Moye: A Wallbreaker for Monolithic Firmware

Jintao Huang[1,2] Kai Yang[3*] Gaosheng Wang[1,2] Zhiqiang Shi[1,2*] Zhiwen Pan[1,2] Shichao Lv[1,2] Limin Sun[1,2]

[1]*Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS*
[2]*School of Cyber Security, University of Chinese Academy of Sciences*
Beijing, China
[3]*School of Computer, Electronics and Information, Guangxi University*
Nanning, China
{huangjintao, wanggaosheng, shizhiqiang, panzhiwen, lvshichao, sunlimin}@iie.ac.cn, yangkai@gxu.edu.cn

***Abstract*—As embedded devices become increasingly popular, monolithic firmware, known for its execution efficiency and simplicity, is widely used in resource-constrained devices. Different from ordinary firmware, the monolithic firmware image is packed without the file that indicates its format, which challenges the reverse engineering of monolithic firmware. Function identification is the prerequisite of monolithic firmware's analysis. Prior works on function identification are less effectiveness when applied to monolithic firmware due to their heavy reliance on file formats. In this paper, we propose Moye, a novel method to identify functions in monolithic firmware. We leverage the important insight that the use of registers must conform to some constraints. In particular, our approach segments the firmware, locate code sections and output the instructions. We use a masked language model to learn hiding relationships among the instructions to identify the function boundaries. We evaluate Moye using 1,318 monolithic firmware images, including 48 samples collected from widely used devices. The evaluation demonstrates that our approach significantly outperforms current works, achieving a precision greater than 98% and a recall rate greater than 97% across most datasets, showing robustness to complicated compilation options.**

***Index Terms*—Function Identification, Monolithic Firmware, Unformatted Binary**

I. Introduction

Internet of Things (IoT) devices have witnessed significant growth in popularity over the years, with projections indicating that there will be more than 29 billion IoT connections by 2027 [1]. Despite the convenience these devices bring, their widespread adoption has also brought significant security risks. On average, 54% of organizations face attempted cyber attacks targeting IoT devices weekly [2].

To safeguard these devices, extensive research [3], [4] has been conducted in various domains, such as patch presence testing and bug searching. Most of these research is performed at the function level, highlighting the importance of function identification.

Function identification aims to identify the location of the target function and the functions that call it, enabling analysts to determine which instructions belong to these functions. It has garnered substantial research interest [5], [6]. Wartell et al. [7], Miller et al. [8] and Bao et al. [9] have attempted to recognize functions using probabilistic analysis. In contrast, Bauman et al. [10] and Flores et al. [11] have opted to identify functions using Superset Disassembly and heuristic rules. With the advancement of deep learning and natural language processing (NLP) techniques, researchers have endeavored to train models to recognize functions in binary programs [5], [6], [12], These approaches have improved both the precision and efficiency of binary analysis, particularly for binaries with well-documented file formats. Although these methods are primarily designed for normal executable binaries (e.g., ELF files), they can also be applied to the analysis of ordinary firmware images by extracting their executable binaries.

Nevertheless, function identification remains a challenging task since monolithic firmware doesn't allow for the extraction of its executable binaries. Monolithic firmware, known for its simplicity and efficient execution, is widely used in lower-power, purpose-built and resource-constrained embedded devices [13], [14]. A monolithic firmware image is designed to be self-contained, integrating the entire firmware codebase, data, and even files into a single, contiguous file. This type of firmware does not adhere to any publicly known file format, and the file formats of all the executable binaries are erased before they are packaged into the firmware.

However, monolithic firmware is not without its drawbacks. It runs entirely within a single address space, with the CPU executing in supervisor mode [15]–[17]. This setting implies that exploiting a vulnerability in a low-criticality component can be sufficient to achieve privilege escalation [18], [19].

On the one hand, security analysis is essential for monolithic firmware to protect it from exploitation. On the other hand, the absence of a file format in monolithic firmware builds a high wall that hinders current approaches from identifying functions within it, thereby preventing further downstream analysis. This dilemma has led to the neglect of monolithic firmware by both academia and industry [20], creating a significant gap between the widely used, yet fragile, monolithic firmware and the advanced, systematic downstream analysis techniques.

To bridge the great gap, we propose `Moye`[1] in this paper. `Moye` utilizes a Computer Vision (CV) model to analyze

*Corresponding Authors

[1]Moye is the name of an ancient sword in Chinese legend.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00053

the distribution characteristic of jump instructions, thereby locating code sections within the target firmware. Following this, Superset Disassembly is performed, and execution paths are constructed by Moye. Moye collects the manipulations of registers within these execution paths, and ultimately determines the boundaries of functions using knowledge acquired through a Masked Language Model (MLM).

To evaluate Moye's performance, we compare it with popular function identification methods. The results show that our method significantly outperforms other works in terms of both effectiveness and robustness, achieving a precision greater than 98% and a recall rate greater than 97% across most datasets. Moreover, Moye's excellent robustness to various compilation settings and its independence from file formats make it practical for analyzing monolithic firmware.

The contributions of this paper are summarized as follows:

- We propose a practical and reliable method capable of precisely identifying functions in monolithic firmware with a high recall rate.
- We introduce a novel approach for differentiating code and data. According to our evaluation, this separation scheme benefits both the precision and recall of the function identification process.
- We implement an end-to-end system named Moye, designed to be scalable, and easily extendable to other Instruction Set Architectures (ISAs).
- We conduct comprehensive evaluation on Moye using 1,318 monolithic firmware images, including 48 firmware samples collected from real-world devices. The evaluations comfirm the robustness of Moye and its superiority over other state-of-the-art tools.

II. Motivation

A. Monolithic Firmware

Monolithic firmware refers to a unified and indivisible firmware entity that encapsulates all functionalities into a single image. It is prevalent, particularly in resource-constrained embedded systems, where efficiency and simplicity in design are paramount. The popularity can be attributed to monolithic firmware's foremost advantage: its streamlined execution flow. This characteristic eliminates the overhead associated with inter-module communication and facilitates efficient resource utilization. and is particularly advantageous in scenarios where real-time responsiveness is crucial.

The construction of monolithic firmware involves a systematic process of software development, compilation, integration and firmware image generation. During integration, components necessary for firmware execution, such as device drivers, system routines, and application logic, are combined into a cohesive file. Subsequently, the process of generating the final firmware image creates an image that can be loaded onto the target hardware. This image typically contains bootloader code, configuration data, and the integrated executable.

During the construction process, various intermediate files (e.g., map files and ELFs with symbol tables) are generated, which provide extensive information on how the executable is linked and the locations of public symbols [21]. However, they are considered as internal property, and are neither made public nor included in the final firmware image.

B. Challenges

We address several challenges in function identification for monolithic firmware as follows.

1) Absence of File Format: Typically, devices using monolithic firmware have limited resources. As a result, all content unnecessary for firmware execution, including firmware's own file format, is removed to minimize firmware size. This leads to the absence of metadata that indicates the location and boundaries of code segments.

2) Difficulty in Dynamic Analysis: For a stripped ELF executable, one practical method of analysis is to dynamically execute it using the entry point specified in its ELF header, and then recover function boundaries by analyzing the CPU execution trace. However, this method cannot be applied to monolithic firmware. First, there is no information indicating the firmware's entry point. Second, monolithic firmware often runs on custom hardware platforms, making it challenging to obtain tools that support these platforms. Third, firmware used in embedded systems frequently interacts directly with hardware peripherals. Emulating these interactions poses additional challenges, as it requires specialized tools and a deep understanding of hardware-specific behavior.

3) More Aggressive Compilation Optimization: Monolithic firmware prioritizes meeting constraints on resources over ease of analysis, leading to the use of more aggressive compilation optimization. Such compiling setting can alter the code structure, including the removal of function prologues [5].

C. Limitation of Current Approaches

Numerous approaches have been proposed for function identification in stripped ELF executables, but they face significant challenges when processing monolithic firmware.

FunProbe [22] requires ELF formats to provide enough hints while DeepDi [6] relies on file formats to locate code segments before identifying functions. This dependency hinders it from analyzing monolithic firmware, which lacks such file formats. Yin et al. [23] and D-arm [24] focus on binaries of ARM ISA, while XDA [5] targets x86 and x64 ISAs. The heavy reliance of ISA-specific characteristics limits their applicability in analyzing monolithic firmware, which may encompass diverse ISAs [25]–[27]. Tools like IDA Pro [28], Ghidra [29], Jakstab [30] and radare2 [31] identify functions using function prologues. However, maintaining an up-to-date prologue database is tedious and costly. Currently, IDA Pro's prologue database exceeds 41MB in size, while Ghidra's exceeds 179MB [5]. Worse yet, these function prologues can be eliminated by certain compilation optimizations.

III. Overview

A. Assumption

To identify functions within monolithic firmware, several assumptions are made.

1) Inaccessibility of Firmware Source Code to Users: With source code, anyone can easily identify functions by rebuilding the firmware while retaining symbol tables. External testers lack such access to the source code and other intermediate files. They can only obtain firmware binaries by downloading them from manufacturers' websites or by reading from device flash with hardware connections.

2) Absence of Encryption or Compression in Firmware: Encryption serves to protect proprietary information, while compression reduces firmware size. However, both techniques can modify firmware bytes, thereby obscuring instruction characteristics and complicating firmware analysis. Given the scope of this paper, decryption and decompression of firmware images are not addressed, and we assume the firmware image remains unencrypted and uncompressed.

3) Lack of Code Obfuscation: Code obfuscation is employed to safeguard programs, however, it can introduce overhead that terribly impacts execution performance on resource-constrained devices. Therefore, code obfuscation is rarely used in monolithic firmware, and is not considered in this paper.

B. Insights

The insights behind our approach are illustrated as follows.

1) Jump Instructions: Jump instructions are widely used in monolithic firmware. According to our analysis conducted on 2,843,880 functions extracted from 10,826 programs, 62.599% of functions contain jump instructions, and 30.324% of functions contain more than 5 jump instructions. On average, each function contains 6.520 jump instructions. Additionally, jump instructions tend to be close to the entry points of their related functions. Our findings indicate that 78.66% of jump instruction addresses are within 1024 bytes of the entry points of their related functions.

2) Register Manipulation: Registers are important to program execution, with most instructions operating on them. The manipulations of registers are governed by both explicit and implicit rules. For instance, some registers will be modified at the end of function. According to our analysis performed on 7,379,646 execution paths in ARM ISA, 83.569% of them end with setting a value for register `PC`. Additionally, a general-purpose register is seldom read if it has not been initialized with a value. Consider the ARM instruction "*MOV R3, R12*", which fetches the data from register `R12` to set register `R3`. The contents of registers are unpredictable at the very beginning of program execution. If such an instruction is the first one to be executed, the dirty data it reads from `R12` can cause execution to panic. To avoid this, another instruction, which sets the content of `R12`, should be executed first.

C. Our Idea

In monolithic firmware, the absence of indicative file formats makes it essential to precisely locate code segments for accurate function identification. The state-of-the-art tool `ELISA` separates code from data using the features of the bytes located close to the boundary between code and data regions. The use of byte-level features makes it sensitive to the content of data regions. To improve robustness, we focus on the distribution features of jump instructions, which consistently form rectangular clusters in scatter plots (instruction address on the X-axis, jump target on the Y-axis) regardless of the data region content. We recognize these rectangular clusters by employing the Mask R-CNN model,

Once the code segments are located, Superset Disassembly [10] is performed on them, and program analysis is conducted to construct Control Flow Graphs (CFGs). A CFG contains multiple execution paths, each comprising numerous instructions. We extract register manipulations from the instructions in each execution path and use a MLM to learn the relationships between these operations. These relationships are some constraints on register operation that remain constant across various compilation settings. Although Superset Disassembly may generate false register manipulations, but these false register manipulations rarely meet the register operation constraints. Compared to the function prologues used by other tools, which are only a few bytes long and can be easily affected by compilation optimizations, register manipulation constraints should be satisfied in multiple instructions consisting of tens of bytes. As a result, byte sequences that meet these constraints are much less likely to appear in data regions than sequences resembling function prologues. Finally, we leverage the learned relationships to determine function boundaries.

D. Users and Usage Scenarios

`Moye`'s target users are external analysts outside the manufacturer seeking to perform downstream analysis on monolithic firmware.

IV. Moye

The overview of `Moye` is illustrated in Fig. 1. The lines and data in gray denote the training process, while those in orange represent the prediction procedure.

During training, steps ❶ to ❸ involve training the Mask R-CNN model used for code separation. CFGs for functions are then constructed based on the ground truth function boundaries (step ❹). Using these CFGs, register manipulations are extracted (step ❺) and fed into a MLM to learn the relationships between manipulations (step ❻). Finally, using a transfer learning approach, `Moye` transfers the learned relationships to facilitate function identification (step ❼).

In prediction process, `Moye` extracts all jump instructions from target firmware image, and passes them to Mask R-CNN to locate code regions (step ❶ to ❸). Subsequent Superset Disassembly retrieves all possible instructions (step ❹), enabling CFG construction (step ❺) and extraction of register manipulations (step ❻). Finally, the trained MLM uses these register manipulations to identify functions (step ❼).

A. Code Separation

To distinguish code from data in the target firmware, `Moye` first identifies all byte sequences that can be disassembled as jump instructions. Then it creates a scatter plot where jump instruction addresses are plotted on the X-axis and their targets

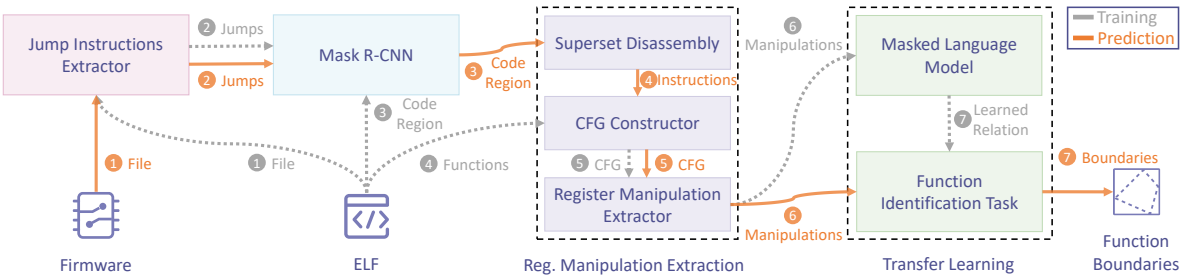


Fig. 1. The overview of Moye. Lines and data in gray indicate the training procedure, while those in orange represent the prediction process.

on the Y-axis. An example of such a plot is shown in Fig. 2(a), where orange points represent actual jump instructions and blue points denote the misidentified. Since jump instructions and their targets reside within code segments, all the orange points cluster within the actual code region delineated by a red rectangle. Worthy to note that the color difference and the red rectangle are to show the clustering of real jump instructions. In the actual prediction process, all points will be in the same color, with no rectangular box.

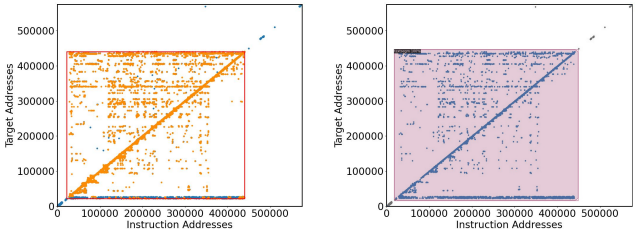
Moye uses the Mask R-CNN model to identify these regions. The architecture of Mask R-CNN, depicted in Fig. 3, comprises several components: a backbone network for feature extraction, a Region Proposal Network (RPN) for generating candidate object regions, RoI Align for spatial alignment of features, an object detection head for class probability and bounding box refinement, and a mask prediction branch for pixel-level segmentation. Through end-to-end training with multiple loss functions, including classification, bounding box regression, and mask segmentation losses, Mask R-CNN achieves superior performance in instance segmentation tasks.

The segmentation outcome produced by Mask R-CNN for Fig. 2(a) is shown in Fig. 2(b). Points within the red area are assumed to denote real jump instructions.

To determine the starts of code sections, Moye uses *addrs* and *tgts* to represent these instructions' addresses and target addresses, respectively. If $min(tgts) < min(addrs)$, then $min(tgts)$ is regarded as the start of the code section. If $min(tgts) \geq min(addrs)$, then we disassemble from $min(addrs) - 1024$, construct execution paths, and determine the starting point *ES* of the function with trained MLM model. The *ES* is regarded as the start of the code segment. We also disassemble from $max(addrs \bigcup tgts)$ and construct execution paths. The byte immediately following the last instruction in the execution path is regarded as the end of code segment.

B. Function Identification

Once the code segment is located, Moye employs Superset Disassembly, disassembling from every byte to capture all possible instructions. Then, it constructs execution paths with disassembled instruction, considering instruction semantics. Register manipulations are then extracted from the instructions within these execution paths. Finally, Moye utilizes a trained MLM to determine function boundaries with these register manipulations. The effectiveness of using MLM for function



(a) A scatter plot of jump instructions. (b) Segmentation for code regions.

Fig. 2. An example of code separation.

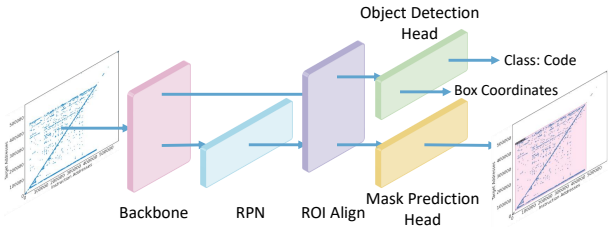
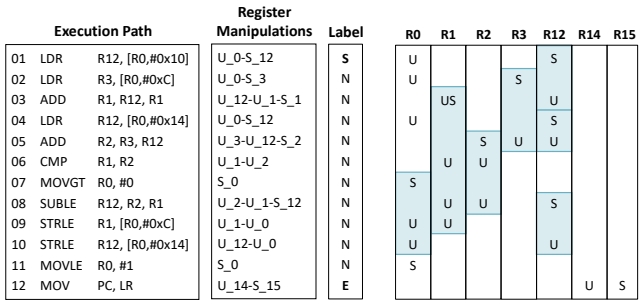


Fig. 3. The architecture of the Mask R-CNN model.

identification has been previously demonstrated by XDA, which fed raw bytes into the MLM for analysis.

Moye identifies functions by analyzing register manipulations, as illustrated in Fig. 4(a). Only tokens indicating register manipulations are fed into Moye's MLM, which then labels these tokens as function starts (S), function ends (E) or non-boundary instructions (N). To clarify how register manipulations aid in function identification, we visualize these manipulations in Fig. 4(b), where blue boxes represent manipulation cycles for registers. An instruction that breaks from the middle of boxes indicates that it uses registers before setting them.

The instruction at line 01 is identified as the function start, as it initiates the use of the register used for argument passing (R0) and initializes the value of R12, which is used later. Other instructions are not considered as the function start, because the register manipulations in their subsequent instructions will not conform to the following constraints: (1) general registers, excluding those used for argument passing (e.g., R0 and R1), are initialized before use and not consecutively reinitialized; (2) all registers are eventually used after initialization, except R0, which stores the function's return value. For predicting



(a) An execution path and register manipulation example

(b) The manipulation sequence for registers

Fig. 4. An example of function identification

function ends, another register manipulation constraint is considered: an execution path should end with instructions that influence the value of `PC`. This is because most functions are expected to return the execution flow after they finish execution. Functions that do not return execution flow typically call a system function to exit execution (e.g., "*BL exit*") or perform an infinite loop. Apparently, the only possible function end is the instruction at line `12`, which sets `PC` with the address of the next instruction of the call site stored in `LR`.

To enable `Moye` to identify functions using these implicit register manipulation rules, we prepare a MLM for it and divide the training of MLM into 2 stages: pre-training and fine-tuning. During pre-training, `Moye` first generates tokens representing register manipulations for the instructions within each execution path. These tokens are then sequenced to represent the execution path, as depicted in Fig. 4(a). The tokens use specific labels: "*U_x*" indicates "use register `Rx`", "*S_y*" signifies "set register `Ry`", "*STM_m*" denotes "push register `Rm`'s value onto the stack" and "*LDM_n*" indicates "pop data from the stack to set register `Rn`". Additionally, function calls and jump instructions are denoted by "<*call*>" and "<*jump*>", respectively.

To help MLM learn the relationships between tokens, we randomly mask 8% of the tokens in a sequence with a special token "<*mask*>". Each token, whether masked or not, is then embedded as a one-hot vector of fixed size. We also encode positions of tokens in the token sequence into another vector of the same dimension. These token vectors and positional vectors are combined to form the actual embedding of the input execution path, and are updated by a multi-head self-attention mechanism. Finally, MLM iteratively predicts the masked tokens and updates the model's parameters until it achieves low perplexity in predicting masked tokens.

During fine-tuning, the model is trained with a labeled dataset. Each sample in this dataset is an execution path that starts from the entrance of a function and ends at the exit of the function. Instructions in sample execution paths are also abstracted as tokens representing register manipulations, and each of these tokens is labeled with `S`, `N`, and `E`. `Moye`'s MLM learns to classify each token as one of three classes using the knowledge obtained during pre-training.

V. Evaluation

We implement `Moye` using fairseq [32], a toolkit for sequence modeling. Both the training and inference of `Moye` are performed on a Linux server running Ubuntu 16.04, equipped with an Intel Xeon CPU E5-2620 v4 at 2.10GHz and 8 Nvidia GeForce GTX 1080 Ti GPUs. Through evaluation, we aim to answer the following questions:

- RQ1: How effective is `Moye` in identifying functions?
- RQ2: How robust is `Moye` to various compiling options?
- RQ3: How effective is the code separation method? How does it contribute to function identification in monolithic firmware performed by `Moye`?
- RQ4: How effective are the register manipulations?
- RQ5: How does the effectiveness of MLM compared to other models?
- RQ6: How effective is `Moye` in improving the performance of downstream applications?
- RQ7: What has MLM learned? How does it determine function boundaries?

A. Evaluation Setup

1) Baselines: We compare `Moye` against several established tools: `IDA Pro 7.6 SP1`, `Ghidra 10.1.5`, `D-arm`, `radare2 5.8.7`, `XDA` and `DeepDi`. `IDA Pro` is a renowned commercial disassembly tool developed by Hex-Rays. Both `radare2` and `Ghidra` are open-source reverse engineering tools, and `Ghidra` is developed by the NSA. `D-arm`, `XDA` and `DeepDi` are state-of-the-art tools proposed in academic research. For `D-arm`, `XDA` and `DeepDi`, which require ELF formats to locate code segments, all bytes of target firmware are passed to them as potential code due to the absence of file formats in monolithic firmware. Additionally, we integrate them with `ELISA` [33], a state-of-the-art code separation tool that uses a Conditional Random Field (CRF) [34]. The integrated tools are denoted as `D-arm/ELISA`, `XDA/ELISA` and `DeepDi/ELISA` respectively.

As `XDA` targets on x64 ISA only, we follow the instructions provided by its authors to train it using the same corpus and dataset splits as `Moye` to enable it to identify functions on ARM ISA. Since both `IDA Pro` and `Ghidra` integrate prologue-based schemes as part of their identification strategies, we do not conduct separate evaluations of tools that rely solely on function prologues.

`FunProbe` [22] is excluded because we are unable to provide the necessary ELF format hints that it requires for function identification.

2) Dataset and Ground Truth: The detail of datasets are presented in Table I. *UCSB* is collected by Gritti et al. [27] from various research datasets [14], [35]–[41]. To construct the ground truth, we filter out the firmware that is not accompanied by an associated ELF file. Datasets *Zephyr*, *VxWorks*, *RIOT*, *LiteOS*, *Contiki-NG* and *Embox* are built by us on the sample codes of Real-Time Operating Systems (RTOSs) Zephyr [42], VxWorks [43], RIOT [44], LiteOS [45], Contiki-NG [46] and Embox [47], respectively. All these firmware images

TABLE I
EVALUATION DATASET

Dataset	Source	ISA	# File	Size (MB)	# Function
DeviceFirm	Collected	ARM	48	201.29	454,666
UCSB	Collected	ARM	36	3.67	15,807
VxWorks	Built	ARM	26	20.13	91,125
RIOT	Built	ARM	23	1.46	13,185
LiteOS	Built	ARM	94	9.17	47,391
Embox	Built	ARM	55	48.38	169,846
Contiki-NG	Built	ARM	41	2.01	20,167
Zephyr	Built	ARM	995	34.69	263,621
Total	-	-	**1,318**	**320.80**	**1,075,808**

TABLE II
THE DETAIL OF THE DATASET DEVICEFIRM

Vendor	# Model	# File	Size (MB)	# Function
TP-Link	14	34	122.44	302,629
FAST	3	3	12.54	26,084
MERCURY	3	4	17.03	39,892
D-Link	2	2	3.25	6,566
Tenda	3	3	13.31	24,249
RICOH	2	2	32.71	55,246

are constructed using the default compiling settings set by developers. VxWorks is a priority-based commercial RTOS widely used in defense, automotive, and other fields [43]. Zephyr, RIOT, LiteOS and Embox are open-source RTOSs, deployed in various markets, such as Sirius satellite radio receivers and Sony Playstation 3 Wi-Fi modules [42], [44]–[47]. We retain the symbol tables of ELFs associated with the firmware for ground truth construction. Dataset *DeviceFirm* is collected from embedded device manufacturers. We download thousands of firmware and retain those where the related symbol table was accidentally included. Finally, 48 of them are retained. The details are shown in Table II.

The ground truth for all datasets is established before conducting the evaluation. For each firmware in *UCSB* and the datasets we build, we parse the symbol table of the associated ELF file using `IDA Pro`. For real-word firmware, we extract the symbol tables that are unexpectedly retained. Using these symbol tables, we construct function boundary ground truth.

3) Evaluation Metrics: We measure tools' performance using precision ($\frac{TP}{TP+FP}$, where TP means "true positive", FP indicates "false positive"), recall ($\frac{TP}{TP+FN}$, where FN denotes "false negative") and F1 ($\frac{2*precision*recall}{precision+recall}$). Higher precision reflects the tool's credibility, while higher recall means less human efforts required for function identification. Since both precision and recall are essential in realm of function identification, F1 is chosen rather than F0.5.

4) Pre-training & Fine-tuning: We pre-train and fine-tune our model using 1,612 binaries collected from 2 sources. Dataset *BAP* is collected from previous research [9], containing 410 executable programs. Dataset *Ubuntu* is collected from Ubuntu repositories using the script provided by Han et al. [48]. The details of the binary corpus division are shown in Table III. In total, 450,740 functions and 54,757,662 execution paths are involved in the process.

TABLE III
BINARY CORPUS FOR PRETRAIN/FINE-TUNE

Phase	Subset	Dataset	# File	# Func.	# Exec. Path
Pretrain	Training	BAP	202	45,225	4,121,609
		Ubuntu	600	154,813	18,186,713
	Validation	BAP	100	19,880	1,636,009
		Ubuntu	300	115,155	17,299,689
	Total	-	1202	335,073	41,244,020
Fine-tune	Training	BAP	60	10,142	796,485
		Ubuntu	160	52,629	8,639,584
	Validation	BAP	50	7,755	825,543
		Ubuntu	140	45,141	3,252,030
	Total	-	410	115,667	13,513,642
Total	-	-	**1612**	**450,740**	**54,757,662**

We pre-train `Moye` with unlabeled register manipulation sequence for 12 epochs. After that, the model's perplexity (PPL) decreases from 265.16 to 1.71. In fine-tuning, `Moye` is trained for 4 epochs using labeled register manipulation sequences, where labels indicate the type of register manipulation tokens (e.g., `S`, `N` or `E`). Throughout both pre-training and fine-tuning, the maximum sequence length is set to 1,024.

B. Performance of Moye

1) Precision and Recall (RQ1): We compare `Moye`'s performance with baselines, and summarize the results in Table IV. As illustrated in the table, even commercial tools like `IDA Pro` fail to identify functions in datasets *UCSB*, *LiteOS*, *Contiki-NG*, *Embox*, *RIOT*, and *Zephyr*. Although `IDA Pro` achieves high precision in datasets *DeviceFirm* and *VxWorks*, its recall remains lower than 0.400, leading to a poor F1 score.

`Ghidra` has precision comparable to `IDA Pro` in datasets *DeviceFirm* and *UCSB*. Unlike `IDA Pro`, `Ghidra` is able to identify functions in all datasets. However, `Ghidra` can only achieve recall lower than 0.870, indicating that significant human effort is required to recognize the missed functions when analyzing monolithic firmware. Compared to `Ghidra`, `radare2` attains higher recall in some datasets (3/8), but its precision is much lower, implying the low credibility of its output. The performance difference between `Ghidra` and `radare2` indicates that `Ghidra` is more conservative.

Similar to `radare2`, `D-arm` applies radical strategies in recognizing functions. However, compared to `radare2`, `D-arm`'s precision is much lower. Such low precision hinders its use in daily analysis since a large proportion of its results are false positives.

`DeepDi` fails to produce results in most datasets (6/8). Even in dataset where it is able to output identification result, such as *DeviceFirm*, it achieves extremely low precision and recall, implying its poor practicality in identifying functions in monolithic firmware. Although `XDA` utilizes MLM as well, it successfully identifies functions in only 3 datasets, and its performance is relatively poor.

`D-arm/ELISA`, `XDA/ELISA` and `DeepDi/ELISA` fail to obtain better performance even though they are integrated with the code separation tool `ELISA`. For `D-arm/ELISA`,

compared to D-arm, its recall drops sharply after being integrated with ELISA, leading to a drop in its F1 score. For XDA/ELISA, ELISA helps it to output result in the *Contiki-NG* dataset, but it does not significantly improve performance.

Compared to its adversaries, Moye achieves much better performance from the perspective of precision, recall and F1 score. Moye achieves unparalleled recall in all datasets, and in most datasets (7/8), its recall is higher than 0.950. Meanwhile, Moye attains precision comparable to that of both IDA Pro and Ghidra, which are the most widely used commercial and open-source tools, respectively.

We also summarize the relationship between tools' results in Table V. Generally speaking, Moye is able to identify more exclusive functions—functions that can only be located via Moye. As shown in the table, in most datasets (7/8), more than 95% of the functions identified by other tools are also identified by Moye, indicating the great recall of Moye. However, even Ghidra, which has the largest intersection with the results of Moye, can only recognize less than 80% of Moye's result in most datasets, not to mention other tools. The comparison shows Moye's superiority in identifying functions.

2) Robustness (RQ2): Various compilation options are used in monolithic firmware. Since it is challenging to determine the compilation configuration used when constructing the target firmware, it will be practical if the performance of a tool persists in various compilation settings. To assess the robustness of Moye, we test it alongside its competitors using 4,291 firmware built with different compiling settings. The detail of the dataset is shown in Table VI.

Tools' F1 scores are shown in Fig. 5. The figure illustrates that the variety of compilation settings indeed creates challenges in function identification. As the optimization levels used in firmware construction change, most tools' F1 scores fluctuate. For instance, Ghidra's F1 score drops from 0.977 to 0.704 in the *Zephyr* dataset when the optimization level is switched from O0 to Os, a decrease of 0.273. As a comparison, the most significant drop of Moye's performance occurs in the *LiteOS* dataset when the optimization level changes from O0 to O1, with the F1 score decreasing by 0.033 from 0.972 to 0.939. Despite slight fluctuations, Moye's performance remains superior overall, indicating that Moye is more robust than other tools and is more practical for analyzing monolithic firmware built with various compilation settings.

3) Efficiency: Moye is capable of identifying functions for the majority of firmware images (1067/1318) within 3 minutes, while Ghidra identifies functions for 1,165 firmware images within the same time frame. The highest efficiency is achieved by radare2, which identifies functions for all firmware within 3 minutes. While both radare2 and Ghidra have slight advantages in efficiency over Moye, their low recall rates and precision necessitate more human effort for identifying and verifying functions, often taking hours to complete.

C. Ablation Evaluation

To assess the individual contributions of each component of Moye to the precision, recall and robustness of the overall

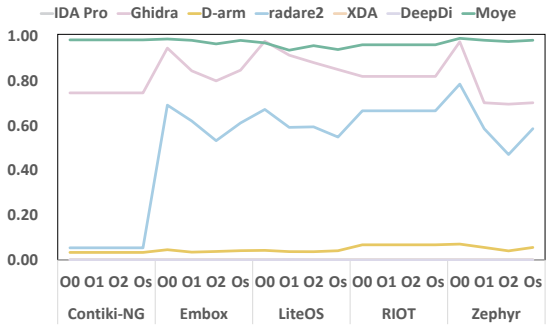


Fig. 5. Tools' F1 scores over the datasets compiled with different settings.

function identification process, we compare Moye's code and data separation technique with ELISA (summarized in Table VIII) and conduct a series of ablation evaluations on the *Zephyr* dataset as summarized in Table VII.

1) Contribution of Code Separation (RQ3): As Table VIII shows, although ELISA performs impressively on the dataset *VxWorks*, its performance fluctuates significantly across other datasets. In contrast, Moye's code and data separation technique is more stable and maintains its superiority in most datasets (6/7). ELISA's performance variability stems from its reliance on the features of bytes located near the boundary between code and data. Such reliance makes ELISA sensitive to the contents of non-instruction parts of the firmware, and leads to inconsistent results in certain cases.

In the comparison between Moye's code separation strategy and ELISA regarding the enhancements they provide to function identification process, the performance of the variants of Moye is detailed in Table VII. In the table, Moye/None means the original code separation method is removed from Moye, while Moye/ELISA stands for that Moye identifies functions based on the code sections located by ELISA.

In the evaluation, it is evident that Moye outperforms both Moye/ELISA and Moye/None in terms of precision and recall. This performance advantage highlights the significant role of code separation in function identification for monolithic firmware. However, the comparison between Moye/ELISA and Moye/None demonstrates that code separation methods also have drawbacks. This is because a code separation approach prevents the tool from misinterpreting data as code by confining the analysis to specific content regions. If certain code regions are not accurately identified, the function identification process may overlook functions within those regions.

In summary, code separation significantly affects both the precision and recall of function identification.

2) Effectiveness of Register Manipulation (RQ4): Register manipulation serves as the core feature that Moye relies on for function identification. To evaluate its contribution to the overall function identification process, we replaced it with the raw bytes of the target firmware and conducted a comparison The modified version of Moye is denoted as Moye/Byte. We evaluate both Moye and Moye/Byte on the *Zephyr* dataset and summarize their performance in Table VII.

TABLE IV
PRECISION (P), RECALL (R) AND F1 OF FUNCTION BOUNDARY IDENTIFICATION SCHEMES

Dataset	Metrics	IDA Pro	Ghidra	D-arm	radare2	XDA	DeepDi	D-arm/ELISA	XDA/ELISA	DeepDi/ELISA	Moye
DeviceFirm	P	**1.000**	0.998	0.020	0.356	0.003	0.001	0.020	0.003	0.001	0.966
	R	0.145	0.655	0.720	0.580	0.000	0.000	0.718	0.000	0.000	**0.978**
	F1	0.253	0.791	0.038	0.441	0.001	0.000	0.040	0.001	0.000	**0.972**
UCSB	P	-	**0.993**	0.022	0.543	-	-	0.022	-	-	0.979
	R	-	0.720	0.480	0.811	-	-	0.042	-	-	**0.957**
	F1	-	0.834	0.043	0.651	-	-	0.028	-	-	**0.968**
VxWorks	P	0.979	**0.996**	0.022	0.130	0.029	-	0.022	0.003	-	0.981
	R	0.400	0.713	0.500	0.058	0.002	-	0.276	0.000	-	**0.858**
	F1	0.568	0.831	0.042	0.080	0.004	-	0.040	0.000	-	**0.916**
RIOT	P	-	**0.983**	0.036	0.550	-	-	0.033	-	-	0.945
	R	-	0.706	0.569	0.849	-	-	0.052	-	-	**0.982**
	F1	-	0.822	0.068	0.667	-	-	0.041	-	-	**0.963**
LiteOS	P	-	**0.995**	0.021	0.482	-	-	0.026	-	-	0.981
	R	-	0.848	0.549	0.538	-	-	0.199	-	-	**0.954**
	F1	-	0.916	0.040	0.509	-	-	0.046	-	-	**0.968**
Embox	P	-	**1.000**	0.023	0.561	-	0.012	0.026	-	0.012	0.984
	R	-	0.870	0.593	0.842	-	0.000	0.193	-	0.000	**0.988**
	F1	-	0.930	0.045	0.673	-	0.000	0.046	-	0.000	**0.986**
Contiki-NG	P	-	**0.999**	0.018	0.039	-	-	0.017	0.008	-	0.984
	R	-	0.594	0.275	0.062	-	-	0.145	0.000	-	**0.990**
	F1	-	0.745	0.033	0.048	-	-	0.031	0.000	-	**0.987**
Zephyr	P	-	0.958	0.030	0.463	0.007	-	0.022	0.002	-	**0.986**
	R	-	0.556	0.364	0.774	0.000	-	0.006	0.000	-	**0.980**
	F1	-	0.704	0.055	0.579	0.000	-	0.009	0.000	-	**0.983**

TABLE V
RELATIONSHIP BETWEEN THE FUNCTIONS IDENTIFIED BY THE TOOL T AND MOYE

Tool T	Relation	DeviceFirm	UCSB	VxWorks	RIOT	LiteOS	Embox	Contiki-NG	Zephyr
IDA Pro	$\|T \cap Moye\|/\|T\|$	-	-	-	-	-	-	-	-
	$\|T \cap Moye\|/\|Moye\|$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Ghidra	$\|T \cap Moye\|/\|T\|$	0.997	0.988	0.993	0.996	0.978	0.994	1.000	0.995
	$\|T \cap Moye\|/\|Moye\|$	0.668	0.742	0.808	0.716	0.869	0.875	0.600	0.569
D-arm	$\|T \cap Moye\|/\|T\|$	0.981	0.985	0.884	0.990	0.973	0.992	0.982	0.982
	$\|T \cap Moye\|/\|Moye\|$	0.722	0.494	0.515	0.574	0.559	0.596	0.273	0.367
radare2	$\|T \cap Moye\|/\|T\|$	0.993	0.978	0.971	0.993	0.970	0.995	0.965	0.992
	$\|T \cap Moye\|/\|Moye\|$	0.589	0.829	0.069	0.858	0.547	0.848	0.060	0.791
XDA	$\|T \cap Moye\|/\|T\|$	1.000	-	0.968	-	-	-	-	1.000
	$\|T \cap Moye\|/\|Moye\|$	0.000	0.000	0.003	0.000	0.000	0.000	0.000	0.000
DeepDi	$\|T \cap Moye\|/\|T\|$	1.000	-	-	-	-	1.000	-	-
	$\|T \cap Moye\|/\|Moye\|$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
D-arm/ELISA	$\|T \cap Moye\|/\|T\|$	0.982	0.994	0.885	0.990	0.988	0.989	0.984	0.951
	$\|T \cap Moye\|/\|Moye\|$	0.721	0.043	0.280	0.052	0.206	0.040	0.144	0.006
XDA/ELISA	$\|T \cap Moye\|/\|T\|$	1.000	-	0.905	-	-	-	1.000	1.000
	$\|T \cap Moye\|/\|Moye\|$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
DeepDi/ELISA	$\|T \cap Moye\|/\|T\|$	1.000	-	-	-	-	1.000	-	-
	$\|T \cap Moye\|/\|Moye\|$	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

TABLE VI
FIRMWARE COUNT OF DATASETS COMPILED WITH VARIOUS OPTIMIZATION LEVELS (OPT. LEVEL).

Opt. Level	RIOT	LiteOS	Embox	Contiki-NG	Zephyr
O0	15	96	30	42	886
O1	15	96	26	42	893
O2	15	99	27	42	894
Os	15	96	27	42	893

As shown in Table VII, the precision and the recall of `Moye` are both close to 1.000, outperforming `Moye/Byte` which only achieves values slightly above 0.000. Our analysis indicates that the significant performance gap is attributed to the presence of data regions between functions. Although these regions may only consist of a few bytes, they are more likely to contain a single byte resembling normal function-start bytes rather than a byte sequence adhering to the constraints of register manipulation. In terms of robustness, the performance of `Moye/Byte` is too inadequate to demonstrate any fluctuation.

In conclusion, the utilization of register manipulation tokens enables `Moye` to achieve superior performance and practicality in function identification.

3) Advantages of MLM (RQ5): We leverage MLM to uncover the implicit relationships between register manipulations. To highlight the advantages of MLM over other NLP models, we replace the MLM component in `Moye` with a RNN

TABLE VII
PRECISION (P), RECALL (R) AND F1 OF TOOLS OVER THE DATASETS COMPILED WITH VARIOUS COMPILATION SETTINGS

Optimization Level	Moye			Moye/None			Moye/ELISA			Moye/Byte			Moye/RED		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
default	0.986	0.980	0.983	0.744	0.424	0.540	0.400	0.200	0.267	0.001	0.000	0.000	0.108	0.000	0.000
O0	0.989	0.996	0.992	0.501	0.109	0.180	0.343	0.054	0.093	0.010	0.000	0.000	0.047	0.000	0.000
O1	0.985	0.982	0.984	0.757	0.380	0.506	0.343	0.187	0.242	0.001	0.000	0.000	0.052	0.000	0.000
O2	0.984	0.972	0.978	0.675	0.374	0.481	0.301	0.208	0.246	0.000	0.000	0.000	0.069	0.000	0.000
Os	0.985	0.982	0.984	0.756	0.387	0.512	0.343	0.188	0.243	0.001	0.000	0.000	0.052	0.000	0.000

TABLE VIII
MOYE'S AND ELISA'S PRECISION (P) AND RECALL (R) ON CODE AND DATA SEPARATION

Tool	Metrics	VxWorks	UCSB	Contiki-NG	Embox	LiteOS	RIOT	Zephyr
Moye	P	0.997	**0.979**	**0.972**	**0.981**	**0.997**	**0.999**	**0.991**
	R	0.991	**0.994**	**0.970**	**1.000**	**0.981**	**0.993**	**0.989**
ELISA	P	**1.000**	0.995	0.846	0.527	0.995	0.964	0.683
	R	**1.000**	0.745	0.665	0.367	0.932	0.443	0.268

Encoder-Decoder model, commonly used in tasks like machine translation. The alternative tool is named `Moye/RED` and is trained on the same dataset for the same number of epochs as `Moye`. The comparison between `Moye/RED` and `Moye` is detailed in Table VII.

As depicted in Table VII, while both `Moye/RED` and `Moye` identify functions base on register manipulation tokens, `Moye/RED` does not achieve the same level of performance as `Moye`. The best performance of `Moye/RED` is observed when tested on firmware compiled with default compilation settings, where the precision is only slightly above 0.100 and the recall is nearly 0.000. Such poor performance makes `Moye/RED` impractical for real-world analysis. Based on our training setup and the performance disparities highlighted in the table, we infer that under identical training conditions, MLM excels in learning to identify functions with register manipulation relationships more efficiently.

D. Downstream Application (RQ6)

In assessing the impact of `Moye` on downstream applications, we perform Function Similarity Detection (FSD) on several monolithic firmware images using `Moye`'s results as input. For comparison, we also conduct FSD using the identified functions from other tools. The detection results were categorized into three types:

- **Matched**: The function identified by the tool correctly matched in FSD.
- **Unmatched**: The function identified by the tool is incorrectly matched in FSD.
- **Unrecognized**: The function is not identified by the tool.

The results of FSD are visualized in Fig. 6. It is evident that `Moye` produces the highest number of *Matched* results and the lowest number of *Unrecognized* results. While tools like `Ghidra` and `radare2` exhibit high precision and recall, respectively, in function identification, neither of them compares to the contribution `Moye` makes to downstream applications.

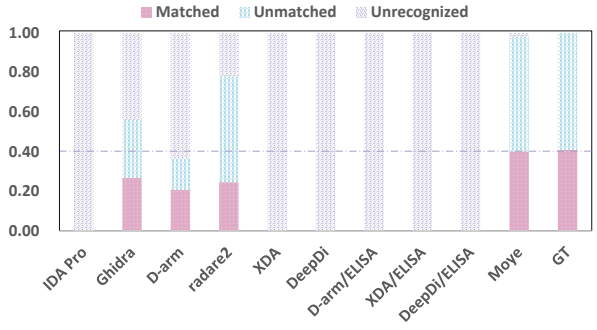


Fig. 6. The ratios of different FSD result types of different tools

To address the significant portion of *Unmatched* results in the figure, we also perform FSD using the ground truth of function addresses, represented by the bar `GT` in Fig. 6. The comparison reveals that even with the ground truth of function address, `jTrans` generates numerous *Unmatched* results. Conversely, `Moye` demonstrates performance very close performance to `GT`, highlighting its superiority in enhancing the performance of downstream applications.

E. Masked Language Model (RQ7)

1) Masked Token Prediction: To test whether MLM has effectively learned the relationships between register manipulations, we conduct an experiment where tokens in register manipulation sequences are masked. The pre-trained MLM was then used to recover these tokens. Consider the register manipulation sequence extracted from an example program of in the *VxWorks* dataset, as shown in Fig. 7. In this sequence, the register manipulation token of the instruction "*CMP R12, #0*" at line `07` is masked out, and `Moye`'s MLM is tasked with predicting the masked token. Notably, register `R12` is set at line `06` and then set again at line `09` using the data read from related address of register `R0`. According to the insights outlined in III-B2, a register is typically used before being set

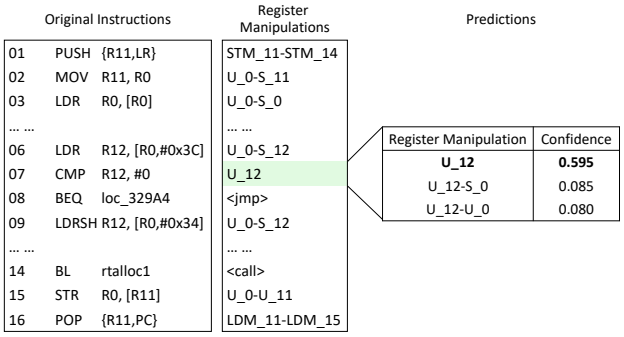


Fig. 7. Prediction for a masked register manipulation token.

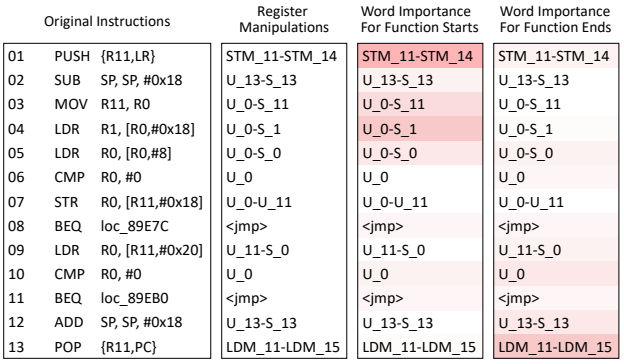


Fig. 8. The word importance for tokens when determining function starts and ends. Deeper color for greater importance.

again. Therefore, the register manipulation token at line `07` should involve the usage of register `R12`.

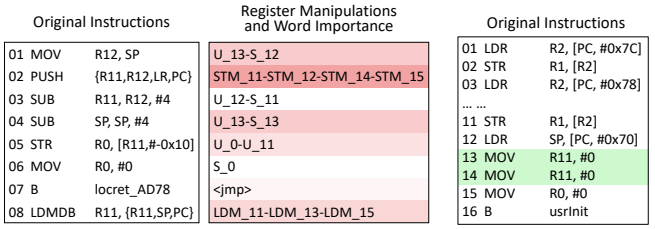
As depicted in the figure, all the top 3 predicted tokens involves "*U_12*", which indicates the use of register `R12`. Similarly, the other predicted tokens in the candidate also involve manipulation of the `R0` register. At line `07`, which lies between two use of register `R0`, both setting and using register `R0` are acceptable and conform to register manipulation constraints. `Moye` ensures that each of the top 3 candidate tokens satisfies the register manipulation constraint when its MLM's vocabulary size exceeds 3,000, and assigns the manipulation token "*U_12*" a high confidence level (> 0.59), This demonstrates that `Moye` effectively considers the relationships between register manipulations when predicting masked tokens.

2) Attention Visualization: To understand the decision-making process of MLM, we calculate the word importance for register manipulation tokens in identifying function starts and ends. The results are shown in Fig. 8.

During the identification of function starts, `Moye` places significant attention on tokens that save the context of caller functions (e.g., "*STM_11-STM_14*") and initiate registers (e.g., "*U_0-S_11*"). In contrast, other tokens receive little attention, as they tend to use registers rather than set them.

For predicting function ends, `Moye` focuses most on the token "*LDM_11-LDM_15*", which is linked to instructions that set the register `PC` with the value popped from the stack and are typically used as the last instruction of functions. Additionally, it pays moderate attention to tokens that utilizes registers (e.g., "*U_0*").

Generally speaking, `Moye`'s attention distribution when predicting function boundaries conforms to our observation: instructions that use but do not set registers–except the registers used for argument passing–are seldom seen at the beginning of functions, they are usually distributed in the middle of a function. Although both "*STM_11-STM_14*" and "*LDM_11-LDM_15*" are commonly associated with function prologues and epilogues, respectively, `Moye`'s relatively high attention on them should not be regarded as reliance on function prologues and epilogues. `Moye` does not use the actual instructions for prediction, it is limited to register manipulation tokens and identifies function boundaries by analyzing the relationships among these register manipulations.



(a) Incorrect Importance Distribution

(b) Uncommon Instruction Usage

Fig. 9. False negative cases of Moye.

F. Case Analysis

We conduct a case analysis with 100 false negative samples and 100 false positive samples randomly selected from various firmware. We try to find out why `Moye` fails to identify certain functions and why it misidentifies some instructions as function boundaries.

1) False Negatives: The causes of `Moye`'s false negative results are categorized into 2 types. The first type of false negative is due to incorrect word importance distributions. We present a case in Fig. 9(a), where the register manipulation token "*U_13-S_12*" corresponding to the actual first instruction "*MOV R12, SP*" receives high word importance. However, its importance is not as significant as the importance of "*STM_11-STM_12-STM_14-STM_15*", which is associated with pushing the register values onto the stack. This discrepancy in word importance ultimately leads to the misidentification of the function start, causing `Moye` to overlook the function. Our analysis indicates that incorrect word importance distributions like this are the primary reason for the majority (99/100) of `Moye`'s false negative results.

The remaining false negative, of the second type, is depicted in Fig. 9(b). In this scenario, the instructions at lines `13` and `14` are identical, both setting `R11` to a constant value of 0. This repetition of setting the same register with the same value consecutively contradicts the insights that `Moye` relies on, leading to the failure of recognizing the function. Fortunately, such instances are rare and only contribute to a small portion of `Moye`' false negative results.

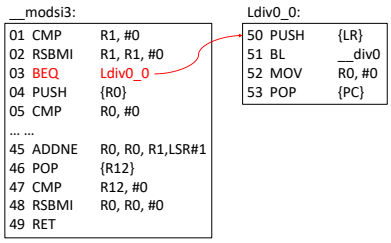


Fig. 10. A false positive case of Moye.

2) False Positives: At times, Moye might misidentify some instructions as function starts. Similar to the causes of false negatives, the most (97/100) of Moye's false positive results stem from improper attention distribution on register manipulation tokens. An example of this scenario has been shown in Fig. 9(a), where the instruction at line 02 is mistakenly identified as a function start.

The remaining 3 false positives are attributed to a different reason, as exemplified in Fig. 10. In this case, the function executes a conditional jump to the instructions starting from line 50 at line 03. Based on the learned register manipulation relationships, Moye interprets the instruction at line 50 as the start of a new function. However, despite being a stack-pushing instruction commonly found at the begining of many functions, the instruction at line 50 is actually a part of the function "*__modsi3*". This misidentification occurs because function "*__modsi3*" scatters the save operations of LR throughout its normal basic blocks, rather than consolidating them in the initial block like other functions.

VI. Related Work

A. Code Separation

Various approaches have been proposed to distinguish code from data within target binaries. Kruegel et al. [49] utilized control-flow analysis while Wartell et al. [50] combined a Predication by Partial Matching model (PPM) with heuristics to identify code segments in x86 files. P. De Nicolao et al. [33] tried to separate code from data with a CRF model, while Karampatziakis [51] presented a code discovery technique leveraging Support Vector Machines (SVMs). Later, Chen et al. [52] proposed to recognize instructions with static binary translation techniques.

B. Function Identification and Disassembly

Identifying functions with function prologues is commonly adopted by popular disassemblers like IDA Pro [28], Ghidra [29], radare2 [31], LEMNA [53], Dyninst [54] and Jakstab [30]. However, maintaining a prologue database is challenging, and a larger prologue database increase the risk of matching these patterns in data sections. To streamline the collection of function prologues, ByteWeight [9] was proposed, which automatically generates function prologues using a probability tree. Despite its claims, Andriesse et al. [55] found that ByteWeight's performance did not meet expectations. Other tools like Objdump [56], PSI [57], Uroboros [58], Nucleus [55], Yin et al. [23] and Qiao et al. [59] employ program analysis and various heuristics to identify functions, while FuncProbe [22] extracts 16 hints from ELF formats to locate function entry points. At the same time, the Shingled Graph Disassembly [7] and Probabilistic Disassembly [8] utilize probabilistic methods to identify genuine instructions. Bauman et al. [10] introduced the Superset Disassembly, while Flores et al. [11] applied a set of rules to differentiate real instructions from all possible instruction results. As deep learning techniques flourish, Shin et al. [12] proved neural network is effective in locating instruction start points, XDA [5] tried to tackle disassembly with general dependencies between bytes, and DeepDi [6] utilized Relational-GCN to find the most likely instruction sequence. TaiE [60] focuses on identifying functions in monolithic firmware. But it uses recursive analysis to identify functions, leading to non-linear increases in time complexity as firmware size grows. Approaches that based on dynamic analysis are also proposed [61]–[65], they are able to achieve better accuracy.

In conclusion, most static schemes require meta information, which is absent in monolithic firmware, to start their analysis. Dynamic analysis-based techniques are not suitable for monolithic firmware as illustrated in II-B2.

VII. Conclusion

Function identification in monolithic firmware is necessary but challenging. In this paper, we proposed Moye, which separates code from data and identifies function boundaries with the hiding relationship of registers manipulations. According to our evaluations, Moye outperforms its adversaries in terms of both effectiveness and robustness, making it more practical in analyzing monolithic firmware.

Acknowledgment

We appreciate all the anonymous reviewers for their invaluable comments and suggestions. The work is supported by Beijing Natural Science Foundation (Grant No. L234033) and Guangxi Science and Technology Base and Talent Special Project(Guike AD21076002). Kailong Wang deserves thanks for compiling the *VxWorks* dataset, and Siyi Zheng also deserves thanks for her work on beautifying the figures. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] I. Analytics, "Number of connected iot devices growing 16% to 16.7 billion globally," 2023, accessed: Dec 11, 2023. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[2] C. P. Research, "The tipping point: Exploring the surge in iot cyberattacks globally," 2023, accessed: Dec 11, 2023. [Online]. Available: https://blog.checkpoint.com/security/the-tipping-point-exploring-the-surge-in-iot-cyberattacks-plaguing-the-education-sector/

[3] X. Xu, Q. Zheng, Z. Yan, M. Fan, A. Jia, Z. Zhou, H. Wang, and T. Liu, "Patchdiscovery: Patch presence test for identifying binary vulnerabilities based on key basic blocks," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–14, 2023.

[4] J. Huang, G. Wang, Z. Shi, F. Lv, W. Zhang, and S. Lv, "SeHBPL: Behavioral semantics-based patch presence test for binaries," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 2023, pp. 92–111.

[5] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*, 2021.

[6] S. Yu, Y. Qu, X. Hu, and H. Yin, "Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2709–2725.

[7] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu, "Shingled graph disassembly: Finding the undecideable path," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2014, pp. 273–285.

[8] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1187–1198.

[9] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 845–860.

[10] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: Statically rewriting x86 binaries without heuristics." in *NDSS*, 2018.

[11] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1075–1092.

[12] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX security symposium (USENIX Security 15)*, 2015, pp. 611–626.

[13] M. A. Ben Khadra, D. Stoffel, and W. Kunz, "Speculative disassembly of binary code," in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, 2016, pp. 1–10.

[14] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise mmio modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1239–1256.

[15] Wikipedia, "Monolithic kernel — wikipedia, the free encyclopedia," 2023, accessed: Dec 12, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Monolithic_kernel

[16] S. Overflow. (2023) What is the difference between a kernel and a microkernel? Accessed: Dec 12, 2023. [Online]. Available: https://stackoverflow.com/questions/9187690/difference-between-monolithic-kernel-and-microkernel

[17] CodeDocs. (2023) Monolithic kernel. Accessed: Dec 12, 2023. [Online]. Available: https://codedocs.xyz/0intro/plan9/kernel.html

[18] S. Pinto and C. Garlati, "Multi zone security for arm cortex-m devices," in *Embedded World Conference*, vol. 2020, 2020.

[19] J. Jacobs, S. Romanosky, I. Adjerid, and W. Baker, "Improving vulnerability remediation through better exploit prediction," *Journal of Cybersecurity*, vol. 6, no. 1, p. tyaa015, 2020.

[20] E. D. Gustafson, *Discovery and Remediation of Vulnerabilities in Monolithic IoT Firmware*. University of California, Santa Barbara, 2020.

[21] Microsoft, "Generate mapfile," https://learn.microsoft.com/en-us/cpp/build/reference/map-generate-mapfile?view=msvc-170, 2023, accessed: Dec 22, 2023.

[22] S. Kim, H. Kim, and S. K. Cha, "Funprobe: Probing functions from binary code through probabilistic analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1419–1430.

[23] X. Yin, S. Liu, L. Liu, and D. Xiao, "Function recognition in stripped binary of embedded devices," *IEEE Access*, vol. 6, pp. 75 682–75 694, 2018.

[24] Y. Ye, Z. Zhang, Q. Shi, Y. Aafer, and X. Zhang, "D-arm: Disassembling arm binaries by lightweight superset instruction interpretation and graph modeling," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2391–2408.

[25] E. Gustafson, P. Grosen, N. Redini, S. Jha, R. Wang, A. Continella, K. Fu, S. Rampazzi, C. Kruegel, and G. Vigna, "Shimware: Toward practical security retrofitting for monolithic firmware images," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2023.

[26] G. Farrelly, P. Quirk, S. S. Kanhere, S. Camtepe, and D. C. Ranasinghe, "Splits: Split input-to-state mapping for effective firmware fuzzing," in *Proceedings of the 28th European Symposium on Research in Computer Security (ESORICS)*, 2023.

[27] F. Gritti, F. Pagani, I. Grishchenko, L. Dresel, N. Redini, C. Kruegel, and G. Vigna, "Heapster: Analyzing the security of dynamic allocators for monolithic firmware images," in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2022.

[28] Hex-Rays, "Ida pro – hex rays," 2023, accessed: Dec 11, 2023. [Online]. Available: https://hex-rays.com/IDA-pro/

[29] N. S. A. R. Directorate, "Ghidra software reverse engineering framework," 2023, accessed: Dec 11, 2023. [Online]. Available: https://ghidra-sre.org/

[30] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries: Tool paper," in *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20*. Springer, 2008, pp. 423–427.

[31] Radare, "Free reversing toolkit - radare," 2023, accessed: Dec 11, 2023. [Online]. Available: https://www.radare.org/n/

[32] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[33] P. De Nicolao, M. Pogliani, M. Polino, M. Carminati, D. Quarta, and S. Zanero, "Elisa: Eliciting isa of raw binaries for fine-grained code and data separation," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, pp. 351–371.

[34] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, p. 282–289.

[35] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 135–150.

[36] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, 2020.

[37] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1237–1254.

[38] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Network and Distributed System Security Symposium*, 2018.

[39] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bootstomp: On the security of bootloaders in mobile devices," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 781–798.

[40] H. Wen, Z. Lin, and Y. Zhang, "Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 167–180.

[41] J. Friebertshäuser, F. Kosterhon, J. Classen, and M. Hollick, "Polypyus–the firmware historian," in *Workshop on Binary Analysis Research (BAR)*, vol. 2021, 2021, p. 21.

[42] "The zephyr project – a proven rtos ecosystem, by developers, for developers," https://www.zephyrproject.org, accessed: Dec 22, 2023.

[43] "Vxworks — industry leading rtos for embedded systems," https://www.windriver.com/products/vxworks, accessed: Dec 22, 2023.

[44] RIOT-OS, "Riot - the friendly os for iot," 2023, accessed: Dec 22, 2023. [Online]. Available: https://www.riot-os.org/

[45] Huawei, "Huawei liteos - lightweight operating system for iot," 2023, accessed: Dec 22, 2023. [Online]. Available: https://www.huaweicloud.com/product/liteos.html

[46] Contiki-NG, "The contiki-ng open source operating system for next generation iot devices," 2023, accessed: Dec 22, 2023. [Online]. Available: https://www.contiki-ng.org/

[47] Embox, "Modular and configurable os for embedded applications," 2023, accessed: Dec 22, 2023. [Online]. Available: https://embox.github.io/
[48] H. Gao, S. Cheng, Y. Xue, and W. Zhang, "A lightweight framework for function name reassignment based on large-scale stripped binaries," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ser. ISSTA 2021. Association for Computing Machinery, 2021.
[49] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX security Symposium*, vol. 13, 2004, pp. 18–18.
[50] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 522–536.
[51] N. Karampatziakis, "Static analysis of binary executables using structural svms," *Advances in Neural Information Processing Systems*, vol. 23, 2010.
[52] J.-Y. Chen, B.-Y. Shen, Q.-H. Ou, W. Yang, and W.-C. Hsu, "Effective code discovery for arm/thumb mixed isa binaries in a static binary translator," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 2013, pp. 1–10.
[53] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," in *proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 364–379.
[54] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, 2005.
[55] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 177–189.
[56] GNU, "The gnu binary utilities - objdump," https://web.mit.edu/gnu/doc/html/binutils_5.html, 2023, accessed: Dec 22, 2023.
[57] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2014, pp. 129–140.
[58] S. Wang, P. Wang, and D. Wu, "Uroboros: Instrumenting stripped binaries with static reassembling," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 236–247.
[59] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in cots binaries," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 201–212.
[60] J. Huang, K. Yang, G. Wang, Z. Shi, S. Lv, and L. Sun, "TaiE: Function identification for monolithic firmware," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 403–414.
[61] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.
[62] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation," in *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 2004, pp. 81–92.
[63] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh, "Bird: Binary interpretation using runtime disassembly," in *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 2006, pp. 12–pp.
[64] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 2003, pp. 265–275.
[65] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.