can see in the figure above, the superblock at hand spans the basic blocks with addresses ending in: *0x1EF5*, *0x1F46*, and *0x1F98*.

6.2.2 Test rule

In order to match the capability listed in the previous subsection, we use the following simple capa rule:

```
rule:
  meta:
    name: Initialize File Sharing Server (superblock test rule)
    namespace: communication
    authors:
      - "Yacine Elhamer (@yelhamer)"
    scopes:
      static: superblock
      dynamic: unsupported
  features:
    - and:
      - basic block:
        - or:
          - api: getaddrinfo
          - api: _getaddrinfo
      - basic block:
        - or:
          - api: socket
          - api: _socket
      - basic block:
        - or:
          - api: bind
          - api: _bind
```

The rule searches for at most 3 basic blocks in succession that contain the API calls *getaddrinfo()*, *socket()*, and *bind()*.

6.2.3 Results

Now with the rule and verification sample ready, we move to execution. We will be running capa with the previously described sample and the previously-listed rule, and will be showing the results in simple, verbose, and very verbose output modes.

Simple

As we can see in Figure 6.2, the superblock capability was indeed detected and correctly outputted.

Figure 6.2: Capa Execution Result (Simple Output Mode)

Verbose

Moving on to the verbose display mode, we can see in Figure 6.3 more information, and in particular, that the basic blocks that constitute the superblock are indeed the ones we previously highlighted when selecting the sample.

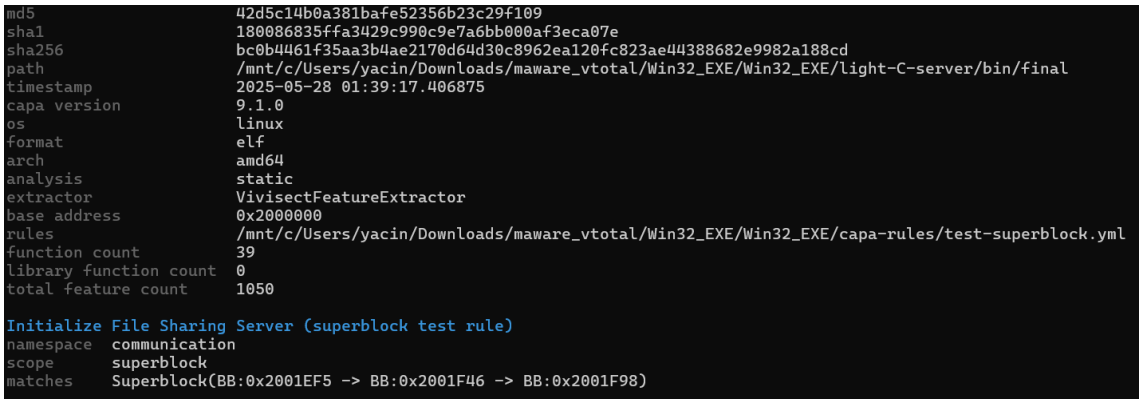


Figure 6.3: Capa Execution Result (Verbose Output Mode)

Very verbose

Finally, we execute capa for the last time using the very verbose (-vv) flag, and we can see in Figure 4.3 that capa correctly outputs the addresses corresponding to each basic block.

```
md5                   42d5c14b0a381bafe52356b23c29f109
sha1                  180086835ffa3429c990c9e7a6bb000af3eca07e
sha256                bc0b4461f35aa3b4ae2170d64d30c8962ea120fc823ae44388682e9982a188cd
path                  /mnt/c/Users/yacin/Downloads/maware_vtotal/Win32_EXE/Win32_EXE/light-C-server/bin/final
timestamp             2025-05-28 01:41:21.996260
capa version          9.1.0
os                    linux
format                elf
arch                  amd64
analysis              static
extractor             VivisectFeatureExtractor
base address          0x2000000
rules                 /mnt/c/Users/yacin/Downloads/maware_vtotal/Win32_EXE/Win32_EXE/capa-rules/test-superblock.yml
function count        39
library function count  0
total feature count     1050

Initialize File Sharing Server (superblock test rule)
namespace  communication
author     Yacine Elhamer (@yelhamer)
scope      superblock
superblock @ Superblock(BB:0x2001EF5 -> BB:0x2001F46 -> BB:0x2001F98)
  and:
    basic block: @ 0x2001EF5
      or:
        api: getaddrinfo @ 0x2001F0C
    basic block: @ 0x2001F46
      or:
        api: socket @ 0x2001F55
    basic block: @ 0x2001F98
      or:
        api: bind @ 0x2001FB4
```

Figure 6.4: Capa Execution Result (Very Verbose Output Mode)

6.3 Testing

Now that we have confirmed that our contribution does indeed function and that it is capable of extracting superblock matches, we now try to adjust one of capa's existing rules to make it more accurate using the new superblock scope.

6.3.1 Test case

We have identified a false positive for the "RC4 PRGA" malware capability present in a native Windows binary named *AgentService.exe*. The match was identified in

the function with the address *0x1400056E0*, however, after navigating to that function we can see that it is a statically linked wmemcmp function from the standard C run-time library as shown in Figure 6.5. This method is responsible for comparing characters in two buffers [21].

```
.text:00000001400056E0 ; =============== S U B R O U T I N E =======================================
.text:00000001400056E0
.text:00000001400056E0
.text:00000001400056E0 ; int __cdecl wmemcmp(const wchar_t *S1, const wchar_t *S2, size_t N)
.text:00000001400056E0 wmemcmp         proc near               ; CODE XREF: boost::filesystem::detail
.text:00000001400056E0                                         ; DATA XREF: .rdata:000000014009D234↓o
.text:00000001400056E0
.text:00000001400056E0 arg_8           = qword ptr  10h
.text:00000001400056E0 arg_10          = qword ptr  18h
.text:00000001400056E0
.text:00000001400056E0                 mov     [rsp+arg_8], rbx
.text:00000001400056E5                 mov     [rsp+arg_10], rdi
.text:00000001400056EA                 xor     eax, eax
.text:00000001400056EC                 mov     rdi, rdx
.text:00000001400056EF                 cmp     cs: Avx2WmemEnabledWeakValue, eax
```

Figure 6.5: Disassembly of the Match Location in the Test Sample

Further examining of the match reveals that the constituent features were found on disperse basic blocks, and then falsely correlated together under the assumption that they were all in sequence. Therefore, we deduce that by changing the scope fo the original rule to the newly-added superblock scope we would fix this false positive.

6.3.2 Updated rule

The original rule relies on counting the number of references to the current function, as well as the number of basic blocks it contains, in order to try to minimize the rate of false positives. However, this does not completely prevent them as previously seen with the *wmemcmp()* standard C function in the default *AgentService.exe* binary. The original rule is shown in the following snippet:

```
features:
  - and:
    - count(characteristic(nzxor)): 1
    - or:
      - match: calculate modulo 256 via x86 assembly
      - count(mnemonic(movzx)): 4 or more
    - count(characteristic(calls from)): (0, 4)
    - count(basic blocks): (4, 50)
    - match: contain loop
    - optional:
      - or:
        - number: 0xFF
        - number: 0x100
```

After inspecting the original rule that caused the false positive, we suggest an updated rule that does not rely on counting function cross references or the number of basic blocks in a function to eliminate false positives. Instead, it follows a more natural approach by trying to find the necessary logic and artifacts being in close sequence. More precisely, it tries to find a non-zeroing xor operation on the same superblock with other basic blocks that contain a total of 4 or more movzx instruction (commonly used in RC4 PRGA implementations) or a basic block that computes the 256 module using x86 assembly. If such superblock match is found, and a loop is found alongside it in the same containing function, then a discovered RC4 PRGA capability match is returned. Full description of the the updated rule can be seen in the following snippet:

```
rule:
  meta:
    name: encrypt data using RC4 PRGA (Superblock)
    namespace: data-manipulation/encryption/rc4
    authors:
      - Yacine Elhamer (@yelhamer)
    scopes:
      static: function
      dynamic: unsupported
  features:
    - and:
      - superblock:
        - and:
          - or:
            - match: calculate modulo 256 via x86 assembly
            - count(mnemonic(movzx)): 4 or more
          - characteristic: nzxor
      - match: contain loop
```

6.3.3 Results and Comparison

In order to verify whether the new superblock-based RC4 PRGA rule eliminates the observed false positive or not, we will execute capa with the new rule on the *AgentService.exe* benign executable that contained the false positive, as well as three different malware samples that were confirmed to be using RC4 encryption. These three samples were originally listed as example malware samples in the original RC4 PRGA capa rule. The list of used false positive (FP) and true positive (TP) samples

as well as their associated MD5 hashes can be seen in Table 6.3.3.

Sample Name	Contains RC4 PRGA?	MD5 Hash
FP1	No	5F2DD9A74F198BA69135BCE9BC534560
TP1	Yes	34404A3FB9804977C6AB86CB991FB130
TP2	Yes	9324D1A8AE37A36AE560C37448C9705A
TP3	Yes	73CE04892E5F39EC82B00C02FC04C70F

Table 6.1: Samples Used for Testing The RC4 PRGA Superblock Rule

Next, we proceed to execute capa with both the old and new RC4 PRGA detection rule, and we observe which of those rules matches for each sample. First, we begin by running capa on the TP samples, and then the FP samples. After that, we compile the obtained results and compare the old capa rule and the new superblock one. Capa analysis results for the sample TP1 can be seen in Figure 6.6, with the match for the superblock-based RC4 PRGA rule being highlighted in white.

Capability	Namespace
log keystrokes	collection/keylog
log keystrokes via polling	collection/keylog
capture screenshot	collection/screenshot
hash data with CRC32 (3 matches)	data-manipulation/checksum/crc32
encode data using XOR (5 matches)	data-manipulation/encoding/xor
encrypt data using RC4 KSA	data-manipulation/encryption/rc4
encrypt data using RC4 PRGA (2 matches)	data-manipulation/encryption/rc4
encrypt data using RC4 PRGA (Superblock) (2 matches)	data-manipulation/encryption/rc4
generate random numbers using the Delphi LCG (3 matches)	data-manipulation/prng/lcg
read clipboard data	host-interaction/clipboard
set environment variable	host-interaction/environment-variable
get common file path	host-interaction/file-system
create directory	host-interaction/file-system/create
delete directory (2 matches)	host-interaction/file-system/delete
delete file (2 matches)	host-interaction/file-system/delete
check if file exists	host-interaction/file-system/exists
enumerate files on Windows	host-interaction/file-system/files/list
read file on Windows (3 matches)	host-interaction/file-system/read
write file on Windows (5 matches)	host-interaction/file-system/write
get graphical window text	host-interaction/gui/window/get-text
get session user name	host-interaction/session
create thread	host-interaction/thread/create
terminate thread	host-interaction/thread/terminate
link function at runtime on Windows (4 matches)	linking/runtime-linking
linked against XZip	linking/static/xzip
enumerate PE sections (2 matches)	load-code/pe

Figure 6.6: Capa Analysis Results for Sample TP1

Capa analysis results for sample TP2 can be seen in Figure 6.7, with the match for the superblock-based RC4 PRGA rule being highlighted in white.

Capability	Namespace
contain obfuscated stackstrings (14 matches)	anti-analysis/obfuscation/string/stackstring
receive data (2 matches)	communication
send data (2 matches)	communication
resolve DNS (2 matches)	communication/dns
get socket information	communication/socket
get socket status (2 matches)	communication/socket
initialize Winsock library	communication/socket
set socket configuration (2 matches)	communication/socket
act as TCP client	communication/tcp/client
encrypt data using RC4 KSA	data-manipulation/encryption/rc4
encrypt data using RC4 PRGA	data-manipulation/encryption/rc4
encrypt data using RC4 PRGA (Superblock)	data-manipulation/encryption/rc4
query environment variable	host-interaction/environment-variable
get common file path (2 matches)	host-interaction/file-system
copy file	host-interaction/file-system/copy

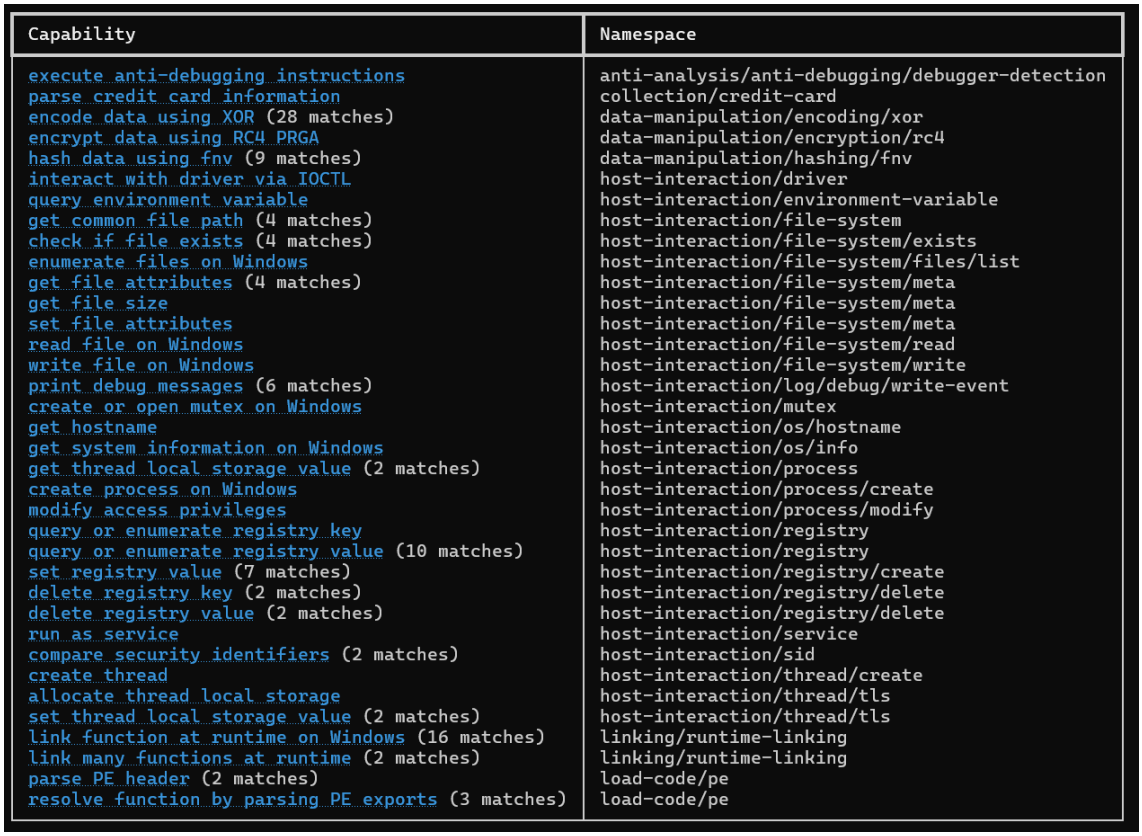
Figure 6.7: Capa Analysis of Sample TP2

Capa analysis results for sample TP3 can be seen in Figure 6.8, with the match for the superblock-based RC4 PRGA rule being highlighted in white.

Capability	Namespace
execute anti-debugging instructions	anti-analysis/anti-debugging/debugger-detection
check HTTP status code	communication/http/client
hash data with CRC32	data-manipulation/checksum/crc32
encode data using Base64	data-manipulation/encoding/base64
encode data using XOR (13 matches)	data-manipulation/encoding/xor
encrypt data using AES (2 matches)	data-manipulation/encryption/aes
encrypt data using Curve25519 (2 matches)	data-manipulation/encryption/elliptic-curve
encrypt data using RC4 KSA	data-manipulation/encryption/rc4
encrypt data using RC4 PRGA	data-manipulation/encryption/rc4
encrypt data using RC4 PRGA (Superblock)	data-manipulation/encryption/rc4
encrypt data using Salsa20 or ChaCha	data-manipulation/encryption/salsa20
encrypt data using TEA	data-manipulation/encryption/tea
hash data using djb2 (2 matches)	data-manipulation/hashing/djb2
resolve function by parsing PE exports	load-code/pe

Figure 6.8: Capa Analysis of Sample TP3

Proceeding to the false positive sample (FP1), the capa analysis results for it can be seen in Figure 6.9, with no match for the superblock RC4 PRGA rule.



Capability	Namespace
execute anti-debugging instructions	anti-analysis/anti-debugging/debugger-detection
parse credit card information	collection/credit-card
encode data using XOR (28 matches)	data-manipulation/encoding/xor
encrypt data using RC4 PRGA	data-manipulation/encryption/rc4
hash data using fnv (9 matches)	data-manipulation/hashing/fnv
interact with driver via IOCTL	host-interaction/driver
query environment variable	host-interaction/environment-variable
get common file path (4 matches)	host-interaction/file-system
check if file exists (4 matches)	host-interaction/file-system/exists
enumerate files on Windows	host-interaction/file-system/files/list
get file attributes (4 matches)	host-interaction/file-system/meta
get file size	host-interaction/file-system/meta
set file attributes	host-interaction/file-system/meta
read file on Windows	host-interaction/file-system/read
write file on Windows	host-interaction/file-system/write
print debug messages (6 matches)	host-interaction/log/debug/write-event
create or open mutex on Windows	host-interaction/mutex
get hostname	host-interaction/os/hostname
get system information on Windows	host-interaction/os/info
get thread local storage value (2 matches)	host-interaction/process
create process on Windows	host-interaction/process/create
modify access privileges	host-interaction/process/modify
query or enumerate registry key	host-interaction/registry
query or enumerate registry value (10 matches)	host-interaction/registry
set registry value (7 matches)	host-interaction/registry/create
delete registry key (2 matches)	host-interaction/registry/delete
delete registry value (2 matches)	host-interaction/registry/delete
run as service	host-interaction/service
compare security identifiers (2 matches)	host-interaction/sid
create thread	host-interaction/thread/create
allocate thread local storage	host-interaction/thread/tls
set thread local storage value (2 matches)	host-interaction/thread/tls
link function at runtime on Windows (16 matches)	linking/runtime-linking
link many functions at runtime (2 matches)	linking/runtime-linking
parse PE header (2 matches)	load-code/pe
resolve function by parsing PE exports (3 matches)	load-code/pe

Figure 6.9: Capa Analysis of Sample FP1

We summarize the results obtained in table 6.3.3, which shows that the updated RC4 PRGA capa rule that utilized the superblock scope was matched successfully on a set of three malware samples (TP1, TP2, and TP3) with slightly different implementations of the RC4 PRGA algorithm, while at the same time not matching on the *wmemcmp()* standard C method within the benign sample (FP1).

Sample Name	Has RC4?	Old Rule Match?	New Rule Match?
FP1	No	Yes	No
TP1	Yes	Yes	Yes
TP2	Yes	Yes	Yes
TP3	Yes	Yes	Yes

Table 6.2: Capa Analysis Results for The Chosen Samples

After close examining the false positive match on the sample FP1, we can see that one crucial element to the legacy RC4 PRGA rule match was the presence of 4 or more *movzx* CPU instructions in the given function. This condition as well as the addresses for the retrieved *movzx* instructions can be seen highlighted in white within Figure 6.10.

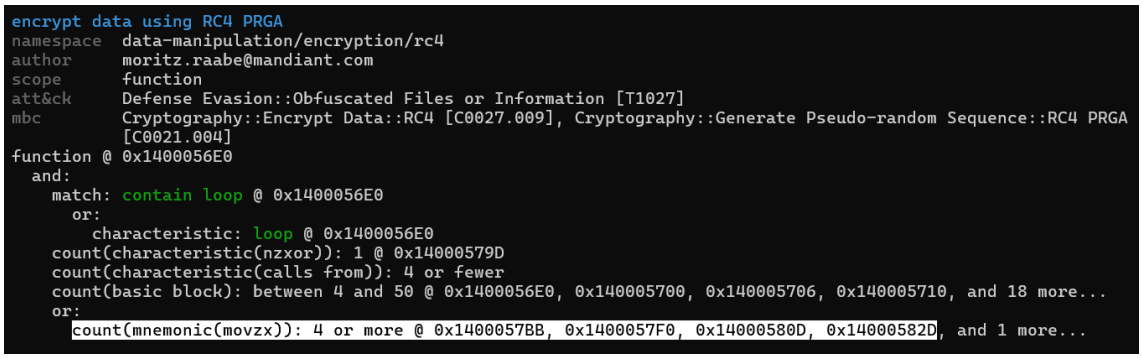


Figure 6.10: Verbose Output for The Capa Analysis of Sample FP1

By opening the FP1 sample in IDA then navigating to the location of the match (i.e., the statically-linked *wmemcmp()* method), and then rearranging all of the *movzx* CPU instructions present in the function to be adjacent to each other in the flow graph (as shown in Figure 6.11), we can see that these instructions are far apart within the function and not closely related such as the case with RC4's PRGA usually. And as a consequence of the legacy rule having the *function* scope, these

sparse *movzx* instructions are incorrectly assumed by capa to be in close relationship to each other like the RC4 PRGA necessitates, which is not the case, and we got a false positive as a result.

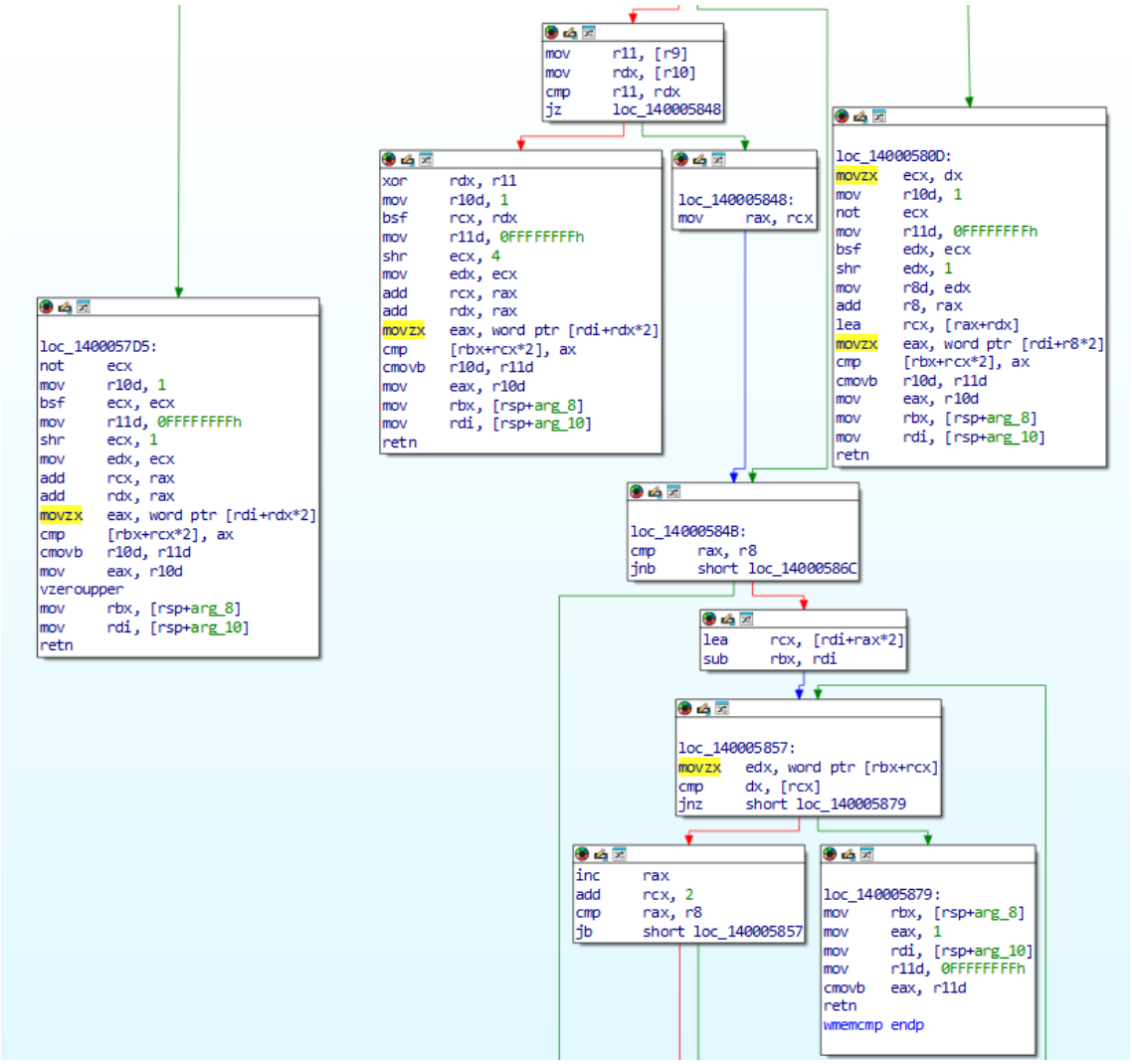


Figure 6.11: IDA Disassembly of The *wmemcmp()* Standard C Function (in FP1)

6.3.4 Conclusion

In conclusion, we have demonstrated that our newly added superblock scope gives rule authors more expressibility with regards to describing malware capabilities,

which in turns will help reduce false positives and false negatives when using the tool.

False negatives are reduced by giving rule authors a new matching scope that is not vulnerable to variation in the control flow of programs. For instance, if a superblock rule requires two Windows API calls to be made in order for it to match, and the malware author performs some error checking after the first Windows API call that exits the program on a fail, but continues to the second Windows API call on success, then that would not affect the superblock-scope rule since the basic blocks hosting each of the Windows API calls would be tied together and existing on the same superblock. This can be clearly seen in the verification example shown in section 6.2.1.

As for reducing false positives, the new superblock scope gives rule authors the ability to write capability-describing rule logic that necessitates for its constituent features to be present within small proximity (i.e., same or in-direct-sequence basic blocks) in a function. This gives rule authors the ability to write precise rules that are immune to branch instructions inserted by the compiler for either performance or error-checking reasons, without sacrificing on the rule's matching potential.

7 Conclusion

In this thesis, we have identified an area of improvement in automated static malware capability extraction in general, and in the capa tool in precise. We examined other major malware capability extraction tools in use today, and determined that none of them provide a solution to the identified issue in capa, which was the first research question of our thesis. Therefore, we proceeded with addressing the discovered issue, and providing a solution for it.

The identified area of deficiency had to do with reducing the rate of false negatives and false positives in the target system at hand, and that is by providing rule malware capability rule authors with a new scope with which they could organize and group rule matching logic.

The identified false negatives were due to existing solutions not being able to circumvent error-checking logic in input samples that might split one basic block into multiple ones, therefore resulting in the match not being discovered. As for false positives, they were primarily due to the fact that many rule authors attempted to reduce false negatives by making the scope wider which resulted in many matches being the result of false correlation.

As for the second objective of this work, we proposed, designed, implemented, and tested a working solution for a matching scope that exists between the too-narrow basic block scope, and too-wide function scope as illustrated in the previous paragraph. This scope was named the *Superblock* scope, and it was defined roughly

to be a succession of basic blocks with no interruptions or inter-basic block loops in it.

Our newly added superblock scope gave more flexibility for authoring rules, as seen in the example regarding the extraction of network-related capabilities, as well as the RC4 example. This positively answers the second research question we introduced at the start of this thesis, and that the newly-added superblock scope does indeed reduce false positives and gives rule authors more expressibility.

As for answering the third research question of this thesis, then we have showed that the existing capa rule for detecting the pseudo-random generation algorithm (PRGA) for the RC4 cryptographic algorithm suffered from a false positive, and that it was incorrectly matching at the *wmemcmp()* standard C runtime method responsible for matching whether two buffers are equal, and we have shown that using the newly-introduced superblock for that rule does indeed reduce the observed false positive by correlating features relevant for the logic together, while also retaining the true positives.

7.1 Limitations

One limitation for this work is the lack of extensive and wide-scoped testing that showcases how the superblock scope improved the accuracy of capa. Instead, and due to the effort required to assemble an extensive set of false positive and true positive rules that we could accurately test on, as well as the need to verify whether the resulting capabilities are true or false positives, we refrained from carrying out such extensive testing and settled on verifying our contribution on a small scale, and demonstrated how the results could be extrapolated to a larger dataset of samples.

While robust and wide-range testing of our contribution would make it more concrete, we note that the introduced scope could still be added into capa as is, which would provide capa rule authors with greater rule expressibility. Then, the

scope could then be tested more thoroughly as it matures and more capa false positives are identified, and more superblock scope rules are written.

7.2 Future Work

Future work that could build on this thesis might revolve around updating the existing set of capa rules to better utilize the superblock scope, and thereby minimize false positives and false negatives as much as possible.

Another avenue might be to modify our contribution to allow for basic block gaps larger than one, and to give the choice of that number to the rule author, which might provide them with more expressibility.

Additionally, it might also be worthwhile to integrate simple binary emulation into capa to help with superblock formation by predicting the likely path of execution of a binary at a given basic block.

References

[1] Internet Crime Complaint Center (IC3), *Internet crime report 2023*, https://www.ic3.gov/annualreport/reports/2023_ic3report.pdf, [Accessed 13-03-2025], 2023.

[2] Lockheed Martin, *Gaining the advantage: Applying cyber kill chain methodology to network defense*, https://www.lockheedmartin.com/content/dam/lockheed-martin/rms/documents/cyber/Gaining_the_Advantage_Cyber_Kill_Chain.pdf, [Accessed 13-03-2025], 2015.

[3] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st. USA: No Starch Press, 2012, ISBN: 1593272901.

[4] Mandiant, *M-Trends 2024 report*, https://cloud.google.com/security/resources/m-trends, [Accessed 13-03-2025], 2024.

[5] *Capa: Automatically Identify Malware Capabilities | Mandiant | Google Cloud Blog — cloud.google.com*, https://cloud.google.com/blog/topics/threat-intelligence/capa-automatically-identify-malware-capabilities/, [Accessed 12-06-2025].

[6] O. Alrawi, M. Ike, M. Pruett, *et al.*, "Forecasting malware capabilities from cyber attack memory images", in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 3523–3540, ISBN: 978-1-

939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/alrawi-forecasting.

[7] VirusTotal, *YARA - The pattern matching swiss knife for malware researchers*, https://virustotal.github.io/yara/, [Accessed 23-03-2025].

[8] VirusTotal, *Welcome to YARA's documentation!; yara 4.5.0 documentation*, https://yara.readthedocs.io/en/latest/, [Accessed 23-03-2025].

[9] Y. team, *YaraRules Project — yara-rules.github.io*, https://yara-rules.github.io/blog/, [Accessed 26-03-2025], 2015.

[10] J. Daemen and V. Rijmen, "The block cipher rijndael", in *Smart Card Research and Applications*. Springer Berlin Heidelberg, 2000, pp. 277–284, ISBN: 9783540445340. DOI: 10.1007/10721064_26.

[11] R. Rivest, *The MD5 Message-Digest Algorithm*. Apr. 1992. DOI: 10.17487/rfc1321. [Online]. Available: http://dx.doi.org/10.17487/RFC1321.

[12] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (blowfish)", in *International workshop on fast software encryption*, Springer, 1993, pp. 191–204.

[13] Intel471, *Hunting for Persistence: Registry Run Keys / Startup Folder — intel471.com*, https://intel471.com/blog/hunting-for-persistence-registry-run-keys-startup-folder, [Accessed 27-03-2025], 2021.

[14] S.-J. Hwang, A. Utaliyeva, J.-S. Kim, and Y.-H. Choi, "Bypassing heaven's gate technique using black-box testing", *Sensors*, vol. 23, no. 23, p. 9417, Nov. 2023, ISSN: 1424-8220. DOI: 10.3390/s23239417.

[15] Mandiant, *GitHub - mandiant/capa-rules: Standard collection of rules for capa: The tool for enumerating the capabilities of programs — github.com*, https://github.com/mandiant/capa-rules/, [Accessed 28-03-2025].

[16] *The Official YAML Web Site — yaml.org*, https://yaml.org/, [Accessed 28-03-2025].

[17] Mandiant, *GitHub - mandiant/capa-testfiles: Data to test capa's code and rules. — github.com*, https://github.com/mandiant/capa-testfiles, [Accessed 28-03-2025].

[18] J. Vrancken, *Detecting capabilities in malware binaries by searching for function calls*, M.Sc. thesis, Radboud University. 2022.

[19] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W.-m. W. Hwu, "Superblock formation using static program analysis", in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, Austin, Texas, USA, 1993, pp. 247–255. DOI: 10.1109/MICRO.1993.282760.

[20] MITRE, *Malware Behavior Catalog*, version 3.0, 2024. [Online]. Available: https://github.com/MBCProject/mbc-markdown.

[21] Microsoft, *Memcmp, wmemcmp | microsoft learn*, https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/memcmp-wmemcmp, [Accessed 28-05-2025], 2022.