Automatically Inferring Image Base Addresses of ARM32 Binaries Using Architecture Features

Daniel Chong(✉), Junjie Zhang, Nathaniel Boland, and Lingwei Chen

Department of Computer Science and Engineering
Wright State University
Dayton 45435, USA
{`chong.8, junjie.zhang, boland.5, lingwei.chen`}`@wright.edu`

Abstract. We designed an innovative method, namely *iBase*, which automatically infers the image base address of an ARM32 binary by statistically, structurally, and semantically correlating the absolute and the relative addresses contained in the binary. *iBase* exploits ARM32's architecture features, and hence it is immune to variances introduced by software development and compilation. In addition, *iBase* is parameter-free and it requires no manual configuration. We implemented *iBase* and performed evaluation using 20 ARM32 binaries. Our evaluation results have shown that *iBase* successfully detects base addresses for all of them and outperforms start-of-the-art tools including `Ghidra` and `Radare2`.

Keywords: Reverse Engineering; Image Base Address; Microcontrollers; Binary Analysis; Embedded Systems

1 Introduction

Software reverse engineering occupies a critical role in mitigating threats to the Internet of Things (IoT). Specifically, it enables a variety of security tasks such as malware analysis, vulnerability detection, and binary optimization. Effective software reverse engineering usually relies on the correct identification of the image base address (a.k.a., the base address) of an analyzed binary, which refers to the lowest virtual address from which this binary is loaded into the memory. Image base addresses are indispensable to disassemble and emulate binaries, which, for example, will need them to determine the correct destination addresses of branch instructions. While the base address is readily available for a binary compiled for debugging, it is commonly concealed in a binary for release, especially for firmware binaries [1, 2] used for IoTs, whose auxiliary information has been extensively stripped away to reduce their footprints in memory and ROM. Therefore, it is a highly-demanded but challenging task to accurately identify the image base addresses of IoT firmware binaries.

A few attempts [3–5] have been made towards this objective. Specifically, Skochinskey et al. [3] proposed to leverage jump tables, string tables, and initialization code to infer the base address. Unfortunately, this method requires a reverse engineer to manually identify one or more of these structures within the binary. Alternatively, Zachry Basnight et al. [4] proposed to use immediate values in instructions. The immediate values are offset by candidate image base values. If the offset value points to a structure such as the beginning of a string, it is considered to be a match. Therefore, a high match percentage indicates a high likelihood of the candidate being the correct image base. Nevertheless, this method requires the candidate image addresses to be manually identified and it requires significant time to complete the analysis. Zhu et al. [5, 6] devised an automated method to determine the base addresses of binaries compiled to the ARM architecture, which reportedly host approximately 63% of embedded systems. Unfortunately, this method relies on the accurate identification of function entry tables (FETs) [5] or function entry addresses [6], which limit its practical applicability. On the one hand, the method [5] is inapplicable to binaries without FETs and those with undetectable FETs; the FET identification process used by this method is sensitive to manually-configured parameters. On the other hand, compiler optimizations and architecture variety makes it challenging to detect function entry addresses.

In this paper, we present an innovative method, namely *iBase*, which automatically infers the image base address of an ARM32 binary. Same as existing work [3–6], *iBase* focuses on binaries compiled for the 32-bit ARM architecture (a.k.a ARM32), a leading target architecture used by an enormous number of high-profile binaries. However, fundamentally different from these existing methods that rely on software-specific heuristics, *iBase* exploits architecture features that are inherent to all ARM32 binaries. As a consequence, *iBase* is immune to variances introduced by software development and compilation. In addition, *iBase* is parameter-free and it requires no manual configuration. We have implemented *iBase* in Python and performed evaluation using 20 real-world ARM32 binaries. Our evaluation results have demonstrated that *iBase* successfully detects base addresses for all of them and outperforms start-of-the-art tools including `Ghidra` [7] and `Radare2` [8]. To summarize, our work has made the following novel contributions.

– We have designed a novel, parameter-free method to automatically infer image base addresses of stripped ARM32 binaries.
– We have implemented a system, namely *iBase*, and we plan to publish its source code once the paper is accepted.
– We have evaluated our system using 20 binaries and accomplished a high accuracy with negligible performance overhead.

The rest of the paper is organized as follows. Section 2 discusses the related work. The system design is presented in Section 3. Section 4 organizes the evaluation results including effectiveness and performance. Section 5 discusses the potential future work and Section 6 concludes.

2 Related Work

Image bases are essential for reverse engineering [9]. They are indispensable for disassembling [10] and emulating [11] stripped binaries, exemplified by those binaries extracted from IoT devices. A few methods [3–5] have been proposed. Skochinskey et al. [3] use four software features including self-relocating code, the initialization code, jump tables, and string tables. Unfortunately, while parts of this method can be automated, this method relies heavily upon the engineer's ability to identify these code structures and does not yield a final, definitive value.

Basnight et al. described another method for image base determination called the "load immediate instructions trick" [4]. In this method, all load register (LDR) instructions referencing immediate (i.e. absolute) values are collected. These references are then offset by candidate image base addresses to see which candidate correctly aligns the references with target data such as strings or function entries. A fundamental limitation of this method is that a reverse engineer needs to manually determine the candidates of image addresses. In addition, the "load immediate instructions" technique is a brute force method and requires significant time to complete.

Zhu et al. proposed an automated detection method that relies on the identification of function entry tables (FETs) to determine the image base of a binary file [5]. It first identifies FETs. Then the lowest and highest addresses contained in the FET are used in conjunction with the size of the binary file to calculate the range of the image base. The algorithm sequentially applies each candidate image base value to the FET addresses and attempts to match the function prologue with the newly-offset address. The number of matched function addresses is recorded. The candidate address where the percentage of matched function entry addresses exceeds a threshold is considered the true image base. However, despite the fact that this method is automated and its detection result is conclusive, it has a few fundamental limitations. First, if the source code's author does not implement an array of function pointers, there will not be FETs present within the binary, and the algorithm will fail. Additionally, the threshold which determines the success or failure of a candidate image base must be configured by the user. Therefore, the user must interpret the results of this algorithm and use these interpretations to modify the threshold parameter to correctly arrive at a final value. Zhu et al. also proposed an alternative solution [6] to detect base address using function address rather than FETs. Nevertheless, similar to [5], this method leverages software heuristics. Specifically, it is based on the observation that ARM binaries often load function addresses into registers with the `LDR` instruction. it also needs to assess whether a candidate base address results in points to correct function prologues. Unfortunately, these software heuristics can be easily subverted by compiling optimization or obfuscation [12]. For example, a compiled binary does not have to use the `LDR` instruction to load function addresses, and therefore this method will not be able to obtain a data set of addresses; a binary may not necessarily follow a specific function

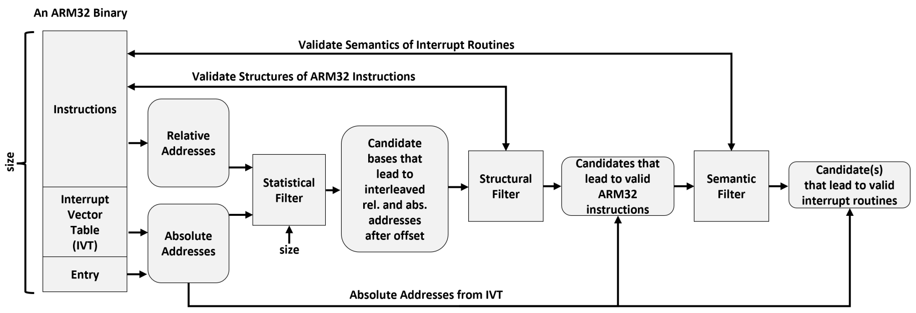
An ARM32 Binary
Validate Semantics of Interrupt Routines
Validate Structures of ARM32 Instructions
size
Instructions
Interrupt Vector Table (IVT)
Entry
Relative Addresses
Absolute Addresses
Statistical Filter
size
Candidate bases that lead to interleaved rel. and abs. addresses after offset
Structural Filter
Candidates that lead to valid ARM32 instructions
Semantic Filter
Candidate(s) that lead to valid interrupt routines
Absolute Addresses from IVT

Fig. 1. The Architectural Overview of *iBase*

prologue pattern [13], making it extremely challenging to detect functions' entry addresses.

3 System Design

iBase focuses on inferring the base address by statistically, structurally, and semantically correlating the absolute and the relative addresses contained in an ARM32 binary, where these addresses are partially manifested by ARM32's architecture specifications. Specifically, *iBase* exploits the following three of ARM32's architectural features:

- Entry Point: the entry point, a.k.a the initial value of the program counter, is the address of the first instruction to be executed. The entry point of an ARM32 binary is stored in the second dword of this binary (i.e, with the address of `0x00000004`), where a dword contains 4 bytes. The entry point is an absolute address.
- Interrupt Vector Table (IVT): the IVT immediately follows the second dword and starts from `0x00000008`. This table is a list of addresses for all of the interrupt service routines. All these addresses are absolute addresses. Yet, the size of the IVT is unknown.
- Instruction Section: the instruction section, which is a sequence of instructions, immediately follows IVT. The starting address of the instruction section is not manifested in the binary. Since IVT's size is unknown, the starting address of the instruction section is also unknown. Both absolute and relative addresses are used in this section, but they are distinguishable based on their associated instructions.

The architectural overview of *iBase* is presented in Figure 1. Specifically, *iBase* derives both absolute and relative addresses from an ARM32 binary. It then infers candidate image base addresses by statistically correlating absolute and relative addresses. *iBase* further filters out irrelevant candidate image base addresses using structural analysis and finally detects the actual image base address using lightweight semantic analysis.

3.1 Retrieving Absolute and Relative Addresses

Both the entry point and addresses contained in the IVT are absolute addresses. Comparatively, certain instructions such as branch statements in the instruction section use relative addresses. We can therefore efficiently parse the entry point, the interrupt vector table, and branch statements to retrieve absolute and relative addresses. Nevertheless, the IVT's size varies for different binaries but it is not manifested in the binary. Therefore, the ending address of the IVT, and thus the starting address of the instruction section, are unknown.

In order to address this challenge, we leverage the entry point (denoted as *entry*) and the size of the binary (denoted as *size*). We use S_{abs} to denote the set of absolute addresses, where $S_{abs} = \{entry\}$. Specifically, after a binary is loaded into the memory, the highest possible address of a byte in the binary is $entry + size$; similarly, the lowest possible one is $entry - size$. In other words, when this binary is loaded into the memory, it will reside in the range of $R_{mem} = [entry - size, entry + size]$. It is worth noting that this range is also applicable to absolute addresses in IVT. *iBase* therefore enumerates each dword denoted as dw starting from `0x00000008`. If dw resides in the range (i.e., $entry - size \leq dw \leq entry + size$), it will be identified as an absolute address in IVT and added to $S_{abs} = S_{abs} \cup \{dw\}$, and *iBase* will proceed to the next word. When *iBase*'s enumeration reaches the first dword in the instruction, because the literal value of an instruction has no intrinsic meaning, the dw value is highly likely to exceed the upper bound of this range (i.e., $dw > entry + size$). In other words, a binary value indicating an address must fall within a certain range because the literal value of the dword is an address; however, a binary value indicating an instruction is uninhibited by this constraint as its literal value has no meaning. Therefore, when *iBase* encounters a dw where $dw > entry + size$, it stops the enumeration and considers this dword as the first dword of the instruction section. This enumeration process will lead to a set of absolute addresses, which is denoted as S_{abs}.

Once the enumeration stops, *iBase* starts to disassemble instructions. The disassembling continues sequentially until either the end of the binary is reached or a disassembling attempt fails. *iBase* only concerns branch instructions that use relative addresses as arguments, ignoring those using registers as arguments. This step will lead to a set of relative addresses, denoted as S_{rel}.

3.2 Inferring Candidate Base Addresses

After *iBase* identifies S_{abs} and S_{rel}, it will leverage them to derive the range of the image base addresses, denoted as R_{base}. We denote the minimal and the maximal values in S_{abs} as $min(S_{abs})$ and $max(S_{abs})$, respectively. We will then have $base + size \geq max(S_{abs})$. This indicates that when the binary is loaded into the memory with the image base address of *base*, the maximal absolute address can never exceed $base + size$. Similarly, we will have $min(S_{abs}) \geq base$ to ensure $min(S_{abs})$ is accessible when the binary is loaded. Therefore, we can infer that $base \in R_{base} = [max(S_{abs}) - size, min(S_{abs})]$.

Given the range of *base*, we can further derive a set of candidate *base* values. Our method takes advantage of ARM32's two architecture specifications, where i) the image base address is always aligned with the address of a memory page [14] and ii) the page size is typically 1KB or 4KB. Therefore, we use a finer granularity of 1KB as the page size (i.e., $page\_size = 1\text{KB}$). Specifically, *iBase* derives a set of candidate image bases as $S_{base} = \{addr \,\&\, \neg(page\_size - 1) \mid max(S_{abs}) - size \leq addr \leq min(S_{abs})]\}$.

3.3 Statistically Filtering

iBase next filters out invalid image base values from S_{base} using statistical analysis. Such analysis leverages our observation that absolute addresses, after the correct image base address is subtracted, should be interleaved with relative addresses. Specifically, we can use a candidate image, say $base_i \in S_{base}$, and then subtract it from all absolute addresses in S_{abs}, resulting in S'_{abs}. If $base_i$ is likely to be a reasonable image base, values in S'_{abs} should be very close to those in S_{rel} (i.e., values from these two sets are interleaved); otherwise, values from S'_{abs} and S_{rel} are likely to disperse, introducing outliers.

In order to quantitatively assess this observation, we have leveraged interquartile range (IQR) [15], an efficient, parameter-free method to statistically detect outliers. Algorithm 1.1 presents how *iBase* integrates S_{base}, S_{rel}, and S_{abs} to filter out irrelevant values from S_{base}. Our algorithm joins S'_{abs} and S_{rel} and then applies IQR to detect outliers. Specifically, if an outlier is detected, S'_{abs} and S_{rel} are not considered to be sufficiently interleaved and the corresponding $base_i$ is discarded; otherwise, S'_{abs} and S_{rel} are considered to be sufficiently close to each other and the corresponding $base_i$ is preserved.

```
Validate-Distance(combined_set):
    quartile_1 = median(combined_set.first_half)
    quartile_3 = median(combined_set.second_half)
    iqr = quartile_3 - quartile_1
    upper_limit = quartile_3 + iqr * 1.5
    lower_limit = quartile_1 - iqr * 1.5
    for a in combined_set:
        if a > upper_limit or a < lower_limit:
            return False
    return True

Statistic-Filter(candidates, abs, rel):
    for c in candidates:
        combined_set = rel
        for r in abs:
            combined_set.append(r - c)
        if Statistic-Filter(combined_set) == False:
            candidates.remove(c)
    return candidates
```

Listing 1.1. Statistically Filtering

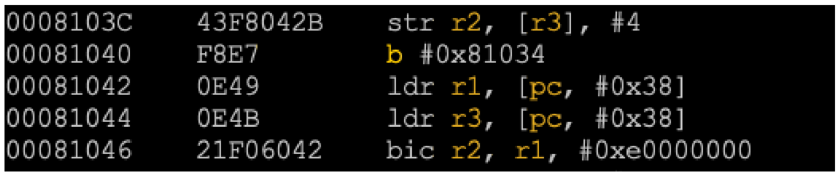
```
0008103C     43F8042B     str r2, [r3], #4
00081040     F8E7         b #0x81034
00081042     0E49         ldr r1, [pc, #0x38]
00081044     0E4B         ldr r3, [pc, #0x38]
00081046     21F06042     bic r2, r1, #0xe0000000
```

Fig. 2. A disassembled ARM32 binary snippet whose instruction size is 1 or 2 bytes

3.4 Structurally Filtering

After removing invalid candidates of base addresses using the statistical filter, *iBase* conducts additional filtering using ARM32's instruction sizes. Specifically, instructions in ARM32 can only be 1 or 2 bytes long [16]. Figure 2 presents a disassembled ARM32 binary snippet, whose instructions are either 1 or 2 bytes.

Our algorithm of structurally filtering is presented in Listing 1.2. Specifically, *iBase* offsets each absolute address (i.e., abs_j) by a candidate base (i.e., $base_i$). This results in an address (i.e., $abs_j - base_i$) that should point to an instruction in the binary. If this candidate base, $base_i$, is correct, $abs_j - base_i$ should always point to the beginning of a 2-byte instruction or that of a 1-byte instruction. Therefore, if $abs_j - base_i$ points to the middle of a 2-byte instruction, $base_i$ will be considered as an invalid candidate base and hence filtered out.

```
Validate-Inst(absolute, base):
    for a in absolute:
        address = a - base
        instruction = binary_code[address]
        if instruction == None:
            return False
        return True


Structural-Filter(candidates, absolute):
    for c in candidates:
        if Validate-Inst(absolute, c) == False:
            candidates.remove(c)
    return candidates
```

Listing 1.2. Structurally Filtering

3.5 Semantically Filtering

iBase next performs lightweight semantic analysis for each candidate image base address that survives both the statistical and structural filters. It leverages the nature of an interrupt event and the expected semantics to process it. Specifically, an interrupt event is unpredictable since it is typically triggered by an external event. In the event of an interrupt, a context switch is triggered. For an ARM32 system, the context switch involves five steps. First, the current instruction finishes executing. Next, registers R0, R1, R2, R3, R12, the link register (LR), the program counter (PC), and the program state register (PSR) are pushed onto the stack. Third, LR is set to a special value which tells the processor how many values to pop from the stack when returning to normal execution. Fourth, the interrupt program state register (IPSR) is set to the number of the triggered interrupt. Finally, the PC is loaded with the interrupt vector [17].

Notably, while the register values are saved, the registers themselves are not initialized to specific values. Therefore, when the routine that handles this interrupt begins, registers are in undetermined states (e.g., they can contain arbitrary values written by other routines). In other words, it is unreasonable to directly read a register, particularly a general purpose input/output (GPIO) register, since its value can be arbitrary. Specifically, it would be illegitimate if an instruction uses a register for arithmetic, comparison, and addressing without firstly loading its value from the memory. Comparatively, it would be valid if the routine starts with pushing registers' values onto the stack (i.e., to restore registers after this routine completes). ARM32 supports 232 possible machine instructions. Therefore, it is expensive to track system states after the execution of each instruction and then determine an illegitimate instruction. *iBase* mitigates this challenge by only investigating the first instruction in the routine.

Specifically, *iBase* parses the first instruction in the routine and extracts its opcode. It classifies an instruction as arithmetic, comparison, branching, modifying, loading (or storing), or stack control based on the opcode extracted from this instruction. For example, an instruction will be considered an arithmetic instruction if its opcode is `adc`, `add`, `mul`, `rsb`, `rsc`, `sbc`, `sdiv`, `udiv`, or `sub`; one will be considered as a comparison instruction if its opcode is `cmp` or `cmn`; it will be considered as a branch instruction if its opcode is `b`, or `cbz`. If the first instruction in the routine is an arithmetic, comparison, branching, modifying, or loading (or storing) instruction and it uses a register as its operand, it will be considered illegitimate; if the first instruction in the routine is a stack control instruction but it is not `push`, it will be considered illegitimate.

Algorithm 1.3 presents how *iBase* leverages such analysis to identify the final candidate(s). Specifically, for each candidate base address (i.e., $base_i$), *iBase* subtracts it from each absolute address (i.e., abs_j) derived from the interrupt vector table to get an offset value (i.e., $abs_j - base_i$). It then attempts to disassemble the instruction with this offset in the binary, which will be the first instruction of an interrupt routine if the selected image base address is correct. If this instruction contains the illegitimate use of any register, the selected candidate image base address will be eliminated.

```
Validate-Prologue(absolute, base):
    for a in absolute:
        address = a - base
        interrupt_first_instruction = binary_code[address]
        if is_illegal_reg_usage(interrupt_first_instruction):
            return False
        else:
            return True

Semantic-Filter(candidates, absolute):
    for c in candidates:
        if Validate-Prologue(absolute, c) == False:
            candidates.remove(c)
    return candidates
```
Listing 1.3. Semantically Filtering

4 Evaluation

Since the manufacturer of a microcontroller determines the exact value of the image base for a binary to be executed in this microcontroller, we have used PlatformIO, an embedded development tool [18] that is capable of compiling binaries for different microcontrollers. We developed a program and compiled its source code into both a stripped binary and a binary of executable and linking format (ELF) for each microcontroller: the ELF binary contains the image base and therefore offers the ground-truth information; the stripped binary does not have such information but is commonly encountered for reverse engineering. We totally generate 20 elf-stripped pairs, where each pair of binaries are for an ARM32 microcontroller with a distinct manufacturer. We use *iBase* to infer the image base of a stripped binary and validate the result using the ground-truth image base derived from its ELF counterpart. Since none of the existing methods [3–6] publishes its tool, we cannot experimentally compare them with *iBase*. We also exclude `IDA` since its free version only works with `PE` files and not stripped binaries (i.e., `.bin` files). We compare *iBase* with two state-of-the-art reverse engineering tools including `Ghidra` and `Radare2`. The experiments were conducted on a workstation with Intel Core i7 with 32 GB RAM.

Table 4 presents the evaluation results, including the number of absolute addresses, the number of relative addresses, the inferred range of the image base, the number of image base candidates after the statistical, structural, and semantic filters, the running time measured in seconds, and the validation result using the available ground truth. The detection results of `Ghidra` and `Radare2` are presented in the last two columns. As shown by the evaluation results, *iBase* has successfully detected image base addresses in all tested binaries, with a negligible running time of less than 1 second. Comparatively, `Ghidra` and `Radare2` fail to detect all of them.

Table 1. Detection Accuracy and System Performance

Microcontroller	Abs.	Rel.	Range of Base Addresses	Stat.	Struc.	Semantic	Runtime (s)	*iBase*	*Ghidra*	*Radare2*
TI MSP-EXP432401R	3	2362	0x0000000 - 0x0002800	10	5	1 (0x0000000)	0.16	Yes	No	No
Arduino Due	16	818	0x007C400 - 0x0080800	17	2	1 (0x0080000)	0.07	Yes	No	No
NXP LPC1768	45	838	0x0000C00 - 0x0004000	13	1	1 (0x0004000)	0.11	Yes	No	No
Black STM32F407VE	30	1010	0x7FFD800 - 0x8001400	15	1	1 (0x8000000)	0.06	Yes	No	No
Black F407VG	30	1010	0x7FFD800 - 0x8001400	15	1	1 (0x8000000)	0.06	Yes	No	No
STM32duino	3	1097	0x7FFA800 - 0x8000000	22	16	1 (0x8000000)	0.09	Yes	No	No
Blue STM	30	1010	0x7FFD800 - 0x8001400	15	1	1 (0x8000000)	0.07	Yes	No	No
Olimex F103	18	940	0x7FFD800 - 0x8001000	14	2	1 (0x8000000)	0.05	Yes	No	No
Microduino32	18	499	0x8003400 - 0x8005000	7	1	1 (0x8005000)	0.03	Yes	No	No
BlackPill STM32F401CC	21	571	0x7FFeC00 - 0x8001000	9	1	1 (0x8000000)	0.04	Yes	No	No
NXP LPC1769	45	1056	0x0000000 - 0x0004000	16	1	1 (0x0004000)	0.07	Yes	No	No
Adafruit Feather	30	1010	0x7FFD800 - 0x8001400	15	1	1 (0x8000000)	0.06	Yes	No	No
ST STM32F0308DISCOVERY	16	1104	0x7FFDC00 - 0x8001400	14	4	1 (0x8000000)	0.10	Yes	No	No
Olimex Olimexino-STM32F03	24	1083	0x7FFD800 - 0x8001800	16	1	1 (0x8000000)	0.06	Yes	No	No
STM3210C-EVAL	2	333	0x7FFEC00 - 0x8000000	5	4	1 (0x8000000)	0.02	Yes	No	No
STM32F103C4	16	927	0x7FFDC00 - 0x8001000	13	2	1 (0x8000000)	0.05	Yes	No	No
STM32F446RE	30	1101	0x7FFD800 - 0x8001800	16	1	1 (0x8000000)	0.06	Yes	No	No
STM32F7508	2	340	0x7FFEC00 - 0x8000000	4	1	1 (0x8000000)	0.02	Yes	No	No
SparkFun SAMD51	11	528	0x0001800 - 0x0004000	10	2	1 (0x0004000)	0.04	Yes	No	No
STM32F103R4	16	927	0x7FFDC00 - 0x8001000	13	2	1 (0x8000000)	0.05	Yes	No	No

5 Discussion

Although the interquartile range (IQR) method is effective for *iBase* to perform statistical filter, one can use other methods [19, 20] to perform outlier detection. While *iBase* currently only examines the first instruction of an interrupt handler to perform semantically filtering, experiments have shown that it has been entirely sufficient to determine the eligibility of the candidate base. Nevertheless, only can extend this filter by emulating more than one instruction (e.g., by performing taint analysis) in the function prologue to decide whether a register is illegitimately used. The advantage of this system is its leverage of intrinsic ARM32 features which enables it to avoid being constrained by the specifics of source code implementation and to be independent of compilers. While *iBase*is bound to ARM32 binaries, each of its components can be extended to other architectures. Specifically, the entry point, the interrupt vector table, and interrupt routines are almost pervasive for all binaries and particularly those for embedded systems. The next step of our work is to extend *iBase* to support ARM64 and integrate it into `Ghidra`.

6 Conclusion

iBase automatically infers the image base address of an ARM32 binary by statistically, structurally, and semantically correlating the absolute and the relative addresses contained in the binary. Since it exploits ARM32's architecture features, *iBase* is immune to variances introduced by software development and compilation. This design distinguishes *iBase* from existing methods that rely on manual configuration, software features, or both. Our evaluation results have demonstrated that *iBase* successfully detects base addresses for all of tested binaries and outperforms start-of-the-art tools including `Ghidra` and `Radare2`.

References

1. Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 95–110, 2014.
2. Zachry H Basnight. Firmware counterfeiting and modification attacks on programmable logic controllers. Technical report, AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH GRADUATE SCHOOL OF . . . , 2013.
3. Igor Skochinsky. Intro to embedded reverse engineering for pc reversers. In *REcon conference, Montreal, Canada*, 2010.
4. Zachary Basnight. Firmware counterfeiting and modification attacks on programmable logic controllers. Master's thesis, Air Force Institute of Technology, March 2013.
5. Ruijin Zhu, Yu-an Tan, Quanxin Zhang, Fei Wu, Jun Zheng, and Yuan Xue. Determining image base of firmware files for arm devices. *IEICE Transactions on Information and Systems*, E99-D(2):351–359, February 2016.

6. Ruijin Zhu, Baofeng Zhang, Yu-an Tan, Yueliang Wan, and Jinmiao Wang. Determining the image base of arm firmware by matching function addresses. *Wireless Communications and Mobile Computing*, 2021, 2021.
7. Chris Eagle and Kara Nance. *The Ghidra Book: The Definitive Guide*. no starch press, 2020.
8. Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software*, 163:110538, 2020.
9. Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.
10. Yidong Du, Wenshuo Wang, Zhigang Wang, Hua Yang, Haitao Wang, Yinghao Cai, and Ming Chen. Learning symbolic operators: A neurosymbolic solution for autonomous disassembly of electric vehicle battery. *arXiv preprint arXiv:2206.03027*, 2022.
11. Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security Symposium*, pages 1099–1114, 2019.
12. Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.
13. Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.
14. Arm paging. https://wiki.osdev.org/ARM_Paging, August 2019.
15. Identifying outliers: Iqr method. https://online.stat.psu.edu/stat200/lesson/3/3.2, 2022.
16. ARM. *ARM Architecture Reference Manual*, 2005.
17. Jonathan Valvano and Ramesh Yerraballi. *Embedded Systems - Shape The World*. Jonathan Valvano, Texas, 5 edition, 2014.
18. Platoformio. https://docs.platformio.org/en/latest/, 2014.
19. Irad Ben-Gal. Outlier detection. *Data mining and knowledge discovery handbook*, pages 131–146, 2005.
20. Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. Deep learning for anomaly detection: A review. *ACM computing surveys (CSUR)*, 54(2):1–38, 2021.