Automatically Inferring Image Bases of ARM32 Binaries

Daniel T. Chong
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/2574

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.
AUTOMATICALLY INFERRING IMAGE BASES OF ARM32 BINARIES

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering

by

DANIEL T. CHONG
B.S.C.E., Wright State University, 2021
B.S.C.S., Wright State University, 2021

2022
Wright State University
WRIGHT STATE UNIVERSITY
GRADUATE SCHOOL

04/19/22


Junjie Zhang, Ph.D.
Thesis Director

Michael Raymer, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on Final Examination:

Junjie Zhang, Ph.D.

Meilin Liu, Ph.D.

Lingwei Chen, Ph.D.

Barry Milligan, Ph.D.
Vice Provost for Academic Affairs
Dean of the Graduate School
ABSTRACT


Reverse engineering tools rely on the critical image base value for tasks such as correctly mapping code into virtual memory for an emulator or accurately determining branch destinations for a disassembler. However, binaries are often stripped and therefore, do not explicitly state this value. Currently available solutions for calculating this essential value generally require user input in the form of parameter configurations or manual binary analysis, thus these methods are limited by the experience and knowledge of the user. In this thesis, we propose a user-independent solution for determining the image base of ARM32 binaries and describe our implementation. Our solution makes use of features present in all ARM32 binaries, utilizing statistical, structural, and semantical filtration to automatically calculate the image base value. We implemented our tool in 335 lines of Python. We tested our tool on 20 stripped binaries, and it successfully determined the image bases of each binary.
# Contents

1 Introduction .......................................................... 1  
   1.1 Motivation ....................................................... 1  
   1.2 Background ..................................................... 2  
      1.2.1 Interquartile Range Method .......................... 2  
      1.2.2 Interrupts ............................................... 3  
   1.3 Goals ........................................................... 4  
   1.4 Organization .................................................. 5  

2 Related Work ......................................................... 6  

3 Design .................................................................. 8  
   3.1 Problem Formulation ........................................... 8  
   3.2 Address Collection ............................................ 9  
   3.3 Inferring Candidate Base Addresses .................... 10  
   3.4 Statistic Filtration ............................................. 11  
   3.5 Structural Filtration ......................................... 12  
   3.6 Semantic Filtration .......................................... 13  

4 Implementation .................................................... 15  
   4.1 Disassembly Framework .................................... 15  
   4.2 IVT Parsing ...................................................... 16  
   4.3 Interquartile Range Method ............................... 17  
   4.4 Semantic Analysis ............................................ 18  

5 Evaluation ............................................................ 21  
   5.1 Dataset .......................................................... 21  
   5.2 Effectiveness .................................................... 21  

6 Discussion ............................................................ 24  

7 Conclusion ............................................................. 25  

Bibliography ............................................................ 26
## List of Figures

1. 1.1 IQR Method Example [1] ........................................... 3
2. 3.1 System Design .............................................................. 8
3. 3.2 Instruction Sizing Example ............................................. 12
4. 4.1 Interrupt Vector Table ................................................... 16
5. 5.1 ARM Memory Map [2] ..................................................... 22
List of Tables

3.1 Arduino Due Addresses ........................................... 9
5.1 Binary Statistics and System Performance ................. 23
# Listings

3.1 Statistic Filtration Pseudocode ........................................ 11
3.2 Structural Filtration Pseudocode ...................................... 13
3.3 Semantic Filtration Pseudocode ........................................ 14
4.1 IQR Implementation ...................................................... 17
4.2 Semantic Analysis ........................................................ 19
A.1 System.py ............................................................... 28
Acknowledgment

First and foremost, I want to thank my advisor, Dr. Junjie Zhang, for his invaluable advice and constant support throughout my graduate study. His considerable knowledge and willingness to teach have greatly benefited me in my research. Additionally, I am grateful to my coworker, Nathaniel Boland, for his aid in developing my research. I would also like to express my gratitude to Dr. Meilin Liu and Dr. Lingwei Chen for taking the time to evaluate my thesis. Finally, I want to thank my family and friends for their encouragement and support throughout my studies.
Abbreviations

IVT — Interrupt Vector Table
PC — Program Counter
LR — Link Register
Introduction

We propose a solution for determining the image base of a stripped ARM32 binary. Our solution collects absolute and relative addresses in a binary which are then used to create a range of potential image base values. A three-step filtering process involving statistical, structural, and semantical analysis is used to narrow down the group of candidate values. Here, we describe our motivation for this thesis. Additionally in this section, we cover background material as well as our goals and organization of this paper.

1.1 Motivation

Reverse engineering software is critical to cybersecurity, playing a large role in malware analysis and vulnerability detection. A binary’s image base (also referred to as the base address) refers to the lowest address in the code segment of memory. In other words, the image base is the point where the binary is loaded into memory. This value is essential for effective reverse engineering. Specifically, disassemblers rely on the image base to deduce information about the binary’s structure. Without the proper base address, binary’s cannot be properly loaded into emulators. Additionally, the image base is required for emulators and disassemblers to determine the correct destination of branch instructions [3]. The image base is available in binaries compiled for debugging. In binaries for releasing or firmware binaries (bin files) whose debugging information has been stripped, the image base is not stated. Thus, a method to accurately determine the base address of a binary is highly demanded.
Though a few methods have been proposed to determine the image base, they are typically limited in their capabilities. These proposed methods involve manual analysis by engineers. The effectiveness of such tools are thus limited by the experience of their users [4]. Our objective is to create a tool that is capable of automatically determining the image base of a binary without requiring any manual analysis or user input.

1.2 Background

We utilize several technical concepts in this thesis, and we describe these concepts in this section. Namely, we discuss the Interquartile Range Method which is used to identify outliers in data sets. We also discuss software interrupts which is a concept critical to the accurate identification of the image base.

1.2.1 Interquartile Range Method

In a given data set, some points may lie outside of the general range of the other points. These points are called outliers [1]. One method to identify these outliers is the Interquartile Range Method. This method calculates an upper and lower bound on a data set, and any points lying outside of these bounds are identified as outliers.

To calculate the interquartile range, IQR, the first and third quartiles must be determined. The first quartile ($Q_1$) represents the median of the first half of the list (i.e., a quarter of the way through the whole list). The third quartile ($Q_3$) represents three-quarters of the way through the list (the median of the second half of the list). The IQR is defined as the first quartile subtracted from the third quartile. Specifically, $IQR = Q_3 - Q_1$. The lower bound is defined as $Q_1 - 1.5(IQR)$, and the upper bound is set to $Q_3 + 1.5(IQR)$. Any data points outside of these bounds are considered outliers [1].
1.2.2 Interrupts

Instead of continuously checking the status of an external device, most modern processors support interrupts to enable efficient, event-driven processing. Interrupts are signals to the processor that request a change of execution due to an external event, referred to as a trigger [5]. Interrupts halt currently executing code, and the processor begins executing the interrupt handling code to process the event quickly. Most application programs execute in user mode; however, in the case of an interrupt, the processor is changed to privileged mode [6]. During an interrupt, a certain sequence of events, known as a context switch, are triggered. For ARM32, five events are triggered. First, the current instruction is completed. Second, the currently running program is suspended, and eight registers (R0, R1, R2, R3,
R12, LR, PC, and PSR) are pushed onto the stack. Register LR is then set to 0xFFFFFFFF9. The interrupt program status register, IPSR, is set to the interrupt number that was triggered. Finally, the program counter, PC, is set to the starting address of the interrupt [5].

The interrupt vector table contains addresses which designate the first instruction of each interrupt handler. In ARM32 devices, this table is located at the beginning of the binary and continues up until the first instruction [7]. Branches executed in user mode use addresses that point to instructions relative to the base address of the binary. However, the addresses stored within the interrupt vector table consider the whole memory space. In other words, interrupt handler addresses contain the image base offset.

1.3 Goals

In this thesis, we developed a tool for calculating the image base of an ARM32 stripped binary. The image base is the address at which code is mapped into memory and is essential to reverse engineering [8].

The 32-bit ARM architecture is a leading architecture that frequently supports high-profile, stripped binaries for various systems, particularly embedded and IoT systems. The ARM architecture reportedly hosts approximately 63% of embedded systems [3]. In this thesis, our system focuses on determining the image base for ARM32 binaries.

Existing methods [3, 8, 4, 9] rely on software-level heuristics. These methods are prone to inconsistent results due to variances introduced by software development and compilation. Our method must leverage intrinsic architectural features of ARM32 to avoid such shortcomings.

Current methods [3, 8, 4, 9] depend on the experience of the user as they require manual analysis and configuration. Our system must be parameter-free, requiring no user input, to prevent potential user-error issues.
1.4 Organization

Chapter 1 describes the motivation, background, and goals of this thesis. Chapter 2 discusses current work related to our system and differentiates our system. Chapter 3 describes the design of our system, and challenges that we had to overcome. Chapter 4 discusses our implementation of our system. Chapter 5 addresses our testing of the effectiveness of our system, and Chapter 6 concludes this thesis.
Related Work

Because of the critical nature of the image base, a number of methods have been proposed to calculate this value. Generally, while some methods yield promising results, they suffer from design features that limit their capabilities. In this section, we discuss a subset of these methods and how our method addresses their limitations.

Skochinskey et al. [4] proposed to leverage jump tables, string tables, and initialization code to infer the base address. Their technique suggests starting with an image base of 0 as a starting point. If 0 does not work, the user must identify structures within the binary to determine the image base through trial and error. Many times, compilers will use jump tables to implement switch statements [4]. The offsets in the jump table can point to valid code close to the indirect jump instruction. These offsets can be used to guess the value of the image base. Additionally, programs may use string tables that are typically represented by an array of offsets to strings. Subtracting these offsets produces the string lengths which can be matched against strings within the binary. If the previous methods fail, the usual steps to startup code involve copying the code for faster execution to RAM, copying initialized data from ROM to RAM’s data segment, and finally zeroing uninitialized data [4]. Engineers must identify these steps to help determine the base address of the code. This proposed method requires significant manual labor and is thus limited heavily by the expertise of the engineer.

Alternatively, Zachry Basnight [9] described a method involving the use of immediate values in instructions. This load immediate technique involves searching for all load regis-
Addressader (LDR) instructions that reference immediate values. Some image base values are more commonly used in ARM32 binaries. These common values are used in conjunction with the immediate references to attempt to guess the true image base through trial and error. This method also requires manual efforts from the engineer and is prone to error from its involved approximations.

Ruijin Zhu et al. [3] proposed an automated method to determine the base addresses of binaries compiled for the ARM architecture. This method relies on the accurate identification of function entry tables (FETs). After using the FIND-FET algorithm to identify the FETs in a binary, the FIND-BASE algorithm then attempts to calculate the image base by locating the functions referenced by the FETs. While this method has demonstrated promising results, it is limited by the presence and detectability of FETs in a binary. Additionally, the FET identification process is sensitive to manually-configured parameters, limiting its practical applicability.

Another method was proposed by Ruijin Zhu et al. [8] which was a modified, automated version of the proposal by Zachry Basnight [9]. This method automates the collection of immediate load references. This algorithm uses these load references to check every potential image base value. In the 32-bit system, this technique checks each potential image base from 0x0 to 0xFFFFFFFF-fileSize, where fileSize is the size of the binary file. This method suffers from inefficiency due to every potential value having to be checked. Also, this system depends on the successful identification of the prologue and epilogue of a function which can vary amongst compilers.

In this thesis, we discuss our image base detection system which requires no manual work from the user. Additionally, our system utilizes features intrinsic to all ARM32 binaries, therefore, it does not fail when there are slight variations in the compilation of the binary.
Design

Next, we describe the overall design of our system and the challenges we had to address to determine the image base value. We discuss the parsing method we used to collect essential data from binaries and how we determine a set of potential base addresses. We then describe the statistic, structural, and semantic filtration steps that are used to produce a final image base.

![Figure 3.1: System Design](image)

3.1 Problem Formulation

The image base is a value which is essential to successfully reverse engineering a binary. This value is not stated explicitly within a stripped binary. Though work has been done to solve this problem for the ARM 32-bit architecture, manual effort, parameter-
configuration, and reliance on potentially non-existant data structures plague these methods [3, 8, 4, 9]. In order to address these issues, we designed our system to be automated thus eliminating the potential for user error. Our system also utilizes architecture features that are present in all ARM32 binaries to avoid issues that may arise from compiler variations.

### 3.2 Address Collection

Addresses containing the image base offset are referred to as absolute addresses. The starting value of the program counter (the second dword of the binary) and all pointers contained in the IVT are absolute addresses. Addresses which are contained in branch statements are relative addresses as they do not consider the image base offset. We parse the starting program counter value, the interrupt vector table, and instructions to collect absolute and relative addresses, respectively. Table 3.1 demonstrates a collection of absolute and relative addresses from an Arduino Due binary with an image base of 0x80000 [10]. Nevertheless, the boundary between the IVT and instructions (i.e., the size of the IVT is unknown).

<table>
<thead>
<tr>
<th>Absolute Addresses</th>
<th>Relative Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x81F71</td>
<td>0x00106</td>
</tr>
<tr>
<td>0x81F73</td>
<td>0x0010A</td>
</tr>
<tr>
<td>0x81F77</td>
<td>0x00126</td>
</tr>
<tr>
<td>0x81F7B</td>
<td>0x00132</td>
</tr>
<tr>
<td>0x80B65</td>
<td>0x00148</td>
</tr>
<tr>
<td>0x81E91</td>
<td>0x00154</td>
</tr>
<tr>
<td>0x81EC9</td>
<td>0x0015C</td>
</tr>
<tr>
<td>0x81F01</td>
<td>0x00170</td>
</tr>
<tr>
<td>0x81F39</td>
<td>0x00174</td>
</tr>
<tr>
<td>0x80B79</td>
<td>0x001AC</td>
</tr>
</tbody>
</table>

In order to address this challenge, we utilize the starting program counter value (denoted as IPC) and the binary’s size (denoted as size). When mapped into the memory, the
highest possible address for an instruction in the binary is IPC + size. Similarly, the lowest possible address is IPC - size. When this binary is loaded into memory, the binary will be within the range of [IPC - size, IPC + size]. Additionally, this address range applies to every entry in the IVT because it points to the first instruction of an interrupt service routine. Consequently, our system iterates through each dword starting from 0x00000008. If a given dword is determined to be within this range, it is considered as an absolute address to an interrupt service routine, and our system continues to parse the next dword. Otherwise, the parsing is stopped and the dword is considered to be the first instruction. This step leads to a set of absolute addresses, denoted as $S_{abs}$.

Our system then continues and sequentially disassembles instructions beginning at the boundary of the IVT throughout the length of the binary. During this disassembly, it considers all branch instructions which use relative addresses as arguments. In other words, our system ignores all instructions that use a register as a branch argument. Potentially, a binary may contain instructions, including branch instructions, that are never executed at runtime, referred to as dead code. Our system must exclude these dead code branch instructions to prevent contamination of the relative dataset. This process leads to a set of relative addresses, denoted as $S_{rel}$.

### 3.3 Inferring Candidate Base Addresses

After collecting a set of absolute addresses ($S_{abs}$) and relative addresses ($S_{rel}$), our system draws a correlation between both sets to infer candidate image base addresses. The system can derive a range of possible image base addresses, which is $[\min(S_{abs}) - size, \min(S_{abs})]$. The image base cannot be lower than $\min(S_{abs}) - size$ because some addresses in $S_{abs}$ will be unreachable during runtime. Similarly, an image base larger than $\min(S_{abs})$ would make addresses in $S_{abs}$ unavailable.

Also, it is worth noting the relationship of the image base and ARM32’s memory page
conventions. Specifically, the image base address will always be aligned with the address of a memory page. For ARM32, memory pages are either 1KB or 4KB [11]. Therefore, our system makes use of the finer granularity of 1KB to define the page size (i.e., \( \text{page\_size} = 1KB \)). The system calculates a set of candidate addresses from the range of potential image bases, which is denoted as \( S_{\text{addr}} = \{\text{addr} \& \neg(\text{page\_size} - 1) \mid \text{min}(S_{\text{abs}}) - \text{size} \leq \text{addr} \leq \text{min}(S_{\text{abs}})\} \).

### 3.4 Statistic Filtration

After obtaining a set of candidate values, our system next filters out irrelevant image base addresses by performing statistical analysis. As both absolute and relative addresses are used to access the same segment of memory, when the correct image base is subtracted from the absolute addresses, \( S_{\text{abs}} \) and \( S_{\text{rel}} \) will statistically fit in one data set. Therefore, we iterate through each candidate image base address in \( S_{\text{addr}} \), applying each value to \( S_{\text{abs}} \) (producing \( S'_{\text{abs}} \)). We then apply an efficient, parameter-free statistical method, the Interquartile Range Method [1], to evaluate the closeness of \( S'_{\text{abs}} \) to \( S_{\text{rel}} \).

Listing 3.1 presents pseudocode on how our method integrates \( S_{\text{addr}}, S_{\text{rel}}, \) and \( S_{\text{abs}} \) to filter out irrelevant candidate image base addresses.

```
Listing 3.1: Statistic Filtration Pseudocode

1. Statistic-Filter(combined_set)
2.      quartile_1 = median(combined_set.first_half)
3.      quartile_3 = median(combined_set.second_half)
4.      iqr = quartile_3 - quartile_1
5.      upper_limit = quartile_3 + iqr * 1.5
6.      lower_limit = quartile_1 - iqr * 1.5
7.      for a in combined_set:
8.          if a > upper_limit or a < lower_limit:
```

11
3.5 Structural Filtration

After narrowing the list of potential image base values, our system takes advantage of the unique instruction sizing of ARM32 to further filter out candidates. ARM32 instructions can be either 1 or 2 bytes long [12]. This feature can be used to eliminate potential candidates. Figure 3.2 contains both 1 and 2 byte instructions and illustrates data taken from an ARM32 binary being organized by address, binary code, and the related disassembly.

Listing 3.2 describes how our system uses this instruction sizing. For each potential image base, our system subtracts the candidate from each absolute address to obtain the instruction pointed to by the absolute address. If the image base is incorrect, it is possible that the offset absolute address will point to the middle of a 2 byte instruction, which our
This faulty candidate will be subsequently removed from the list of potentials.

### Listing 3.2: Structural Filtration Pseudocode

```python
1. Structural-Filter(absolute, base)
   2.     for a in absolute:
   3.         address = a - base
   4.         instruction = binary_code[address]
   5.         if instruction == None:
   6.             return False
   7.     return True
8. Filter-Candidates-2()
   9.     for c in candidates:
10.        if Semantic-Filter(absolute, c) == False:
11.            candidates.remove(c)
```

### 3.6 Semantic Filtration

After the statistical and structural filters, there will likely be a small set of remaining candidate image base addresses. To filter this final set, our system performs semantic analysis with each candidate image base address. This semantic analysis leverages the nature of an interrupt event and its expected procedures. Specifically, interrupt events are unpredictable as they are generally triggered by external events. When the routine that handles the interrupt begins executing, the registers are in undetermined states. In other words, the registers will contain values from the previously executing routine, and because interrupts occur unpredictably, the registers will contain arbitrary values. Thus, it is unreasonable for a register’s value to be used during the beginning of an interrupt. Without first initializing a register, any operation by an interrupt handler using a register’s value as an argument is
considered illegal usage. Conversely, it would be valid for a routine to start by saving the registers’ states or loading memory values into the registers.

Listing 3.3 presents how our system leverages such analysis to identify the final candidate(s). Specifically, for each candidate image base offset, our system subtracts it from each absolute address derived from the interrupt vector table to get an address. It then attempts to disassemble the instruction at this address in the binary, which will be the first instruction of an interrupt routine. If the selected image base address is correct, the instruction will not attempt any illegal usage of a register. If this instruction contains the illegitimate use of any register, the selected candidate image base address will be eliminated.

Listing 3.3: Semantic Filtration Pseudocode

```python
1 Semantic-Filter(absolute, base)
2     for a in absolute:
3         address = a - base
4         interrupt_first_instruction = binary_code[address]
5         if interrupt_first_instruction.argument in register_list:
6             return False
7         return True
8
9 Filter-Candidates-3()
10     for c in candidates:
11         if Semantic-Filter(absolute, c) == False:
12             candidates.remove(c)
```
Implementation

We present an implementation to validate our approach to calculating an ARM32 binary’s image base. We describe the implementation of our tool which automatically determines the image base of a stripped ARM32 binary. Our system takes a stripped ARM32 binary as input and outputs the calculated loading point of the binary. This image base can be used with other tools (e.g., disassemblers and emulators) to aid in the reverse engineering process.

4.1 Disassembly Framework

In order to collect the relative addresses, we must first disassemble branch instructions in the binary. To accomplish this, we used the Capstone disassembly framework [13]. Capstone was chosen because it is a lightweight, well-maintained project that is widely used in other popular tools. As a result of selecting Capstone, our tool was written in Python.

To collect these branch statements, starting at the first instruction after the IVT, we sequentially disassemble the binary. A significant feature of ARM32 is that instructions can be either 1 or 2 bytes long [12]. An advantage to using Capstone is its ability to determine the size of an instruction. Using this feature, we are able to group the correct bytes together to disassemble instructions. Capstone is used to fill a Python list with each instruction’s address corresponding to the list’s index and each value in the list containing the instruction. Indices without instructions caused by the instruction sizing are marked. During the
structural filtration process, our system attempts to retrieve the instruction indexed by the offset absolute address. If no instruction is returned, the image base candidate is invalid.

4.2 IVT Parsing

To parse the interrupt vector table, the binary file is converted from its binary representation to hexadecimal using the Python library binascii [14]. The initial program counter value, which is the second dword of the binary, is first added to the list of absolute addresses. The interrupt vector table’s addresses are then added to the absolute addresses until the code segment is encountered. Figure 4.1 shows a memory dump from the beginning of a little-endian ARM32 binary. The first dword, Label 1, is the starting value of the stack pointer. Label 2 is the initial program counter value, and Label 3 is the first interrupt pointer. Values of 0 can be seen in the vector table. While 0 is an invalid address for an interrupt, it does not signify the end of the IVT. Label 4 shows the end of the IVT as it is the first non-zero value which lies outside of the memory range of the binary.

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x10000</td>
<td>0x00800000</td>
</tr>
<tr>
<td>2</td>
<td>0x10004</td>
<td>0x00000000</td>
</tr>
<tr>
<td>3</td>
<td>0x10008</td>
<td>0x00000000</td>
</tr>
<tr>
<td>4</td>
<td>0x1000C</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

Figure 4.1: Interrupt Vector Table
4.3 Interquartile Range Method

The interquartile range method [1] is used for the statistical filtration step. The IQR method is implemented as a method using the Python library numpy [15]. Illustrated in Listing 4.1, the filter first sorts the input data set. Numpy is then used to calculate the median values for the first half and second half of the data set. The interquartile range is then calculated. In accordance with the IQR method for finding outliers [1], the upper and lower limits are determined, and the data set is examined for outliers.

Dead code instructions are instructions that are never executed during runtime. In addition to being used to eliminate candidate base addresses, the interquartile range method is used to identify dead code branch instructions. Often, dead code branch instructions will lie outside of the range of executable memory. These invalid addresses will skew results if they are not identified. Our implementation of the IQR method both identifies and removes dead code branches from the relative address data set.

Listing 4.1: IQR Implementation

```python
1 def stat_filter(data_set):
2     front_half = 0
3     back_half = 0
4     data_set.sort()
5     if len(data_set)%2 == 0:
6         front_half = len(data_set)/2
7         back_half = len(data_set)/2
8     else:
9         front_half = int(float(len(data_set))/2-0.5)
10        back_half = int(float(len(data_set))/2+0.5)
11        Q1 = np.median(data_set[:front_half])
12        Q3 = np.median(data_set[back_half:])
```
IQR = Q3 - Q1
upper_limit = Q3 + IQR * 1.5
lower_limit = Q1 - IQR * 1.5

high = 0
low = 0

while (min(data_set) < lower_limit):
data_set.remove(min(data_set))
low += 1

while (max(data_set) > upper_limit):
data_set.remove(max(data_set))
high += 1

if high == 0 and low == 0:
    return True
else:
    return False

4.4 Semantic Analysis

The semantic filtration requires parsing and classifying ARM32 opcodes. Specifically, the mnemonics of a given instruction must be classified. We manually classified ARM32’s instructions in a Python module called by our main script. This module classifies mnemonics as arithmetic, comparison, branch, modify, or loading and parses the list of input arguments.

Illustrated in Listing 4.2, our system uses the module to parse the supposed starting instruction of each absolute address. Based on the mnemonic classification of the instruction, the semantic filter can determine if the instruction makes an illegal use of an uninitialized register.
def sem_filter(base, ABSOLUTE):
    for j in ABSOLUTE:
        addr = j-base-1
        instr = parser.Instruction(parser.Mnemonic(
            MEM_INSTR[addr].instr), parser.Operands(
            MEM_INSTR[addr].op))
        mnemonic_type = instr.mnemonic.mnemonic_type
        op_list = instr.operands._list
        mnemonic_name = str(instr.mnemonic.name)
        if mnemonic_type == mnemonic_type.ARITHMETIC:  # Arithmetic
            if op_list[1] in NON_PC_REGS:
                return False
            if len(op_list) > 2:
                if op_list[2] in NON_PC_REGS:
                    return False
            elif mnemonic_type == mnemonic_type.COMPARISON:  # Comparison
                for i in op_list:
                    if i in NON_PC_REGS:
                        return False
            elif mnemonic_type == mnemonic_type.BRANCH:  # Branch
                if MEM_INSTR[addr].instr in CONDITIONAL_BRANCHES:
                    return False
                for i in op_list:
                    if i in REGISTER_NAMES:
                        return False
            elif mnemonic_type == mnemonic_type.MODIFY:  # Modify
                if 'mov' in mnemonic_name and (op_list[1] in ...
                NON_PC_REGS):
                    return False
        # op_list[1] contains the arguments for the address to ...
if 'ldr' in mnemonic_name or '1s' in mnemonic_name or ...
    'eor' in mnemonic_name:
        if op_list[1] in NON_PC_REGS:
            return False
    elif mnemonic_type == mnemonic_type.LOAD_STORE: #Load/Store
        if 'str' in mnemonic_name and str(op_list[1]).replace('[','').replace(']','').replace('"','"') in ...
            NON_PC_REGS:
                print 'invalid store'
                return False
    elif mnemonic_type == mnemonic_type.STACK_CONTROL: #Stack ...
        control
            if mnemonic_name == 'pop':
                return False
        return True
Evaluation

To test the effectiveness of our methods, we evaluated our system using a set of 20 stripped binaries compiled for ARM32.

5.1 Dataset

While ARM specifies the uses for different segments of memory (Figure 5.1 [2]), the specific ARM32 implementation decides the exact allocations within these segments. Therefore, while each image base address will fall within ARM’s designated code segment, the exact value depends upon the implementation. To evaluate the effectiveness of our tool, we have collected 20 samples compiled for 20 unique ARM32 implementations. To collect these samples, we used the PlatformIO embedded development tool [16] to compile stripped binaries for the different implementations. PlatformIO is also capable of compiling ELF files which contain debugging information. We compiled ELF files along with stripped binaries in order to collect ground-truth information to confirm our results.

5.2 Effectiveness

Our system proved to be effective in our tests, as it successfully determined the image bases of all cases. Table 5.1 presents the statistics of the binary and the performance of our tool. For each microcontroller, we recorded the number of unique absolute and relative
addresses contained in the binary. We then calculated the range of potential image base addresses. As this range was narrowed by the filtration process, we recorded the number of remaining candidate values after each filter. In all 20 test cases, only one candidate remained after the final, semantical filter. All results were confirmed using the binary's corresponding ELF file.

Figure 5.1: ARM Memory Map [2]
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TI MSP-EXP432401R</td>
<td>3</td>
<td>2362</td>
<td>0x00000000 - 0x0002800</td>
<td>10</td>
<td>5</td>
<td>1 (0x00000000)</td>
<td>0.16</td>
<td>Yes</td>
</tr>
<tr>
<td>Arduino Due</td>
<td>16</td>
<td>818</td>
<td>0x007C400 - 0x0080800</td>
<td>17</td>
<td>2</td>
<td>1 (0x00800000)</td>
<td>0.07</td>
<td>Yes</td>
</tr>
<tr>
<td>NXP LPC1768</td>
<td>45</td>
<td>838</td>
<td>0x0000C00 - 0x0004000</td>
<td>13</td>
<td>1</td>
<td>1 (0x00040000)</td>
<td>0.11</td>
<td>Yes</td>
</tr>
<tr>
<td>Black STM32F407VE</td>
<td>30</td>
<td>1010</td>
<td>0x7FFD800 - 0x8001400</td>
<td>15</td>
<td>1</td>
<td>1 (0x80000000)</td>
<td>0.06</td>
<td>Yes</td>
</tr>
<tr>
<td>Black F407VG</td>
<td>30</td>
<td>1010</td>
<td>0x7FFD800 - 0x8001400</td>
<td>15</td>
<td>1</td>
<td>1 (0x80000000)</td>
<td>0.06</td>
<td>Yes</td>
</tr>
<tr>
<td>STM32duino</td>
<td>3</td>
<td>1097</td>
<td>0x7FFA800 - 0x8000000</td>
<td>22</td>
<td>16</td>
<td>1 (0x80000000)</td>
<td>0.09</td>
<td>Yes</td>
</tr>
<tr>
<td>Blue STM</td>
<td>30</td>
<td>1010</td>
<td>0x7FFD800 - 0x8001400</td>
<td>15</td>
<td>1</td>
<td>1 (0x80000000)</td>
<td>0.07</td>
<td>Yes</td>
</tr>
<tr>
<td>Olimex F103</td>
<td>18</td>
<td>940</td>
<td>0x7FFD800 - 0x8001000</td>
<td>14</td>
<td>2</td>
<td>1 (0x80000000)</td>
<td>0.05</td>
<td>Yes</td>
</tr>
<tr>
<td>Microduino32</td>
<td>18</td>
<td>499</td>
<td>0x8003400 - 0x8005000</td>
<td>7</td>
<td>1</td>
<td>1 (0x80050000)</td>
<td>0.03</td>
<td>Yes</td>
</tr>
<tr>
<td>BlackPill STM32F401CC</td>
<td>21</td>
<td>571</td>
<td>0x7FFeC00 - 0x8001000</td>
<td>9</td>
<td>1</td>
<td>1 (0x80000000)</td>
<td>0.04</td>
<td>Yes</td>
</tr>
<tr>
<td>NXP LPC1769</td>
<td>45</td>
<td>1056</td>
<td>0x00000000 - 0x0004000</td>
<td>16</td>
<td>1</td>
<td>1 (0x00040000)</td>
<td>0.07</td>
<td>Yes</td>
</tr>
<tr>
<td>Adafruit Feather</td>
<td>30</td>
<td>1010</td>
<td>0x7FFD800 - 0x8001400</td>
<td>15</td>
<td>1</td>
<td>1 (0x80000000)</td>
<td>0.06</td>
<td>Yes</td>
</tr>
<tr>
<td>ST STM32F0308DISCOVERY</td>
<td>16</td>
<td>1104</td>
<td>0x7FFDC00 - 0x8001400</td>
<td>14</td>
<td>4</td>
<td>1 (0x80000000)</td>
<td>0.10</td>
<td>Yes</td>
</tr>
<tr>
<td>Olimex Olimexino-STM32F03</td>
<td>24</td>
<td>1083</td>
<td>0x7FFD800 - 0x8001800</td>
<td>16</td>
<td>1</td>
<td>1 (0x80000000)</td>
<td>0.06</td>
<td>Yes</td>
</tr>
<tr>
<td>STM3210C-EVAL</td>
<td>2</td>
<td>333</td>
<td>0x7FEC00 - 0x8000000</td>
<td>5</td>
<td>4</td>
<td>1 (0x80000000)</td>
<td>0.02</td>
<td>Yes</td>
</tr>
<tr>
<td>STM32F103C4</td>
<td>16</td>
<td>927</td>
<td>0x7FFDC00 - 0x8001000</td>
<td>13</td>
<td>2</td>
<td>1 (0x80000000)</td>
<td>0.05</td>
<td>Yes</td>
</tr>
<tr>
<td>STM32F446RE</td>
<td>30</td>
<td>1101</td>
<td>0x7FFD800 - 0x8001800</td>
<td>16</td>
<td>1</td>
<td>1 (0x80000000)</td>
<td>0.06</td>
<td>Yes</td>
</tr>
<tr>
<td>STM32F7508</td>
<td>2</td>
<td>340</td>
<td>0x7FEC00 - 0x8000000</td>
<td>4</td>
<td>1</td>
<td>1 (0x80000000)</td>
<td>0.02</td>
<td>Yes</td>
</tr>
<tr>
<td>SparkFun SAMD51</td>
<td>11</td>
<td>528</td>
<td>0x0001800 - 0x0004000</td>
<td>10</td>
<td>2</td>
<td>1 (0x00040000)</td>
<td>0.04</td>
<td>Yes</td>
</tr>
<tr>
<td>STM32F103R4</td>
<td>16</td>
<td>927</td>
<td>0x7FFDC00 - 0x8001000</td>
<td>13</td>
<td>2</td>
<td>1 (0x80000000)</td>
<td>0.05</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Discussion

Because our system could calculate the image bases of every binary in our data set, we believe our method is an effective tool for this task. Often times, the statistical and structural filters proved sufficient for calculating the base value with the semantical analysis providing an extra layer of filtration. We only use data common to all ARM32 binaries. Therefore, our method is immune to variances in different implementations, a fact which is displayed in our evaluation results. While successful when tested against our data set, there is still room for potential future work.

Our system uses a three-step filtering system to select a final candidate image base. While in all of our tests, our system narrowed the candidates down to one value, it is possible that this algorithm could end with more than one potential candidate. Specifically, during the semantic filtration process, there could theoretically be more than one candidate value that does not result in illegal register usage in the first instructions of the interrupts. To address this issue, a more elaborate semantical analysis would have to be developed. This new analysis would require an examination of the entire interrupt handler instead of just the first instruction. We would have to maintain a record to track which registers are initialized and which registers are undefined. If at any point in the handler, a register is used before being initialized, that candidate image base would be eliminated. In this way, the chances of ending with more than one candidate value can be greatly reduced.
Conclusion

In this thesis, we described our tool which utilizes features intrinsic to all ARM32 binaries to infer image bases. Our tool requires no parameter configuration or manual analysis by the user and thus, does not rely on the user’s knowledge or experience. Our system sequentially disassembles the binaries to collect sets of relative and absolute addresses. These sets are used to determine a range in which the correct image base will be contained. Our system then applies the Interquartile Range Method to this range to filter out invalid candidates. Next, the unique instruction sizing structure of ARM32 is used to further eliminate potential values. Finally, our system performs semantic analysis on the remaining values to attempt to calculate a consensus value. Our system proved effective in our evaluation, successfully determining the image bases of 20 different binaries. Though it passed our tests, we identified an improvement that can be made to the semantical analysis. While this improvement was unnecessary in our evaluation, this enhancement would reduce the risk of our tool ending with more than one potential image base.
Bibliography


Appendix A

Program Code

Listing A.1: System.py

```python
'''
ARM 32-bit image base calculator
- Daniel Chong
'''
import sys
import struct
from capstone import *
from collections import OrderedDict
import binascii
import math
import numpy as np
import matplotlib.pyplot as plt
import instructions as prs
import time

FILE_NAME = ''
FILE_LENGTH = 0
IS_THUMB_MODE = 1
RELATIVE = []
MAX_INSTR_SIZE = 8
MD = Cs(CS_ARCH_ARM, CS_MODE_THUMB)
REGISTER_NAMES = ['r0', 'r1', 'r2', 'r3', 'r4', 'r5', 'r6', 'r7', ...
'r8', 'r9'
   , 'r10', 's1', 'r11', 'r12', 'r13', 'r14', 'r15', 'psr', ...
   'lr', 'pc', 'sp', 'ip', 'sb']
NON_PC_REGS = ['r0', 'r1', 'r2', 'r3', 'r4', 'r5', 'r6', 'r7', ...
   'r8', 'r9'
   , 'r10', 's1', 'r11', 'r12', 'r13', 'r14', 'r15', 'psr', ...
   'lr', 'sp', 'sb']
BRANCH_INSTRUCTIONS = {'b', 'bl', 'blx', 'bx', 'b.w', 'bl.w', ...
   'blx.w', 'bx.w'}
CONDITIONAL_BRANCHES = {'blgt', 'blvc', 'blcc', 'blhs', 'blmi', ...
   'blne', 'blal',
```
'blle', 'blge', 'blvs',
'blls', 'bllt', 'bllo', 'blcs', 'blhi', 'bleq', 'blpl', ...
'bgp', 'bvc', 'bcc',
'bhs', 'bmi', 'bne', 'bal', 'ble', 'bge', 'bvs', 'bls', ...
'bit', 'blo', 'bcs',
'bhi', 'beq', 'bpl', 'bxgt', 'bxvc', 'bxcc', 'bxhs', ...
'bxmi', 'bxne', 'bxal',
'bxle', 'bxge', 'bxvs', 'bxls', 'bxlt', 'bxlo', 'bxcs', ...
'bxhi', 'bxpl',
'bixgt', 'bixvc', 'bixcc', 'bixhs', 'bixmi', 'bixne', ...
'bixal', 'bixle', 'bixge',
'blxs', 'blxls', 'blxlt', 'blxlo', 'blxcs', 'blxhi', ...
'blxeq', 'blxpl',
'cbz', 'cbnz', 'blgt.w', 'blvc.w', 'blcc.w', 'blhs.w', ...
'blmi.w', 'blne.w', 'blal.w',
'blle.w', 'blge.w', 'blvs.w', 'blls.w', 'bllt.w', ...
'bilo.w', 'blcs.w', 'blhi.w', 'bleq.w',
'blpl.w', 'bgp.w', 'bvc.w', 'bcc.w', 'bhs.w', 'bmi.w', ...
'bne.w', 'bal.w', 'ble.w', 'bge.w',
'bls.w', 'blt.w', 'blv.w', 'blcs.w', 'blhi.w', ...
'bleq.w', 'blpl.w', 'bxtgt.w', 'bxvc.w',
'bxcc.w', 'bxhs.w', 'bxmi.w', 'bxne.w', 'bxal.w', ...
'bxle.w', 'bxge.w', 'bxvs.w', 'bxls.w',
'bxlt.w', 'bxlo.w', 'bxcs.w', 'bxhi.w', 'bxeq.w', ...
'bxpl.w', 'bxtgt.w', 'bxvc.w', 'bxcc.w',
'bixhs.w', 'bixmi.w', 'bixne.w', 'bixal.w', 'bixle.w', ...
'bixge.w', 'bixvs.w', 'bixls.w',
'bixlt.w', 'bxilo.w', 'bxcs.w', 'bxhi.w', 'bixeq.w', ...
'bxpl.w', 'cbz.w', 'cbnz.w'}

#Record the location of branch instructions in the binary
BRANCHES = {}
MEM_INSTR = []
STARTING_ADDRESS = ''
HEX_DATA = ''
ISR_POINTERS = []

# Takes a hex representation and returns an int
def endian_switch(val):
    return(int(tmp,16))

class instr_data(object):
    def __init__(self, instr, op):
        self.instr = instr
        self.op = op

class DisassemblerCore(object):
    def __init__(self, filename):
        global MEM_INSTR
        global BRANCHES
        self.filename = filename
        self.file_data = ''
self.starting_address = ''
self.beginning_code = ''
self.stack_top = ''
self.isr_num = 0
self.isr_table_length = 0
ISR_POINTERS = []
self.curr_mnemonic = ''
self.curr_op_str = ''
self.done = False
# Keep track of the size of the instruction (can be ...
determined by Capstone)
self.size = 0
self.subroutine_branch = []

def run(self):
    self.load_file()
    for i in range(len(self.file_data)):
        MEM_INSTR.append(0)
    self.disassemble()
disassembled_instrs = 0
    for i in range(len(MEM_INSTR)):
        if MEM_INSTR[i] != 0:
            disassembled_instrs += 1
    return True

def load_file(self):
    with open(self.filename, 'rb') as f:
        self.file_data = f.read()
f.close()
FILE_LENGTH = len(self.file_data)
HEX_DATA = binascii.hexlify(self.file_data)
# Stack top stored in first word, starting address in second
self.stack_top = endian_switch(HEX_DATA[0:8])
self.starting_address = endian_switch(HEX_DATA[8:16])
print hex(self.starting_address)
STARTING_ADDRESS = self.starting_address
if self.starting_address % 2 != 0:
    IS_THUMB_MODE = 1
else:
    IS_THUMB_MODE = 0
index = 16
while (True):
    address = endian_switch(HEX_DATA[index:index+8])
    index += 8
    if address != 0:
        if ((address % 2 == 0) or
            (address > self.starting_address + ...
             len(self.file_data)) or
            (address < self.starting_address - ...
             len(self.file_data))):
            # Weird offset because of "index+=8" and ... self.beginning_code-thumb_mode
            self.beginning_code = (index-8)/2 + 1
            break
        else:
            self.beginning_code-thumb_mode
break;
if(address != 0):
    self.isr_num += 1
if (address != 0) and (address not in ISR_POINTERS):
    ISR_POINTERS.append(address)
    self.isr_table_length += 1
# Disassemble ONE instruction

def dasm_single(self, md, code, addr):
    # Keep track of the number of instructions disassembled
    count = 0
    for(address, size, mnemonic, op_str) in md.disasm_lite(code, addr):
        count += 1
        self.curr_mnemonic = str(mnemonic)
        self.curr_op_str = str(op_str)
        instr = self.curr_mnemonic + ' ' + self.curr_op_str
        MEM_INSTR[address] = instr_data(self.curr_mnemonic, ...
        self.curr_op_str)
        if self.curr_mnemonic in BRANCH_INSTRUCTIONS or ...
        self.curr_mnemonic in CONDITIONAL_BRANCHES:
            if self.curr_op_str not in REGISTER_NAMES:
                RELATIVE.append( ...)
                int(self.curr_op_str.split('#')[1],16))
        '''dasm_single is given 4 bytes. If Capstone is only able ...
        to disassemble 1 2-byte instruction,
        the second 2 bytes of the 4 belong to the next ...
        instruction.'''
        if count == 1 and size == 2:
            return False
        else:
            return True
# https://www.capstone-engine.org/lang_python.html

def disassemble(self):
    start = (self.beginning_code - 1) * 2
    self.curr_instr = start
    self.curr_addr = self.beginning_code - IS_THUMB_MODE ... # offset for thumb
    # Section of code to be disassembled
    end_index = self.curr_instr+MAX_INSTR_SIZE
    code = HEX_DATA[self.curr_instr:end_index]
    code = code.decode('hex')
    prev_addr = 0
    while(self.curr_instr+MAX_INSTR_SIZE < len(HEX_DATA)):
        if self.dasm_single(MD, code, self.curr_addr):
            self.curr_instr += MAX_INSTR_SIZE
            self.curr_addr += 4
        else:
            self.curr_instr += MAX_INSTR_SIZE/2
            self.curr_addr += 2
            end_index = self.curr_instr+MAX_INSTR_SIZE
            code = HEX_DATA[self.curr_instr:end_index]
            code = code.decode('hex')
def stat_filter(data_set):
    front_half = 0
    back_half = 0
    data_set.sort()
    if len(data_set)%2 == 0:
        front_half = len(data_set)/2
        back_half = len(data_set)/2
    else:
        front_half = int(float(len(data_set))/2-0.5)
        back_half = int(float(len(data_set))/2+0.5)
    Q1 = np.median(data_set[:front_half])
    Q3 = np.median(data_set[back_half:])
    IQR = Q3 - Q1
    upper_limit = Q3 + IQR * 1.5
    lower_limit = Q1 - IQR * 1.5
    high = 0
    low = 0
    while(min(data_set) < lower_limit):
        data_set.remove(min(data_set))
        low += 1
        if len(data_set) == 0:
            break
    while(max(data_set) > upper_limit):
        data_set.remove(max(data_set))
        high += 1
        if len(data_set) == 0:
            break
    if high == 0 and low == 0:
        return True
    else:
        return False

def struc_filter(min_imagebase, max_imagebase):
    potential_bases = []
    test_base = min_imagebase
    while test_base < max_imagebase + 1024:
        bad_base = False
        for i in ISR_POINTERS:
            if i-test_base-1 >= len(MEM_INSTR) or i-test_base-1 < 0:
                bad_base = True
                break
            if MEM_INSTR[i-test_base-1] == 0:
                bad_base = True
                break
        if bad_base == False:
            potential_bases.append(test_base)
        test_base += 1024
    print 'struc filter ', len(potential_bases), potential_bases
    return potential_bases

def find_imagebase(data_set, confirmed_addrs):
test_imagebase = 0x00000
iteration = 0
page_size = 1024
min_imagebase = (min(confirmed_addrs) - FILE_LENGTH) & ¬(page_size-1)
if min_imagebase < 0:
    min_imagebase = 0
iteration = min_imagebase/page_size
max_imagebase = min(confirmed_addrs) & ¬(page_size-1)
tmp = [i - page_size*iteration for i in confirmed_addrs]
result = ''
tmp = [i - page_size*iteration for i in confirmed_addrs]
result = stat_filter(tmp+data_set)
print('range', hex(min_imagebase), hex(max_imagebase))
while result != True:
    iteration += 1
    tmp = [i - page_size*iteration for i in confirmed_addrs]
    result = stat_filter(tmp+data_set)
    print('stat filter', (max_imagebase-min_imagebase)/1024, ...
    hex(min_imagebase), hex(max_imagebase), iteration)
    min_imagebase = iteration*page_size
    return struc_filter(min_imagebase, max_imagebase)
def sem_filter(base, ABSOLUTE):
    for j in ABSOLUTE:
        addr = j-base-1
        instr = prs.Instruction(prs.Mnemonic(
            MEM_INSTR[addr].instr),prs.Operands(MEM_INSTR[addr].op))
        mnemonic_type = instr.mnemonic.mnemonic_type
        op_list = instr.operands._list
        mnemonic_name = str(instr.mnemonic.name)
        if mnemonic_type == mnemonic_type.ARITHMETIC: #Arithmetic
            if op_list[1] in NON_PC_REGS:
                #print 'arithmetic'
                return False
            if len(op_list) > 2:
                if op_list[2] in NON_PC_REGS:
                    return False
            elif mnemonic_type == mnemonic_type.COMPARISON: #Comparison
                for i in op_list:
                    if i in NON_PC_REGS:
                        #print 'comp'
                        return False
            elif mnemonic_type == mnemonic_type.BRANCH: #Branch
                if MEM_INSTR[addr].instr in CONDITIONAL_BRANCHES:
                    print 'bran'
                    return False
                for i in op_list:
                    if i in REGISTER_NAMES:
                        #print 'bran'
                        return False
            elif mnemonic_type == mnemonic_type.MODIFY: #Modify
                #special case
                return False
        elif mnemonic_type == mnemonic_type.MODIFY: #Modify
            return False
        else: #special case
            return False
if 'mov' in mnemonic_name and (op_list[1] in ...
NON_PC_REGS):
    #print 'mod'
    return False
#op_list[1] contains the arguments for the address to ...
load from
if 'ldr' in mnemonic_name or 'ls' in mnemonic_name or ...
'eor' in mnemonic_name:
    if op_list[1] in NON_PC_REGS:
        #print 'mod'
        return False
elif mnemonic_type == mnemonic_type.LOAD_STORE: #Load/Store
    #special case
    if 'str' in mnemonic_name and str(op_list[1]).replace("
    '[','').replace(']',',').replace('"','') in ...
    NON_PC_REGS:
        #print 'load/store'
        return False
elif mnemonic_type == mnemonic_type.STACK_CONTROL: #Stack ...
    control
    #special case
    if mnemonic_name == 'pop':
        #print 'stack'
        return False
    return True

#0x40000000 is the max number where code can be stored
# Main
def main():
tmp = False
    if len(sys.argv) > 1:
        FILE_NAME = str(sys.argv[1])
        with open('startup.txt', 'w') as f:
            f.write(FILE_NAME)
            f.close()
    else:
        with open('startup.txt', 'r') as f:
            FILE_NAME = f.readline()
            f.close()
        if len(FILE_NAME) == 0:
            print('No file found')
            return True
        dc = DisassemblerCore(FILE_NAME)
dc.run()

RELATIVE.sort()
stat_filter(RELATIVE)
ABSOLUTE = ISR_POINTERS
ABSOLUTE.append(STARTING_ADDRESS)
print 'Absolute ' + str(len(ABSOLUTE))
print 'Relative ' + str(len(RELATIVE))
potential_bases = find_imagebase(RELATIVE, ABSOLUTE)
print 'Potential bases'
for i in potential_bases:
    print hex(i)

filtered = []
for i in potential_bases:
    if sem_filter(i, ABSOLUTE) == True:
        filtered.append(i)

print '\n\nFINAL:'
for i in filtered:
    print hex(i), MEM_INSTR[STARTING_ADDRESS-i-1].instr, ...
        MEM_INSTR[STARTING_ADDRESS-i-1].op

if __name__ == '__main__':
    startTime = time.time()
    main()
    print('Execution time in seconds: ' + str(time.time()-startTime))