The Last Mile of Fuzzing: An Efficient Fault Localization Framework for ARM Embedded Firmware

Boyu Chang, Binbin Zhao, *Member, IEEE*, Bo Xu, Qiao Zhang, Peiyu Liu, *Member, IEEE*, Qinge Xie, Guozhu Meng, *Senior Member, IEEE*, and Shouling Ji, *Senior Member, IEEE*

***Abstract*—While fuzzing has been widely adopted and proven effective in exposing vulnerabilities in embedded firmware, fault localization, as the last mile of the vulnerability discovery pipeline, still remains a large challenge. Current post-fuzzing analysis heavily relies on manual debugging, which is hindered by the lack of debugging support in embedded environments and the prevalence of overly tainted, noisy suspicious instructions. These factors impose significant burdens on analysts and slow down vulnerability triage. To bridge this gap, we propose FIRMLOCATOR, a highly efficient and automated framework for fault localization in embedded firmware crashes. FIRMLOCATOR introduces an event-driven memory footprint collection mechanism to capture concrete memory accesses during crash reproduction. It then performs a history-guided propagation analysis to precisely trace data dependencies, reconstructing fine-grained data dependency chains. Finally, it applies heuristic strategies to score and prioritize candidate instructions, providing actionable insights for root cause analysis. We evaluate FIRMLOCATOR on 19 ARM firmware binaries and 59 crashing test cases. The results show that FIRMLOCATOR achieves a localization accuracy of 98.3% within the top 10 instructions, demonstrating its effectiveness and practical value in automated fault localization.**

***Index Terms*—Embedded firmware, fault localization, post-fuzzing.**

Received 9 August 2025; revised 27 June 2026; accepted 5 July 2026. Date of publication 13 July 2026; date of current version 1 September 2026. This work was supported in part by NSFC under Grant U24A20336 and Grant 62302443, in part by the Zhejiang Key Laboratory of Decision Intelligence under Grant 2025E10006, in part by the "Pioneer and Leading Goose" R&D Program of Zhejiang under Grant 2025C02033 and Grant 2025C01082, in part by the Fellowship of China National Postdoctoral Program for Innovative Talents under Grant BX20230307, in part by CAS Project for Young Scientists in Basic Research under Grant YSBR-118, and in part by Beijing Natural Science Foundation under Grant L253025. *(Boyu Chang and Binbin Zhao contributed equally to this work.) (Corresponding author: Shouling Ji.)*

Boyu Chang is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with the State Key Laboratory of Internet Architecture, Tsinghua University, Beijing 100084, China (e-mail: bychang@zju.edu.cn).

Binbin Zhao and Shouling Ji are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: binbinz@zju.edu.cn; sji@zju.edu.cn).

Bo Xu is with the School of Cyberspace Security, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: bogger@hust.edu.cn).

Qiao Zhang is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon Hong Kong (e-mail: qzhangdi@cse.ust.hk).

Peiyu Liu is with the College of Control Science and Engineering, Zhejiang University, Hangzhou 310027, China (e-mail: liupeiyu@zju.edu.cn).

Qinge Xie is with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: qgxie@gatech.edu).

Guozhu Meng is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China (e-mail: mengguozhu@iie.ac.cn).

Digital Object Identifier 10.1109/TDSC.2026.3712723

I. INTRODUCTION

FIRMWARE is a specialized software program embedded in hardware devices, providing essential functionalities for initial configurations, managing communications, and performing I/O operations [1]. It plays a pivotal role in our interconnected world, serving as a critical component of devices employed across various infrastructure sectors, such as advanced manufacturing and intelligent healthcare. Given the ubiquity of firmware and its critical role in the proper functioning of these systems, vulnerabilities in firmware can lead to severe consequences, potentially allowing malicious attackers to install malware [2], steal sensitive data [3], and even remotely control compromised devices [4].

With the growing risk of firmware attacks on embedded devices, considerable efforts are needed to mitigate firmware vulnerabilities. One of the most effective methods for discovering these faults is fuzzing, which involves providing randomly generated test cases to explore program behaviors [5], [6], [7], [8]. When the firmware crashes, the fuzzer saves the corresponding test case for later analysis. However, discovering crashing test cases is only the first step in improving firmware security. Identifying the root causes of firmware crashes is a crucial step after fuzzing, as it provides a deeper understanding of the underlying vulnerabilities.

Given crashing test cases, analysts have to undertake the complex task of manually analyzing their root causes, a process known as *fault localization*. This task is extremely tedious and prone to human error [9], [10]. Over the past few decades, researchers have proposed various automated fault localization techniques. *Spectrum-based Fault Localization* methods involve mutating the initial crashing test case to collect two large sets of crashing and non-crashing test cases with similar execution traces. They then statistically correlate program entities (e.g., statements and instructions) with the observed crash [11], [12], [13]. *Postmortem-based Fault Localization* methods start with debugging files (e.g., core dumps, execution traces, and memory snapshots), then perform reverse execution and backward taint analysis to track the propagation of invalid data [14], [15], [16]. More recently, *Learning-based Fault Localization* approaches

have utilized neural networks to understand the code context, then highlight statements or instructions related to the crash [17], [18], [19], [20].

Despite the variety of techniques proposed for software running on resource-rich systems with abundant debugging support, automated firmware crash analysis remains an underexplored area. Identifying the root cause of a firmware crash is still non-trivial due to the following challenges.

Challenge I. Inadequate debugging mechanisms: Intuitively, analysts require a comprehensive understanding of the crash to figure out its root cause. Runtime information, such as the execution trace, register, and memory values during the execution leading up to the crash, can provide valuable insights for analysts. Unfortunately, fuzzing campaigns towards the embedded firmware lack debugging mechanisms (e.g., sanitizers) to automatically identify and extract key information related to the crash. Analysts have to dive into tens of thousands of instructions only with the support of a basic program debugger or raw logs [5] to manually examine both control flow and data flow, which is extremely tedious, time-consuming, and prone to human mistakes. Considering the lengthy execution trace and rich runtime information, it is challenging to automatically extract and utilize essential runtime information for firmware fault localization.

Challenge II. Limited investigation guidance: The embedded firmware is usually stripped into a raw binary file where debugging symbols (e.g., function and variable names) are stripped away, leaving only a bunch of instructions mixed with data. The lack of semantic information in the stripped firmware binary overwhelms the analysts with an abundance of potentially suspicious instructions that still require non-trivial manual investigation. However, previous works either treat all instructions uniformly [9], [15] or fail to differentiate instructions within the same basic block by design [21], [22]. These over-tainted and noisy instructions offer limited practical guidance for the final manual root cause analysis.

Apart from the aforementioned two challenges, there is another common and crucial challenge towards fault localization. Given the execution trace leading up to a crash, analysts can reversely analyze the instructions to figure out how a pointer is initialized with corrupted data, propagated through function calls, and ultimately dereferenced illegally. However, it is tedious and time-consuming to manually dig out the root cause through tens of thousands of instructions. For the automated root cause analysis, the primary obstacle is the memory alias problem. Memory aliasing, where multiple pointers may point to the same underlying memory location, introduces uncertainty into the analysis of the data flow of the crashing test case and greatly complicates the task of accurately tracking data propagation [15], [16].

Our solution: In light of these challenges, we propose FIRMLOCATOR, a postmortem-based fault localization framework tailored for embedded firmware crashes. FIRMLOCATOR aims to identify faulty instructions of a crash reported by the embedded firmware fuzzer. Our solution consists of three key components:

First, to tackle *Challenge I*, we design an event-driven footprint collection mechanism based on firmware rehosting. By capturing concrete memory accesses during crash reproduction—including instruction address, access type (read/write), involved registers, and memory operands—FIRMLOCATOR enriches the reverse execution phase and significantly accelerates the resolution of memory aliasing.

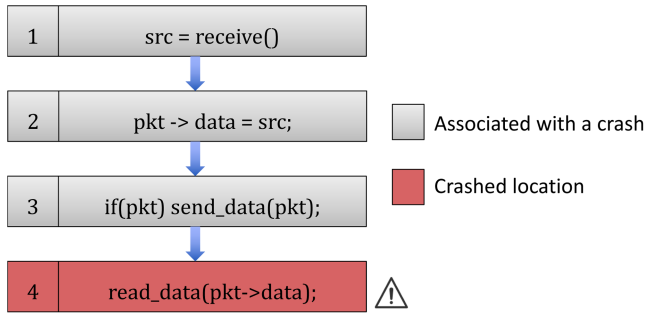

Fig. 1. A simplified crash scenario. A NULL pointer passed through a packet is dereferenced, triggering a crash.

Second, to address *Challenge II*, we develop a two-phase root cause analysis pipeline that computes suspicious scores for tainted instructions. Recognizing that not all instructions contribute equally to a crash, we propose two complementary heuristics to rank instructions based on control flow redundancy and data flow relevance. This prioritization narrows down the candidate set, improving interpretability for analysts.

Third, to mitigate the inherent ambiguity in root-cause attribution, we conservatively select up to five instructions as the root-cause candidate set. These instructions are highly correlated with the data and control dependencies surrounding the crash site, providing a principled balance between comprehensiveness and precision.

Identifying only one as the "sole" cause would be arbitrary and sensitive to compiler optimizations [23]. Determining the "true" root cause at the instruction level is inherently ambiguous in embedded firmware. A single faulty high-level statement is typically compiled into multiple low-level instructions, and none of them alone fully captures the semantic fault. Forcing a single-instruction ground truth is therefore oversimplified and potentially misleading.

To address this, we adopt a pragmatic and semantically grounded evaluation method. For each crashing test case, we manually identify up to five semantically critical instructions that together characterize the faulty logic (e.g., pointer derivation, missing bounds check, unsafe memory dereference). This set is used solely as the ground truth for evaluation. This methodology aligns with the principles of dynamic program slicing, where a fault is viewed as a chain of infected states rather than an isolated point, as shown in Fig. 1.

Importantly, this root-cause instruction set is conceptually distinct from the output of our framework. FIRMLOCATOR produces a ranked list of suspicious instructions. Localization is counted as successful within top k if any instruction in the root-cause set appears in the top k results. This method prevents unfair penalization caused by instruction-level granularity differences and more accurately reflects the cognitive process of a human

analyst who explores the neighborhood of a crash site to understand the failure context [24].

This design is therefore not a trivial extension, but a necessary refinement that more accurately reflects how root causes manifest in optimized embedded firmware binaries.

This paper is an extended and substantially improved version of our previous conference work [25]. While the core idea of event-assisted reverse execution was introduced earlier, this journal version significantly advances the framework in several important aspects.

In summary, we make the following contributions.

- *Enhanced root-cause ranking mechanism:* We redesign and refine the heuristic scoring strategies used for ranking suspicious instructions. Unlike the conference version, where heuristic interactions could occasionally cause negative optimization effects, the new design balances multiple heuristics and achieves more stable and robust ranking results across diverse datasets.
- *Enhanced ISA support for reverse execution:* We substantially broaden the instruction set support of our reverse execution engine, extending the reversely modeled ARM Cortex-M instruction classes from 29 to 43. By aligning with the ARMv7-M architecture specifications [26], this extension ensures semantic continuity during back-tracing and significantly increases the framework's applicability to highly optimized, real-world firmware.
- *Refined root-cause labeling methodology:* We introduce a principled root-cause attribution protocol that allows up to five semantically critical instructions to be labeled as ground truth, explicitly accounting for instruction-level ambiguity and aligning human judgment with ranked outputs.
- *Broadened and diversified evaluation:* Beyond the original Fuzzware dataset, we incorporate additional benchmarks from GDMA, Hoedur, and other public repositories [27], [28], [29], [30], [31], yielding a more diverse collection of firmware images and crashing inputs. Compared to state-of-the-art works, FIRMLOCATOR demonstrates its superiority in 20.3% improvement in full execution trace analysis capability, polynomial-level acceleration in overall efficiency and 62.7% higher success rate within the top 10 instructions in effectiveness.

To foster research on this topic, we open source FIRMLOCATOR at https://github.com/NESA-Lab/FirmLocator.

II. BACKGROUND

Firmware rehosting: In firmware analysis, researchers propose firmware rehosting due to the slow nature and high costs of using actual hardware [32]. Firmware rehosting is a technique to execute firmware binaries in an emulated environment instead of their original hardware devices. The emulation process replaces the need for the physical hardware, allowing users to analyze and even manipulate firmware code in a controlled virtual setting. Due to the effectiveness of firmware rehosting, recent works have adopted this technique for vulnerability discovery [5], [33], [34].

In the context of firmware rehosting, the hook function is a powerful method for intercepting specific actions. For instance, when using Unicorn [35] to emulate a firmware binary, one can design and implement a `simple_test` function, and register it with the `UC_HOOK_CODE` option. When executing the binary, the `simple_test` function is called before the program executes any instructions. Although hooks provide analysts with the ability to monitor the execution process, it is still hard to collect key runtime information related to the crash for the sake of both effectiveness and efficiency in fault localization. In this work, we propose an event-based footprint collection method on top of firmware rehosting to facilitate fault localization.

Reverse execution: In the field of postmortem-based fault localization, analysts perform reverse execution to gain deeper insights into program crashes [15], [16]. The key idea of reverse execution is to *undo* the executed instruction to revert registers and memory values to their previous states, providing analysts with a unique perspective on the evolution of program state over time. To undo the executed instruction, analysts usually design complicated handlers that inversely execute the instruction according to its behavior (i.e., inverse handlers). Besides, the process of reverse execution heavily relies on runtime data collected from the system, typically including detailed logs, memory dumps, and other relevant data that encapsulate the state of the system. In this work, we utilize our collected footprint to improve reverse execution and design multiple inverse handlers for the 32-bit ARM Cortex-M embedded firmware.

Use-define chain: In the field of data flow analysis, a Use-Define Chain (UD Chain) is a data structure that describes a use and all the definitions of a variable in the program, where that variable can reach the use without any other intervening definitions [36]. The use-define chain consists of a bunch of use nodes and define nodes, providing a way to track the data flow by establishing a connection between uses and definitions of different variables. In the firmware analysis scenario, constructing an accurate use-define chain is non-trivial due to the closed-source and symbol-stripped nature of firmware. In this work, we propose an automated method that utilizes the execution trace and disassembled firmware binary to improve the effectiveness of use-define chain construction, upon which we further analyze the propagation of invalid data and taint relevant instructions.

III. OVERVIEW

This section presents the problem scope and design goal of FIRMLOCATOR, followed by a high-level overview of its architecture and key components.

A. Problem Scope

FIRMLOCATOR is designed to automate the identification of root causes in embedded firmware crashes, particularly after a fuzzing campaign based on firmware rehosting [5]. Given a large number of crashing test cases produced during fuzzing, the primary objective of FIRMLOCATOR is to efficiently pinpoint the critical instructions that contribute to each crash, thereby narrowing the scope for human analysts. By highlighting and ranking suspicious instructions, FIRMLOCATOR supports both

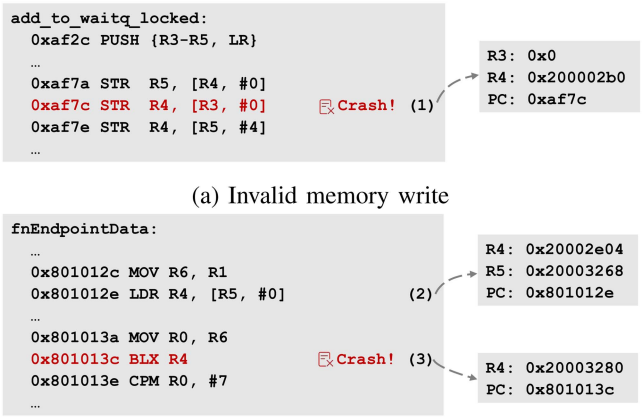

Fig. 2. Two instruction snippets of direct reasons to trigger firmware crashes. The relevant register values before the execution of the instruction are shown on the right. The snippet (a) shows an invalid memory write, and the snippet (b) shows an invalid instruction execution.

novice and experienced analysts in postmortem vulnerability analysis.

The root causes of real-world firmware crashes are often complex and multifaceted, including issues such as use-after-free, double free, and out-of-bounds writes. Manually modeling each type of vulnerability is neither scalable nor practical. However, at the machine instruction level, most crashes can be traced back to two fundamental causes: invalid memory accesses and invalid instruction executions [9]. In this work, we refer to these two reasons as the direct reasons for crashes. Fig. 2 shows two instruction snippets of direct reasons. The relevant register values before executing the instruction are detailed on the right. Fig. 2(a) shows an instruction snippet of the crash site for CVE-2021-3329. When executing `STR R4, [R3, #0]` at `(1)`, the program tries to store the value of `R4` (i.e., `0x200002b0`) in the memory at the address `R3 + 0`. However, the register `R3` happens to be zero, which triggers an unmapped memory access at `R3 + 0` (i.e., zero). Then, the program crashes because of an invalid memory write. Fig. 2(b) illustrates another crash reason. When the `BLX R4` instruction is executed, the program attempts to branch to the address of `R4` (i.e., `0x20003280`), which is a function pointer in the memory area. However, the function pointer references an invalid instruction (i.e., a NULL pointer) at runtime, resulting in a crash. While these direct reasons may not always be the ultimate root causes in all cases, they provide FIRMLOCATOR with the key information (i.e., `R3` in Fig. 2(a) and `R4` in Fig. 2(b)) to perform fault localization, serving as a first step in revealing the underlying issues. Accordingly, instead of modeling tremendous crash types, the FIRMLOCATOR framework is designed to handle these two direct reasons to maximize scalability and applicability.

B. Architecture

Fig. 3 illustrates the high-level architecture of FIRMLOCATOR, which consists of three key components: footprint collection, reverse execution, and root cause analysis.

Footprint collection: To facilitate effective crash diagnosis, FIRMLOCATOR begins by replaying the crashing test case on the firmware binary in a rehosted environment. During this phase, it captures runtime events including memory accesses and control transfers using an event-driven footprint mechanism. This design compensates for the lack of debugging interfaces in embedded systems and provides essential runtime context for subsequent analysis.

Reverse execution: The main purpose of reverse execution is to construct the dynamic propagation of data in a crashing test case. To tackle the prevalent issue of memory aliases that can obscure data flow analysis, FIRMLOCATOR employs a history-driven reverse execution method. This method leverages the concrete memory access data from the footprint collection phase to precisely resolve aliases and track dependencies.

Root cause analysis: As described in Section III-A, FIRMLOCATOR focuses on crashes triggered by invalid memory or instruction operations. The memory address or register involved in the failure is treated as the taint sink. FIRMLOCATOR first applies backward taint analysis to collect all instructions involved in propagating data to the crash point.

Subsequently, a two-phase root cause analysis is performed: first, tainted instructions are extracted via standard backward analysis; then, they are scored and ranked based on behavior-specific heuristics that reflect control flow patterns and memory history relevance. This ranking highlights the most critical instructions for root cause diagnosis and provides actionable insight to human analysts.

IV. DESIGN

This section details the design of FIRMLOCATOR and its core components. The main workflow is as follows. First, FIRMLOCATOR accepts a crashing test case generated by fuzzers and the corresponding firmware binary as inputs. Second, in the footprint collection phase, FIRMLOCATOR reproduces the crash to collect runtime data, including instruction addresses, register and memory values. Third, the reverse execution component uses action events to build a use-define chain and data events to recover unknown data to further construct a data-based use-define chain. Then, FIRMLOCATOR carries out backward taint analysis along the use-define chain to identify instructions related to the crash. Finally, FIRMLOCATOR adopts heuristic ranking strategies to highlight key instructions as the root cause for analysts to investigate.

A. Footprint Collection

To support precise and scalable crash diagnosis, FIRMLOCATOR introduces an event-based footprint collection strategy atop firmware rehosting. Specifically, FIRMLOCATOR identifies two types of key events for understanding firmware crashes, including data events and action events. In the following, we introduce details of the event-based footprint collection methodology of FIRMLOCATOR.

1) Data Events: Many firmware crashes are triggered by unexpected memory accesses. Fig. 4 shows an example of a crash due to an invalid read. The program first stores

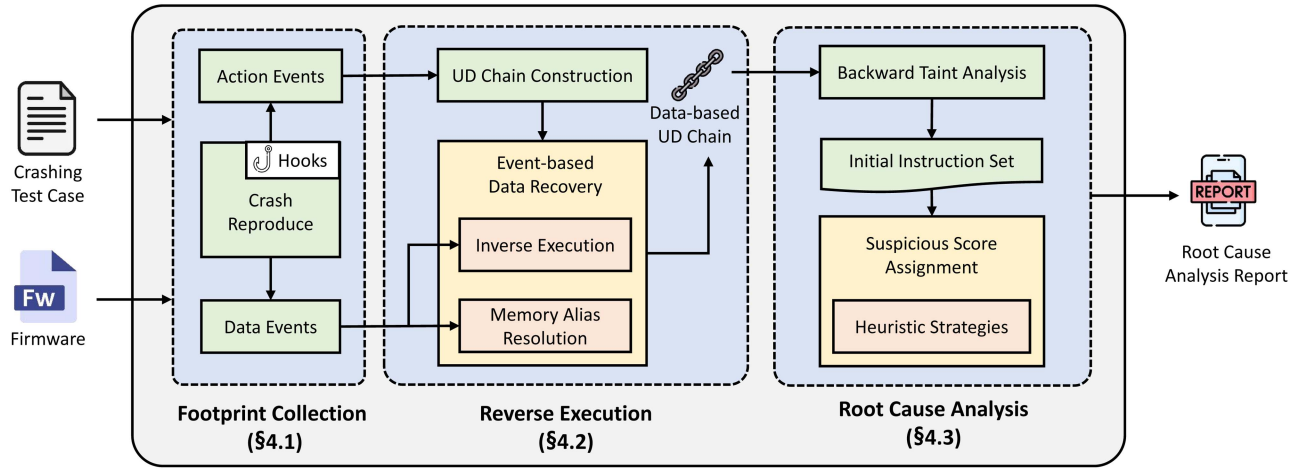

Fig. 3. Overview of FIRMLOCATOR. The architecture of FIRMLOCATOR consists of three components, including footprint collection, reverse execution, and root cause analysis.

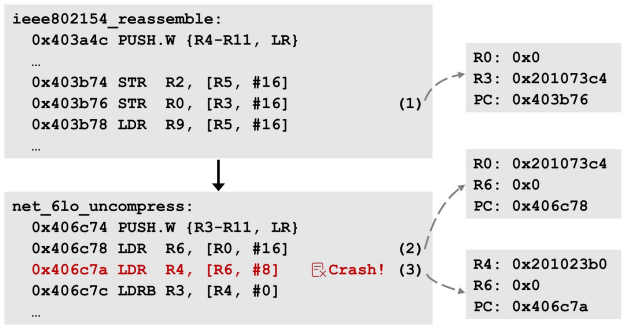

Fig. 4. An instruction snippet of CVE-2021-3322, where a radio frame is inserted and marked for fragmentation. The frame contains the full payload size, leading to an unexpected pointer alias, which finally results in an unexpected NULL pointer dereference in `net_6lo_uncompress` function.

`R0` in the memory at the address `R3 + 16` at `(1)` in `ieee802154_reassemble` function, then loads value from the address of `R0 + 16` and assigns it to `R6` at `(2)` in `net_6lo_uncompress` function. Finally, the program loads a value from the address of `R6 + 8` which results in a NULL pointer dereference at `(3)`.

However, given a crashing test case and the firmware binary, it is non-trivial to obtain such knowledge to understand the crash. Analysts have to dive into runtime details, examine every instruction from the crash site and finally figure out the fact that it is `R0 + 16` and `R3 + 16` pointing to the same address (i.e., `0x201073d4`) together with an invalid value (i.e., zero) of `R0` at `(1)` that induces the crash. As there are a large number of instructions that are executed between `(1)` and `(2)`, it is time-consuming and error-prone to manually analyze the crash reason. In addition to manual analysis, the effectiveness of automated analysis methods is also limited by a large number of instructions. Existing automated analysis methods rely on the correct recovery of registers at the crash site and memory data from crash reports or core dump files [9], [15], [16], [37]. However, these methods can only recover a part of the data values along the execution trace due to inherent analysis limitations (e.g., memory aliases and irreversible instructions). The longer the execution trace to analyze, the less data can be recovered, and the less accurate the result will be. Therefore, it is difficult to pinpoint the exact root cause.

To overcome this difficulty, we propose a history-driven method that leverages concrete memory accesses to reconstruct the data flow for the crashing test case. FIRMLOCATOR views the memory access behavior as a data event. Memory access is either a read or write, thus FIRMLOCATOR pays attention to two types of data events. One data event is a successful memory read and the other is a successful memory write. FIRMLOCATOR monitors these two events and immediately pauses emulation as soon as either event occurs. Then, FIRMLOCATOR saves the details of the data event. We design a data structure to store the type of memory access (i.e. read or write), the address of the instruction being executed, the actual data being read (written), and the corresponding memory address. Based on data events, FIRMLOCATOR obtains concrete memory access information and efficiently understands the data flow between memory reads and writes even for extremely long execution traces.

2) Action Events: In addition to the memory access history captured as data events, the execution trace is another crucial piece of information that is utilized by FIRMLOCATOR to enable effective fault localization. We consider the execution trace as a sequence of instructions executed by the firmware. Through the analysis of the execution trace, analysts can understand the behavior of the program leading up to the crash, thereby focusing their analysis on the root cause more effectively.

However, recovering the complete execution trace of a crashing test case is non-trivial. A naïve approach would involve tracing backward from the crash site, following the `PC` values saved on the stack during function calls. The key challenge lies in handling inconsecutive instructions, which can hinder both intraprocedural and interprocedural analysis. Fig. 5 illustrates two main kinds of inconsecutive instructions in intraprocedural and interprocedural analysis. For intraprocedural analysis, the key issue is the inability to reliably resolve conditional branches

```
ieee802154_recv:
  0x40d126 PUSH.W {R4-R8, LR}
  …
  0x40d1de BL   0x40d0a6 <net_buf_simple_pull>          (1)
  0x40d1e2 LDR  R0, [R4, #40]
  0x40d1e4 CBZ  R0, 0x40d1f2 <ieee802154_recv+0xcc>     (2)
  0x40d1e6 LDRB R3, [R4, #44]
  0x40d1ea CMP  R3, #8
  0x40d1ec BNE  0x40d1f2 <ieee802154_recv+0xcc>         (3)
  0x40d1ee BL   0x40d10a <sys_mem_swap>                 (4)
  0x40d1f2 LDR  R0, [R4, #48]                           (5)
  …
  0x40d21c MOV  R0, R4
  0x40d21e BL   0x403a4c <ieee802154_reassemble>        (6)
  …
```

Fig. 5. An instruction snippet of the firmware for CVE-2021-3322. Inconsecutive instructions for intraprocedural and interprocedural analysis are highlighted in different colors.

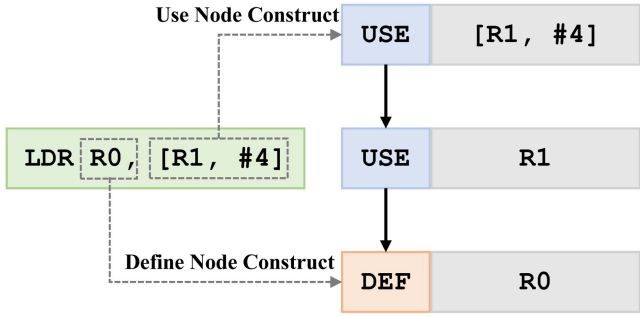

Fig. 6. An example of constructing the use-define chain for the instruction `LDR R0, [R1, #4]`. The source operand `[R1, #4]` and its base register `R1` generate two use nodes. The destination operand `R0` generates a define node.

without access to the actual runtime register values. Although we can construct a control flow graph and find all preceding blocks of any instruction with static forward analysis, it is still challenging to determine the correct block because of the lack of accurate register values for conditional branches. Consider the example instruction `LDR R0, [R4, #48]` at `(5)`: while static analysis can identify the two preceding instructions at `(2)` and `(3)`, the correct preceding block ultimately depends on the values of registers `R0` and `R3`, which are unknown. Incorporating these implicit register dependencies into the analysis can introduce prohibitive overhead, rendering the approach impractical. Similarly, for interprocedural analysis, the challenge lies in the lack of visibility into sibling function executions. Even if the call chain can be reconstructed from the stack trace, there is no way to know whether a relevant function has been invoked or not, as its stack frame would have been popped off. In the example of the `BL 0x403a4c` at `(6)`, the stack frames for `net_buf_simple_pull` at `(1)` and `sys_mem_swap` at `(4)` have already been removed, potentially obscuring crucial context for understanding the root cause.

To address this challenge, we hold the idea of breaking the whole into parts and propose a trigger-based method to first collect each execution of one instruction and then recover the complete execution trace. More specifically, FIRMLOCATOR first regards the execution of one instruction as an *action event*, which is triggered when an instruction is about to be executed. Since ARM architecture stores the next instruction's address in the `PC` register, FIRMLOCATOR intercepts execution at each action event and logs the value of `PC`. Finally, FIRMLOCATOR successfully recovers the complete execution trace by combining the collected action events, which helps identify the root cause that is far from the crash site.

B. Reverse Execution

Through the event-based footprint collection phase, FIRMLOCATOR obtains runtime data consisting of memory accesses and executed instruction addresses. In this section, we introduce how FIRMLOCATOR utilizes this information to perform the reverse execution for further understanding the propagation of the corrupted data.

For an invalid memory access or an invalid instruction execution that triggers a crash, the basic idea for identifying the root cause is to reversely examine instructions in the execution trace from the crash site and figure out why the corresponding data has been corrupted, which is called reverse execution. Unfortunately, even with a complete execution trace available, determining the precise data propagation and correlating individual instructions is a non-trivial challenge, primarily due to the memory alias problem. Consider again the example in Fig. 4, where the instructions at `(1)` and `(2)` access the same memory address (i.e., `0x201073d4`). If we could establish this correlation, we could then blame the instruction at `(1)` as the root cause of the crash. However, the main difficulty is that the concrete memory addresses involved in these instructions are often obscured by the presence of irreversible instructions. Take the instruction `MOV R3, #0` that sets the register `R3` to zero for example. We only know that `R3` holds zero after execution, and there is no hint to recover the original value of `R3` prior to the execution of this instruction. In this case, the reverse execution fails to handle other instructions that use the old value of `R3` before the instruction `MOV R3, #0`. To tackle this challenge, FIRMLOCATOR adopts a two-step history-driven method to perform the reverse execution. First, we construct a use-define chain for analyzing execution traces. Then, on top of the use-define chain, we inversely execute the instruction to recover registers into previous states. We detail our design as follows.

1) Use-Define Chain Construction: The first step in FIRMLOCATOR's reverse execution process is to iteratively reconstruct the complete execution trace using the action events collected during the footprint collection phase. For each instruction encountered, FIRMLOCATOR carefully parses the instruction's behavior, extracting the source and destination operands. Fig. 6 illustrates this use-define chain construction process. Taking `LDR R0, [R1, #4]` as an example. In this case, the firmware reads four bytes from the memory address of `R1 + 4` and stores them in `R0`. The operand `[R1, #4]` is considered a source operand, and `R0` is considered a destination operand. These source and destination operands represent the data flow within the instruction. To obtain the data flow between instructions, FIRMLOCATOR further utilizes the two operands to generate *use nodes* and *define nodes*. Specifically, a use node describes

a read behavior for a register or memory address, while a define node represents a write behavior. For the `LDR R0, [R1, #4]` instruction, the destination operand `R0` is used to create a define node, and the source operand `[R1, #4]` is used to create a use node. Additionally, FIRMLOCATOR also creates extra use nodes for the base register and index register involved in the memory operand. It is important to note that a single instruction may generate multiple explicit and implicit use and define nodes, as FIRMLOCATOR thoroughly analyzes the semantics of each instruction to identify all the relevant source and destination operands. As FIRMLOCATOR parses each new instruction in the execution trace, the corresponding use and define nodes are appended to the growing use-define chain. This carefully constructed use-define chain serves as the foundation for the subsequent data recovery step in FIRMLOCATOR's reverse execution process.

2) Event-Based Data Recovery: After constructing the use-define chain, the next step in FIRMLOCATOR's reverse execution process is to inversely recover the register values preceding each instruction's execution. To achieve this, FIRMLOCATOR employs custom handlers designed to handle specific types of instructions. We selected 43 ARM Cortex-M instructions that appear frequently and have a significant impact on data flow and control flow from the actual execution of some real cases. For instance, we design `add_handler` for the add operation. From the instruction `add R0, R1`, we derive `R0' = R0 + R1`, `R0 = R0' - R1` and `R1 = R0' - R0` where `R0'` denotes the value of register `R0` after execution. If there is only one unknown value in three operands, we can recover it using the other two values and update the corresponding node. Whenever a node in the use-define chain is updated, FIRMLOCATOR examines the entire chain for other possible updates that can be propagated. This iterative process continues until the use-define chain converges, at which point FIRMLOCATOR has recovered as many previous states as possible. However, these methods are insufficient to process the node with memory alias, as aforementioned in Fig. 4. To solve this problem, FIRMLOCATOR utilizes data events to recover the actual runtime data. For example, consider the `STR R0, [R1, #4]` instruction that stores the value of `R0` in the memory at the address `R1 + 4`. During the previous footprint collection phase, FIRMLOCATOR has obtained the target address (i.e., `[R1, #4]`) and concrete data to be stored. Armed with this data event, FIRMLOCATOR can directly recover the value of `R0` from the concrete data, and the value of `R1` by subtracting 4 bytes from the target address. Compared to other techniques such as hypothesis testing [9] and value-set analysis [37], the data event not only guarantees accuracy, but also greatly accelerates reverse execution.

C. Root Cause Analysis

In this section, we introduce a novel two-phase root cause analysis method. First, FIRMLOCATOR performs common backward taint analysis to obtain an initial instruction set, where the instructions are associated with the root cause. Then, FIRMLOCATOR assigns suspicious scores for all tainted instructions to rank and highlight the instructions that have higher priorities in further investigation.

Algorithm 1: Taint Propagation Algorithm.

Input: C, Use-define chain that consists of use and define nodes
Input: t, A register or memory address that is identified as a taint sink
Output: T, Tainted instruction set

```
T = φ ;
u ← MakeUseNode(t);
S = {u};
while IsNotEmpty(S) do
    h ← GetAndRemoveElement(S);
    T ← T ∪ FindInstruction(h);
    d ← FindDefineNode(C, h);
    T ← T ∪ FindInstruction(d);
    U ← FindUseNodes(C, d);
    S ← S ∪ U;
end
```

1) Backward Taint Analysis: The backward taint analysis starts by identifying the direct reason for the crash. FIRMLOCATOR then leverages this information to track the propagation of invalid data backward through the execution trace. In the following, we introduce how FIRMLOCATOR sets the taint sink and taints instructions.

Taint sink: The taint sink is the starting point of backward taint analysis. FIRMLOCATOR identifies concrete memory addresses or registers as the taint sink according to the instruction type at the crash site. For invalid memory read (write) instructions, both the base and index registers are marked as taint sinks. As for invalid instruction execution, we further check where the firmware loads the instruction address. In the case of invalid instruction execution, the `PC` register, which holds the address of the next instruction, could be explicitly or implicitly modified. For instructions that explicitly modify `PC` (e.g., `POP`), the stack address from which `PC` loads the target address is identified as the taint sink. For other instructions that implicitly modify `PC` (e.g., `BLX`), the used register is identified as the taint sink.

Propagation rules: FIRMLOCATOR follows a systematic algorithm to propagate the taint backward through the use-define chain. Starting with the taint sink, it iteratively locates the define nodes and their corresponding use nodes, adding the associated instructions to the tainted set. This process continues until there are no more nodes to process. Algorithm 1 shows how FIRMLOCATOR propagates the taint. First, FIRMLOCATOR converts the taint sink to a use node and adds the node into an empty set S. Then, for every node u in S, FIRMLOCATOR leverages the use-define chain to locate its define node d, then removes u from S. The instructions associated with d and u are then tainted. The data source of d is used to locate its use nodes set U in the use-define chain. FIRMLOCATOR adds nodes set U to S. This backward tainting is executed repeatedly until there is no node in S. By the end of the backward taint analysis, FIRMLOCATOR has identified the initial set of tainted instructions

in T, which represent the potential root causes of the observed crash.

2) Suspicious Score Assignment: Now we revisit the strategy of automated root cause analysis. While the previously introduced history-driven approach effectively resolves memory aliasing, it often results in excessively long use-define chains. Consequently, too many instructions may be identified as potential root causes, which imposes a substantial cognitive burden on analysts.

To improve the efficiency and accuracy of postmortem-based fault localization, we refine the instruction ranking mechanism by introducing two heuristic strategies. Our core insight is that tainted instructions contribute differently to the root cause, and a uniform treatment of all instructions results in suboptimal guidance. For instance, in the case of the `memcpy` function, which iterates through byte-by-byte copying, the repetition of identical instructions occurs frequently. However, the instructions involving loading the target address and copying length hold greater significance, as they pertain to the initialization and termination conditions of the loop. Intuitively, such instructions should have higher ranks. This aids analysts in prioritizing attention toward more critical instructions.

FIRMLOCATOR assigns suspicious scores in two steps. First, each tainted instruction receives an initial score based on its execution order—earlier instructions are assigned higher base scores. Second, we heuristically adjust the scores by modeling instruction behavior, focusing on control and data flow characteristics. In particular, we propose two strategies: (1) a *redundant loop taint suppression* strategy that penalizes repetitive loop instructions, and (2) a *history write taint prioritization* strategy that emphasizes the importance of distant memory writes related to the crash. In the following, we detail the design of these two strategies.

This design intuition is further supported by empirical observations from real-world vulnerability patches. Prior studies on memory corruption and firmware vulnerabilities [38] show that the majority of fixes (approximately 70%) concentrate on semantically critical instructions that govern data propagation and control-flow decisions, rather than on repetitive value-manipulation operations. In particular, large-scale analyses of vulnerability patches report that adding or modifying input validation logic and control-flow guarding conditions (e.g., conditional branches and early exits) account for the dominant portion of fixes. These instructions typically determine whether unsafe data flows are triggered or prevented and therefore play a central role in the manifestation of crashes.

Motivated by this observation, our ranking heuristics prioritize instructions with strong data-flow and control-flow semantics while suppressing the influence of redundant instructions arising from tight loops. By aligning the scoring mechanism with widely observed vulnerability-fixing patterns, FIRMLOCATOR aims to provide more generally applicable and practically meaningful ranking results.

Redundant loop taint suppression: In embedded systems, repetitive instructions caused by tight loops (e.g., `memcpy`-like patterns) may dominate the execution trace. We define such sequences as redundant loops and reduce their contribution to the suspicious score. The `memcpy` function requires three arguments, including two address pointers ptr_1 and ptr_2, and an integer n. While executing, `memcpy` repeatedly copies one byte from ptr_2 to ptr_1 for n times. If n is larger than the capacity of the data structure that ptr_1 or ptr_2 holds, `memcpy` will read or write an illegal address, overwrite other variables, and trigger a firmware crash. In this case, n is usually a large number, resulting in an extremely long execution trace. This leads to a lot of tainted instructions in backward tainting analysis. Although these repetitive instructions are related to the crash, they are simply a sequence of value calculations and data copies that do not provide analysts with any insight into the crash. Our insight is that, the repeatedly executed consecutive instructions are less important than the initialization and termination conditions of the loop. Therefore, we decrease the suspicious score of repeatedly executed consecutive instructions.

History write taint prioritization: Another critical behavior we consider is the history write, which refers to instructions that write to the exact memory address related to the crash site. Our observation is that for a deep root cause, the taint propagation can be extremely long, which taints many instructions. In such cases, the taint source instructions, which perform history writes and are far away from the taint sink, are more important than others along the taint propagation path. Therefore, we increase the suspicious scores of these taint source instructions.

Finally, FIRMLOCATOR sorts all tainted instructions according to their suspicious scores. Instructions with higher scores are given greater priorities and ranks, allowing analysts to focus their attention on the most relevant instructions during the root cause analysis.

V. IMPLEMENTATION

We implement the prototype of FIRMLOCATOR for the purpose of automatically analyzing the root cause of firmware crashes on the 32-bit ARM Cortex-M architecture. FIRMLOCATOR mainly consists of three components, including footprint collection, reverse execution, and root cause analysis. In the following, we introduce the implementation details of these components.

Footprint collection: To capture the desired runtime data during emulation, we develop two kinds of events on top of Unicorn's [35] hooking mechanism. Specifically, we implement conditional callbacks for these two events, including `UC_HOOK_MEM_READ_AFTER` and `UC_HOOK_MEM_WRITE` for data events, and `UC_HOOK_CODE` for action events. By registering these callbacks, we are able to precisely monitor and record the relevant program behaviors and data accesses during the crash reproducing process.

Reverse execution: To analyze the raw binary firmware, we develop an instruction analyzer on top of the widely adopted Capstone disassembly framework [39]. Capstone provides semantic details about the disassembled instructions, including implicit register reads and writes. We design a handler to examine the source and destination operands of each instruction in action events, then create corresponding use and define nodes in the program representation. When the instruction accesses memory locations that have been previously captured by our

data event monitoring, our handler is able to extract the concrete address and data values. This eliminates the need for expensive memory alias resolution, allowing accurate analysis of the full execution trace. Besides, we implement multiple unique instruction handlers to perform inverse transformations on registers to facilitate reverse execution.

Root cause analysis: To identify the critical program points that could lead to potential crashes, we develop a set of taint sink rules covering all relevant instructions. By leveraging our ability to precisely track memory accesses, as described earlier, we are able to apply these taint rules effectively. However, this also induces the risk of over-tainting, where excessive numbers of instructions get flagged as potentially suspicious. To address this challenge, we design a novel heuristic ranking strategy to highlight the most instructive taint sinks. The ranking strategy, as described in Section IV-C2, allows FIRMLOCATOR to focus on reporting the instructions that provide the most valuable guidance for understanding the observed crashes.

VI. EVALUATION

In this section, we assess the performance and capabilities of our FIRMLOCATOR framework from multiple perspectives. First, we examine whether FIRMLOCATOR can accurately identify the root causes of observed firmware crashes. Next, we measure the time breakdown of each component of FIRMLOCATOR and the overall efficiency. Then, we conduct an ablation study to measure the contribution of our heuristic ranking strategy. Finally, we evaluate the event-based design to demonstrate its contribution to FIRMLOCATOR.

We compare FIRMLOCATOR against two of the latest postmortem-based root cause analysis works, POMP [9] and POMP++ [37]. We choose these works because they and FIRMLOCATOR consider the semantics of instructions, which provide more instructive results for analysts than spectrum-based methods [22]. POMP performs backward taint analysis on a core dump file and the execution trace collected by Intel PT [40] to highlight the root cause of a crash. POMP++ further enhances POMP in memory alias resolving with value-set analysis. While these approaches are designed for the x86 Linux platform, we have implemented them on the 32-bit ARM Cortex-M architecture to ensure a fair comparison. We carefully inspect the source code of POMP and POMP++ and migrate them to the 32-bit ARM Cortex-M architecture in the following aspects.

- *Runtime information collection:* POMP and POMP++ rely on hardware feature support (i.e., Intel PT) to obtain the complete execution trace. In order to provide the same level of information, we examine the data structure used by POMP and POMP++ and modify our event-based footprint collection method to offer data in their formats.
- *Reverse execution:* POMP and POMP++ target x86 software, which is totally different from firmware in terms of the instruction set architecture. We design inverse handlers for the 32-bit ARM Cortex-M architecture for the inverse execution of the reverse execution module. In our comparison, POMP, POMP++, and FIRMLOCATOR share the same inverse handlers to prevent extra overhead and side effects.
- *Value-set analysis:* POMP++ employs a customized value-set analysis and leverages static information (e.g., memory region) to solve the memory alias problem. We also migrate their method to the 32-bit ARM Cortex-M architecture to identify memory regions for a fair comparison. Note that POMP++ also utilizes heuristics that only exist in the x86 architecture, such as a special function `get_pc_thunk.cx`, to improve the region identification. We ignore such heuristics because they do not apply in firmware scenarios.

We acknowledge that FIRMLOCATOR and POMP/POMP++ do not share exactly the same scenario, which may introduce performance differences. Our evaluation demonstrates the significant advancements that FIRMLOCATOR brings over these state-of-the-art works in terms of overall accuracy, efficiency, and full execution trace analysis capability. In summary, we evaluate FIRMLOCATOR with the following research questions:

RQ1: What is the ability of FIRMLOCATOR to identify the root cause of an embedded firmware crash? (Section VI-B)

RQ2: What is the overhead of every component of FIRMLOCATOR, and how is the overall efficiency? (Section VI-C)

RQ3: What is the contribution of heuristic ranking strategies? (Section VI-D)

RQ4: How does the event-based design contribute to FIRMLOCATOR's performance? (Section VI-E)

A. Experimental Setup

Experiment settings: We perform the evaluation under the same experiment setting: a 56-core Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40 GHz machine running Ubuntu 22.04.2 LTS with 256 GB RAM.

Dataset and ground truth: To conduct a comprehensive evaluation, we utilize the public datasets provided by the Fuzzware framework [27], [28], [29], [30], [31]. These datasets contain 81 test suites that cover a wide range of crashing behaviors. Each test suite in the dataset includes the raw binary firmware, a basic configuration file, a crashing test case, and a crash description. We follow the official guidance of Fuzzware to deploy the framework in its reproducing mode and successfully reproduce 59 test cases that exhibit crashing behaviors. We assign crash IDs from C_1 to C_{59} for these test cases, which are detailed in Table I.

For each of these test suites, we employ a two-pronged approach to establish the ground truth. First, we extract the instructions identified in the provided crash descriptions as the root causes. Then, for those test suites lacking explicit descriptions, we manually inspect the assembly code of their firmware images and leverage execution logs to identify root causes from their crash sites.

Sensitivity analysis of ground-truth labeling and evaluation threshold: The size of the labeled ground-truth instruction set and the evaluation threshold jointly affect the validity of the reported fault localization. With a small set size, the evaluation may collapse a crash-inducing causal chain into a single instruction and underestimate valid localization results. With a large set size, the evaluation may over-credit loosely related instructions and inflate the fault localization success rate. Similarly, a small

TABLE I
HARDWARE PLATFORMS, FIRMWARE SAMPLES, AND DIRECT CRASH REASONS CORRESPONDING TO CRASH IDS IN FIRMLOCATOR'S EVALUATION

ID	Hardware Platform	Firmware Sample	CVE Number
C_1	ARCH_PRO	Pw_Discovery	-
C_2	EFM32GG_STK3700	Pw_Discovery	-
C_3	EFM32LG_STK3600	Pw_Discovery	-
C_4	LPC1549	Pw_Discovery	-
C_5	LPC1768	Pw_Discovery	-
C_6	MOTE_L152RC	Pw_Discovery	-
C_7	NUCLEO_F103RB	Pw_Discovery	-
C_8	NUCLEO_L152RE	Pw_Discovery	-
C_9	UBLOX_C027	Pw_Discovery	-
C_{10}	STM32F429ZI	CNC	-
C_{11}	STM32F103RB	Gateway	-
C_{12}	STM32F429ZI	PLC	-
C_{13}	STM32F429ZI	PLC	-
C_{14}	STM32F103RB	Robot	-
C_{15}	STM32F103RB	Gateway	-
C_{16}	STM32F103RB	Gateway	-
C_{17}	STM32F103RB	Gateway	-
C_{18}	STM32F103RB	Gateway	-
C_{19}	STM32F429ZI	PLC	-
C_{20}	STM32F429ZI	utasker_USB	-
C_{21}	MAX32600	Thermostat	-
C_{22}	MAX32600	RF_Door_Lock	-
C_{23}	SAMR21	6LoWPAN_Receiver	-
C_{24}	SAMR21	6LoWPAN_Sender	-
C_{25}	MAX32600	RF_Door_Lock	-
C_{26}	STM32F103RE	3DPrinter	-
C_{27}	STM32F429ZI	utasker_MODBUS	-
C_{28}	STM32F429ZI	utasker_MODBUS	-
C_{29}	STM32L432KC	Zepyhr_SocketCan	-
C_{30}	STM32L432KC	Zepyhr_SocketCan	-
C_{31}	STM32F429ZI	utasker_USB	-
C_{32}	SAM4E_XPRO	Socket_Echo_Server	CVE-2020-10064
C_{33}	DISCO_L475_IOT1	Bluetooth_Peripheral_HIDs	CVE-2020-10065
C_{34}	DISCO_L475_IOT1	Bluetooth_Peripheral_HIDs	CVE-2020-10066
C_{35}	SAM4S_XPLAINED	Socket_Echo_Server	CVE-2021-3322
C_{36}	SAM4S_XPLAINED	Socket_Echo_Server	CVE-2021-3323
C_{37}	NRF52840	Bluetooth_Peripheral_HIDs	CVE-2021-3329
C_{38}	SAM4S_XPLAINED	Bluetooth_Peripheral_HIDs	CVE-2021-3330
C_{39}	CC2538	hello-world	CVE-2020-12140
C_{40}	CC2538	snmp-server	CVE-2020-12141
C_{41}	CC2538	hello-world	-
C_{42}	NRF52840	hello-world	CVE-2023-48229
C_{43}	CC2538	hello-world	CVE-2022-41873
C_{44}	CC2538	hello-world	CVE-2022-41972
C_{45}	CC2538	gnrc_networking	CVE-2023-24817
C_{46}	CC2538	gnrc_networking	CVE-2023-24818
C_{47}	CC2538	gnrc_networking	CVE-2023-24819
C_{48}	CC2538	gnrc_networking	CVE-2023-24820
C_{49}	CC2538	gnrc_networking	CVE-2023-24821
C_{50}	CC2538	gnrc_networking	CVE-2023-24822
C_{51}	CC2538	gnrc_networking	CVE-2023-24823
C_{52}	CC2538	gnrc_networking	CVE-2023-24825
C_{53}	CC2538	gnrc_networking	CVE-2023-24826
C_{54}	nRF52832	ccn-lite-relay	-
C_{55}	nRF52832	ccn-lite-relay	-
C_{56}	nRF52832	ccn-lite-relay	-
C_{57}	nRF52832	ccn-lite-relay	-
C_{58}	nRF52832	ccn-lite-relay	-
C_{59}	STM32F103	stm32_sine	-

top k threshold may reject valid candidates ranked slightly lower, whereas a large top k threshold increases the analyst's inspection burden. We therefore treat the two values as quantities to be calibrated and conduct a sensitivity analysis. Specifically, we randomly select 30% (18 out of 59) test cases from the dataset. For each test case, we denote by G_n the set containing the first n labeled ground-truth instructions, truncating those beyond the n-th when a test case has more than n labeled instructions. A crash is counted as successfully localized if any instruction in G_n appears within the top k ranked suspicious instructions.

The result of the sensitivity analysis is shown in Fig. 7. The fault localization success rate increases as both n and top k grow. In particular, the setting $n = 5$ along with the top 10 reaches full coverage. We therefore use $n = 5$ and the top 10 as the primary setting that jointly preserves semantic coverage and bounded analyst effort. Furthermore, 88.1% of our fully annotated test cases (52 out of 59) contain no more than five ground-truth instructions, indicating that $n = 5$ covers the dominant portion of the dataset without truncating their ground-truth sets.

B. *Effectiveness of FIRMLOCATOR (RQ1)*

Root cause analysis ability: We provide each analyzer with the crashing test case and the corresponding firmware to analyze and set a timeout of 48 hours. If the faulty instruction is reported in the the top 10 positions in the fault localization result, we consider it a success [41]. Table II shows the root cause analysis ability of FIRMLOCATOR (F) compared to POMP (P) and POMP++ (P^+). ΔRoot represents the number of instructions executed between the root cause and the crash site, along with its proportion in the complete execution trace of the crashing test case. We first use the full execution trace of the crashing test case to analyze the root cause. The results show that FIRMLOCATOR is able to successfully identify the root cause in 58 out of 59 test cases, achieving a success rate of 98.3%. We underline the failed results (i.e., C_{32}) in the full execution trace setting. POMP and POMP++ only identify 3 and 21 out of all test cases, with success rates of 5.1% and 35.6%, respectively. POMP fails in the majority of test cases due to the huge time demands that exceed the timeout. POMP++ fails because of the inadequacy of its heuristic approaches to resolving memory aliasing, which proves unsuitable for the analysis of embedded firmware. In the full execution trace setting, FIRMLOCATOR is able to complete analysis on all test cases without any timeouts. In contrast, POMP and POMP++ only finish 13.6% and 79.7% of test cases, respectively. To enable a fair comparison, the execution trace under analysis is then limited to the last 50% instructions before the crash site. In this limited setting, FIRMLOCATOR still successfully identifies 50 test cases with a success rate of 84.7%, outperforming POMP (10.2% success rate) and POMP++ (54.2% success rate).

FIRMLOCATOR *failure analysis:* While FIRMLOCATOR is able to successfully identify the root causes in most test cases, it fails on C_{32} test cases. The main reason for the failure of C_{32} is that the underlying assumption of FIRMLOCATOR's approach is not met. In this case, an already corrupted data section leads to a crashed interrupt context restore during an interrupt context switch process. Our event-based footprint collection method fails to collect the execution trace of the interrupt context restore process. Thus, FIRMLOCATOR fails to identify the taint sink and the root cause. However, this is an isolated incident and does not impact the overall performance of FIRMLOCATOR. We discuss possible mitigations in Section VIII-B.

Deep root cause analysis: The root cause of a crash may be far away from the crash site, which we call a deep root cause. For example, there are 98.7% of all instructions executed before the crash in C_{16}. A deeper root cause introduces a longer execution trace between the root cause and the crash site, resulting in more

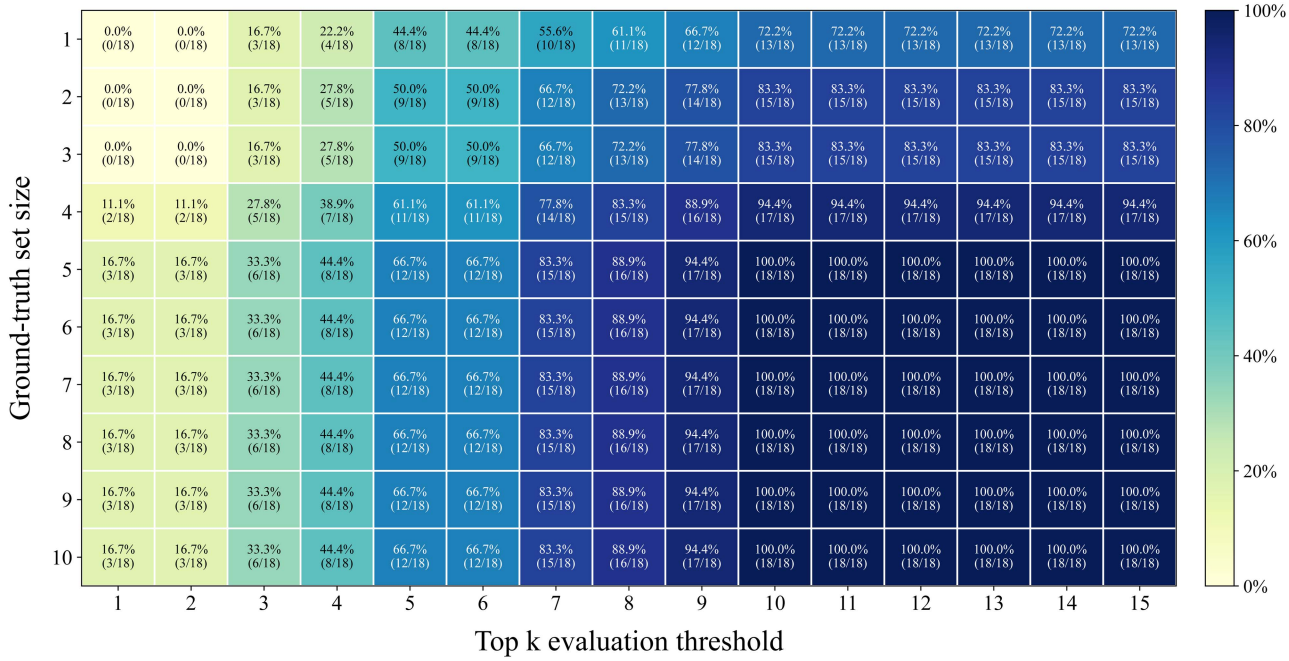

Fig. 7. The fault localization success rate under different settings of top k evaluation threshold and ground-truth set size.

unresolved memory aliases that complicate the analysis. Unresolved memory aliases often lead to considerable analysis time costs and can even result in incorrect fixes, posing challenges for both automated analysis tools and manual investigation. Table II shows the effectiveness of deep root cause analysis. Taking the ΔRoot over 50% as deep root causes, whose crash IDs are marked in gray, FIRMLOCATOR succeeds in 11 out of 11 test cases within the top 10 instructions. Take C_{39} for example, a buffer overflow in the `memcpy` function results in memory corruption. The third argument of `memcpy`, which specifies the number of bytes to copy, is set to a large value, leading to an extremely long execution trace and a deep root cause. FIRMLOCATOR is able to successfully identify the root cause of the C_{39} crash, with a root cause analysis depth of 42,177 (53.6%) instructions.

RQ1: FIRMLOCATOR identifies the root cause of 58 out of 59 test cases with a successful rate of 98.3%, outperforming POMP (5.1% success rate) and POMP++ (35.6% success rate), demonstrating its capability to analyze the full execution trace. FIRMLOCATOR is able to localize deep faults and succeeds in 10 out of 11 crashing test cases whose faults are located more than 50% away from the crash site along the full execution trace.

C. Efficiency of FIRMLOCATOR (RQ2)

This section evaluates the efficiency of FIRMLOCATOR by measuring the overhead of each core component and analyzing the system's overall performance.

Footprint collection overhead: To assess the overhead introduced by the event-based footprint collection, we repeatedly reproduce each crashing test case 50 times with and without event logging enabled. Table III presents the time and space overhead of all test cases.

Our event-driven method incurs an average additional time of 10.68 s (0.73× in addition) and an average log file size increase of 23.58 MB. It is important to emphasize that this footprint collection step is performed separately from the fuzzing process, ensuring that the efficiency of fuzzing campaigns remains unaffected. *Time overhead breakdown:* To understand the computational cost distribution across FIRMLOCATOR's components, we measure the time spent in each phase for full-trace fault localization under a 48-hour timeout. As shown in Fig. 8, the event-based collection phase, although introducing slight overhead, captures precise runtime information crucial for reverse execution. Compared to prior work [9], [13], [15], this trade-off proves to be more advantageous. Notably, reverse execution remains the most time-consuming phase, warranting further optimization. We analyze its subcomponents in greater detail in Section VI-E. Finally, the root cause analysis phase contributes minimal overhead, averaging only 7.22 seconds, with time variations mainly dependent on the number of tainted instructions involved.

Overall time costs: As discussed in Section VI-B, the length of the execution trace under analysis has a significant impact on the efficiency of the root cause analysis. In this work, we define the analysis depth as the number of instructions analyzed leading up to the crash site. A larger value of analysis depth can lead to more unresolved memory aliases, resulting in an increase in the number of use nodes and define nodes whose values are unknown in the constructed use-define chain. These unresolved nodes not only induce extra overhead when updating the use-define chain, but they also limit the ability to perform deep root cause analysis. To empirically quantify this effect, we select four crashing test cases with execution traces exceeding

Fig. 8. Time overhead breakdown of FIRMLOCATOR in a log scale.

TABLE II
COMPARISON OF P, P$^+$, AND FIRMLOCATOR (F) ON FULL AND HALF (LAST 50%) TRACES. ∅, ✗, AND NUMBERS DENOTE TIMEOUT, UNIDENTIFIED ROOT CAUSE, AND ITS RANK, RESPECTIVELY. UNDERLINED RESULTS INDICATE FIRMLOCATOR'S FAILURES IN FULL SETTING. GRAY IDS SIGNIFY ΔROOT (DISTANCE FROM CRASH SITE) > 50%. # TRACES AND # INS REPRESENT TOTAL INSTRUCTIONS IN TRACE AND FIRMWARE.

ID	ΔRoot(%)	# Traces	# Ins	Full			Half		
				P	P$^+$	F	P	P$^+$	F
C_1	2 (<0.1%)	52,080	11,869	∅	✗	1	✗	✗	1
C_2	2 (<0.1%)	371,762	13,374	∅	∅	1	∅	∅	1
C_3	2 (<0.1%)	1,003,421	13,374	∅	∅	1	∅	∅	1
C_4	2 (<0.1%)	228,012	5,021	∅	✗	1	∅	✗	1
C_5	2 (<0.1%)	36,271	11,880	✗	✗	1	✗	✗	1
C_6	2 (<0.1%)	108,294	7,126	∅	✗	1	∅	✗	1
C_7	2 (<0.1%)	619,937	15,140	∅	∅	1	∅	∅	1
C_8	2 (<0.1%)	68,894	15,513	∅	✗	1	✗	✗	1
C_9	2 (<0.1%)	524,897	11,931	∅	∅	1	∅	∅	1
C_{10}	20,203 (18.1%)	111,898	18,536	∅	✗	7	✗	✗	7
C_{11}	78 (<0.1%)	168,542	17,165	∅	1	1	1	1	1
C_{12}	120,472 (51.1%)	235,662	9,299	∅	✗	5	✗	✗	5
C_{13}	2,176 (1.4%)	160,721	9,299	∅	∅	5	∅	✗	5
C_{14}	8,155 (39.5%)	20,630	15,428	✗	✗	4	✗	✗	4
C_{15}	8,719 (73.0%)	11,943	17,165	✗	✗	7	✗	✗	✗
C_{16}	215,102 (98.7%)	217,900	17,165	∅	✗	10	∅	✗	✗
C_{17}	5,183 (3.0%)	173,913	17,165	∅	8	1	✗	8	1
C_{18}	2,759 (1.6%)	169,666	17,165	∅	✗	3	✗	✗	3
C_{19}	5,161 (72.2%)	7,151	9,299	✗	✗	8	✗	✗	✗
C_{20}	306 (<0.1%)	523,873	14,325	∅	✗	9	✗	✗	9
C_{21}	2,013 (1.3%)	150,995	19,706	∅	✗	1	✗	✗	1
C_{22}	401 (0.2%)	238,470	13,954	∅	✗	1	∅	✗	1
C_{23}	53 (<0.1%)	89,136	32,546	∅	✗	1	∅	✗	1
C_{24}	53 (0.1%)	52,909	32,546	∅	✗	1	✗	✗	1
C_{25}	1,789 (0.4%)	434,172	13,954	∅	1	1	∅	1	1
C_{26}	88,746 (45.2%)	196,313	29,993	∅	3	3	3	3	3
C_{27}	27,411 (58.2%)	47,065	15,529	3	3	3	3	3	3
C_{28}	28,771 (58.9%)	48,880	15,529	2	2	2	2	2	2
C_{29}	55,768 (66.4%)	84,028	23,706	∅	✗	5	✗	✗	✗
C_{30}	59,717 (63.3%)	94,362	23,706	∅	✗	5	✗	✗	✗
C_{31}	402 (<0.1%)	514,772	14,325	✗	✗	7	✗	✗	7
C_{32}	86,956 (8.7%)	994,960	27,664	∅	∅	✗	∅	∅	✗
C_{33}	100,512 (48.2%)	208,669	18,464	∅	2	2	∅	2	2
C_{34}	21 (<0.1%)	160,958	18,475	∅	✗	1	∅	✗	1
C_{35}	337 (<0.1%)	1,960,419	26,165	∅	∅	3	∅	∅	3
C_{36}	274,858 (27.1%)	1,016,078	26,167	∅	∅	1	∅	∅	✗
C_{37}	1,458,974 (99.0%)	1,474,404	18,202	∅	∅	2	∅	∅	✗
C_{38}	44,256 (3.4%)	1,300,470	25,887	∅	∅	1	∅	∅	1
C_{39}	42,177 (53.6%)	78,624	15,605	∅	✗	1	✗	✗	1
C_{40}	891 (1.7%)	52,173	11,952	4	4	4	4	4	4
C_{41}	594,471 (95.1%)	625,021	12,258	∅	∅	1	∅	✗	✗
C_{42}	17 (<0.1%)	83,410	24,099	∅	∅	1	1	1	1
C_{43}	3 (<0.1%)	262,836	15,359	∅	1	1	∅	1	1
C_{44}	3 (<0.1%)	537,680	15,690	∅	1	1	∅	1	1
C_{45}	148,956 (27.5%)	542,044	44,640	∅	1	1	∅	1	1
C_{46}	133,883 (25.6%)	544,241	44,628	∅	1	1	∅	1	1
C_{47}	135,650 (25.8%)	525,943	41,731	∅	1	1	∅	1	1
C_{48}	135,074 (25.6%)	526,819	44,637	∅	1	1	∅	1	1
C_{49}	136,382 (25.8%)	527,638	44,640	∅	1	1	∅	1	1
C_{50}	136,024 (25.7%)	528,925	44,645	∅	1	1	∅	1	1
C_{51}	136,314 (25.9%)	526,356	44,652	∅	1	1	∅	1	1
C_{52}	133,636 (25.4%)	525,694	44,636	∅	1	1	∅	1	1
C_{53}	138,182 (26.1%)	529,816	41,745	∅	1	1	∅	1	1
C_{54}	174,437 (35.5%)	490,757	45,298	∅	✗	1	∅	✗	1
C_{55}	152,228 (34.1%)	446,087	45,298	∅	✗	1	∅	✗	1
C_{56}	179,406 (35.9%)	500,135	45,298	∅	✗	1	∅	✗	1
C_{57}	169,896 (35.3%)	481,481	45,298	∅	✗	1	∅	✗	1
C_{58}	157,623 (34.5%)	457,138	45,298	∅	1	1	∅	1	1
C_{59}	11 (<0.1%)	12,930	14,233	∅	1	1	∅	1	1

360,000 instructions and measure the average total analysis time (with a 24-hour timeout). Fig. 9 shows the result of the test cases (C_2, C_{35}, C_{36}, and C_{37}). As the analysis depth increases linearly, POMP demonstrates exponential growth in execution time, and POMP++ shows polynomial-level growth. In contrast, FIRMLOCATOR maintains a relatively modest time increase due to its event-based footprint collection, which alleviates memory aliasing overhead. This difference in performance highlights a key advantage of FIRMLOCATOR's event-based footprint collection that helps resolve memory aliases. FIRMLOCATOR minimizes the impact of unresolved memory aliases, and thus is able to perform a much more efficient fault localization, even for cases with extremely long execution traces. This capability is particularly valuable in complex firmware crash scenarios, where the root cause can be deeply hidden in the execution trace.

> *RQ2:* FIRMLOCATOR's event-based footprint collection introduces an average additional time of 10.68 s (0.73× in addition) and log file size of 23.58 MB. More importantly, it significantly improves the overall efficiency of fault localization, allowing FIRMLOCATOR to outperform both POMP and POMP++ in time scalability and root cause identification depth.

D. Efficacy of Ranking Strategies (RQ3)

To evaluate the impact of our heuristic ranking strategies, we construct four variants of the FIRMLOCATOR framework: (1) FIRMLOCATOR_NR, a baseline version without any ranking strategies; (2) FIRMLOCATOR_RL, which enables only the redundant loop taint suppression strategy; (3) FIRMLOCATOR_HW, which enables only the history write taint prioritization strategy; (4) FIRMLOCATOR, the full version that integrates both heuristic strategies. For each of the 59 crashing test cases in our evaluation dataset, we set a 48-hour analysis timeout in the full execution trace setting. As shown in Fig. 10, all the prototypes successfully identify the root causes of 58 out of 59 test cases within the top 10 ranked instructions, achieving a success rate of 98.3%. *Effectiveness of redundant loop taint suppression:* Compared to FIRMLOCATOR_NR, FIRMLOCATOR_RL significantly improves the ranking performance in cases such as C_{36} and C_{41},

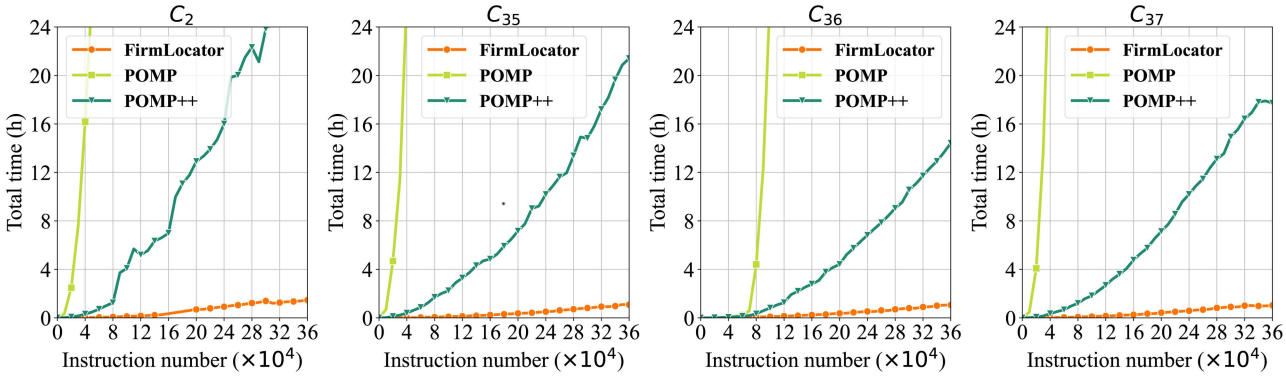

Fig. 9. Overall time costs of fault localization on the different number of instructions under analysis (C_2, C_{35}, C_{36}, and C_{37}).

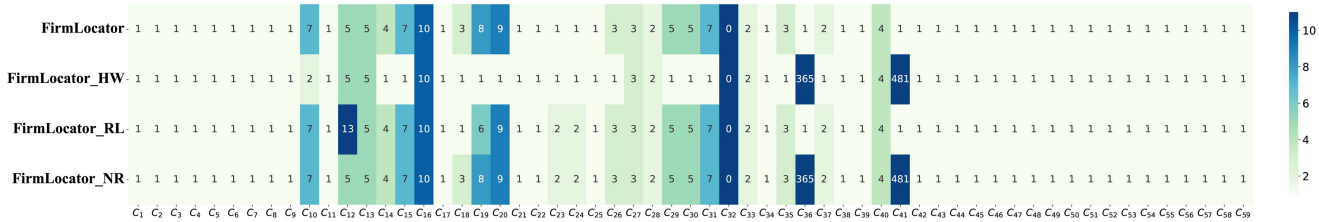

Fig. 10. Comparison results of root cause identification of FIRMLOCATOR_NR, FIRMLOCATOR_RL, FIRMLOCATOR_HW, and FIRMLOCATOR in the heatmap visualization, where lighter shades indicate better results. The number in the cell represents the rank of the root cause instruction in the result.

where the root cause is successfully identified at the the top 1 position. For instance, in C_{41}, FIRMLOCATOR_NR assigns high suspicious scores to hundreds of repetitive `ADDS R0, #0x40` instructions—simple value increments inside a loop that do not contribute to the crash root cause. The loop suppression strategy effectively lowers their scores and promotes the actual root cause instruction in the final ranking, reducing analyst effort.

Effectiveness of history write taint prioritization: FIRMLOCATOR_HW yields even greater improvements over the baseline in 15 (25.4%) cases, including C_{10}, C_{14-15}, C_{17-20}, C_{23-24}, C_{26}, C_{29-31}, C_{35}, and C_{37}. This strategy prioritizes earlier memory write instructions that are semantically significant for the program state, thereby improving root cause visibility. Notably, FIRMLOCATOR_HW achieves the best overall the top 1 performance—successfully ranking the root cause as the highest-ranked instruction in 81.4% of the cases, outperforming FIRMLOCATOR_RL (64.4%), FIRMLOCATOR_NR (59.3%), and even the full version FIRMLOCATOR (66.1%).

Overall strategy synergy and trade-offs: FIRMLOCATOR adopts a dual-heuristic strategy to balance the benefits of both approaches. The redundant loop taint suppression strategy effectively reduces noise from repeated instruction patterns, as demonstrated in cases such as C_{36} and C_{41}. Meanwhile, the history write taint prioritization strategy boosts the ranking of deep root cause instructions by assigning higher scores to semantically critical instructions earlier in the trace, preserving improvements previously observed in 15 challenging cases.

While some test cases (e.g., C_{14}, C_{15}) experience minor degradations in their root cause rank due to conflicts between strategies, the overall effect is beneficial. The combined approach achieves robust performance across diverse crash types: with the exception of a single known failure case, all remaining successful cases rank the root cause within the top 10. This demonstrates that although individual heuristics may excel in specific cases, their integration achieves broader applicability and improved overall stability.

RQ3: All FIRMLOCATOR variants accurately rank root causes within the the top 10 for 58/59 cases. Specifically, redundant loop suppression refines precision in repetitive patterns, while history write prioritization enhances 25.4% of cases. The integrated dual-heuristic strategy effectively balances precision with generality.

E. Event-Based Method Contribution (RQ4)

To evaluate the contribution of FIRMLOCATOR's event-based method, we break down the time overhead of the reverse execution process into two components: (1) reverse, which refers to use-definition chain construction along the execution trace, and (2) resolve, which refers to memory alias resolution and data recovery.

For consistency and fair comparison, we divide POMP and POMP++ into the same two stages. Specifically: The reverse phase builds the dependency chain by traversing instructions backward from the crash site. The resolve phase attempts to recover memory state by resolving pointer aliases and interpreting historical memory accesses.

TABLE III
TIME (T) AND SPACE (S) OVERHEAD FOR CRASH REPRODUCTION. SUBSCRIPTS o, D, A, w REPRESENT NO EVENTS, DATA EVENTS, ACTION EVENTS, AND BOTH ENABLED, RESPECTIVELY. S_o IS OMITTED AS NO FILES ARE GENERATED WHEN EVENTS ARE DISABLED.

ID	T_o (s)	T_d (s)	T_a (s)	T_w (s)	S_d (MB)	S_a (MB)	S_w (MB)
C_1	0.77	1.18	1.56	1.98	1.60	1.99	3.58
C_2	0.74	2.69	4.05	5.72	11.74	14.18	25.92
C_3	0.74	5.82	8.83	13.53	31.97	38.28	70.25
C_4	0.79	1.95	2.85	3.74	6.89	8.70	15.59
C_5	0.71	1.34	1.62	1.77	1.08	1.38	2.47
C_6	0.84	1.52	2.03	2.31	3.27	4.13	7.40
C_7	0.69	4.22	6.20	9.16	19.73	23.65	43.38
C_8	0.77	1.35	1.64	2.17	2.00	2.63	4.63
C_9	0.62	3.62	5.30	7.88	16.88	20.02	36.90
C_{10}	0.80	1.65	2.13	2.42	2.87	4.27	7.09
C_{11}	0.92	1.42	2.55	2.88	2.51	6.43	8.94
C_{12}	0.66	1.48	2.99	3.56	2.64	8.99	11.63
C_{13}	0.78	1.94	2.43	3.18	5.39	6.13	11.52
C_{14}	0.92	1.57	1.59	1.72	0.53	0.79	1.31
C_{15}	0.63	1.07	1.29	1.05	0.26	0.46	0.71
C_{16}	0.69	1.82	2.96	3.33	3.35	8.31	11.51
C_{17}	0.87	1.58	2.81	3.10	2.56	6.63	9.19
C_{18}	0.91	1.41	2.45	2.99	2.48	6.47	8.98
C_{19}	0.73	1.05	1.49	1.27	0.17	0.27	0.44
C_{20}	0.89	2.06	5.48	6.40	5.77	19.98	25.76
C_{21}	0.76	1.68	2.33	2.78	3.64	5.76	9.40
C_{22}	0.80	2.08	3.00	3.73	6.30	9.10	15.39
C_{23}	1.33	1.85	2.20	2.60	2.15	3.40	5.55
C_{24}	0.84	1.51	1.78	1.78	1.38	2.02	3.32
C_{25}	0.57	2.65	4.30	5.94	9.85	16.56	26.41
C_{26}	0.65	1.47	2.72	2.96	2.27	7.49	9.75
C_{27}	0.76	1.40	1.40	1.72	1.01	1.80	2.80
C_{28}	0.95	1.25	1.60	1.66	1.07	1.86	2.94
C_{29}	0.73	1.69	2.13	2.47	2.19	3.21	5.40
C_{30}	0.79	1.26	1.89	2.36	0.88	3.60	6.07
C_{31}	0.74	1.91	5.36	5.94	5.61	19.64	25.24
C_{32}	0.98	5.90	9.24	13.52	29.23	37.95	67.11
C_{33}	1.01	2.15	2.98	3.81	5.02	7.96	12.98
C_{34}	0.82	2.04	2.93	3.28	4.63	6.14	10.77
C_{35}	0.98	9.63	16.47	24.88	57.54	74.78	132.28
C_{36}	1.02	5.16	9.31	13.41	27.40	38.76	66.14
C_{37}	1.18	6.61	12.84	18.03	34.81	56.24	91.06
C_{38}	0.81	7.14	11.30	17.22	38.54	49.61	88.11
C_{39}	0.74	1.80	2.25	2.41	2.67	3.00	5.68
C_{40}	0.80	1.61	1.84	2.21	1.22	1.99	3.21
C_{41}	0.93	6.26	6.92	11.32	28.70	23.84	52.56
C_{42}	19.54	22.21	27.03	29.88	7.38	20.68	28.06
C_{43}	18.71	21.28	26.23	28.23	7.06	20.00	27.06
C_{44}	19.31	22.06	26.23	28.26	7.09	20.06	27.15
C_{45}	18.81	21.70	25.83	28.30	7.11	20.10	27.20
C_{46}	18.49	22.00	26.40	28.85	7.12	20.13	27.25
C_{47}	18.50	22.08	26.68	28.23	7.14	20.18	27.32
C_{48}	18.86	21.43	25.99	27.83	7.10	20.08	27.18
C_{49}	18.64	21.57	26.99	28.35	7.09	20.05	27.14
C_{50}	18.63	21.66	27.02	28.28	7.16	20.21	27.37
C_{51}	18.78	22.06	26.79	28.60	10.63	20.05	30.68
C_{52}	19.28	23.41	26.69	29.13	10.87	20.51	31.38
C_{53}	4.51	5.54	6.43	6.81	1.72	4.22	5.94
C_{54}	25.13	27.81	33.06	35.72	7.67	18.72	26.40
C_{55}	22.23	24.61	30.20	31.86	6.84	17.01	23.86
C_{56}	24.66	28.51	32.24	36.76	7.88	19.07	26.96
C_{57}	23.60	25.30	34.82	33.95	7.41	18.36	25.78
C_{58}	21.21	24.89	32.02	31.11	6.95	17.43	24.39
C_{59}	1.34	2.03	2.66	2.44	0.26	0.49	0.75
Avg.	6.17	8.27	10.68	12.15	8.72	14.85	23.58

We conduct this experiment at a fixed analysis depth of the last 10,000 instructions in the execution trace, ensuring that all test cases can complete within 24 hours. Each test case is run five times, and average timings are reported. The results are shown in Fig. 11. On average: POMP spends 7.30 s in reverse and 766.59 s in resolve. POMP++ improves slightly with 11.66 s and 8.08 s, respectively. FIRMLOCATOR maintains comparable performance in the reverse phase, but significantly outperforms both baselines in the resolve phase, achieving an average resolve time of just <0.01 s. This is attributed to FIRMLOCATOR's event-driven footprint collection, which records critical memory accesses during crash reproduction. This strategy enables highly accurate and efficient memory alias resolution without incurring the cost of speculative symbolic reasoning or value set analysis.

RQ4: FIRMLOCATOR's event-based data recovery design dramatically improves the efficiency of memory alias resolution during reverse execution, resulting in superior overall fault localization performance compared to state-of-the-art baselines.

VII. DISCUSSION

Based on the evaluation results, we provide the following insights into postmortem-based fault localization methods in embedded firmware fuzzing.

Memory alias resolve problem: Recent postmortem-based works hold the view that recording runtime data introduces huge time overheads [9], [14]. Consequently, they infer memory aliases relying on core dumps, the execution trace, and a memory snapshot at the crash site. The difference between inferred data and actual data hinders the performance of reverse execution and backward taint analysis. In contrast, FIRMLOCATOR leverages event-based footprint collection to capture precise memory access behaviors during crash reproduction, thereby resolving memory aliases more accurately and enabling efficient fault localization.

Event recording: In the fault localization for embedded firmware fuzzing scenario, recording the execution trace and memory access history greatly improves efficiency and effectiveness with little overhead. For one thing, the event-based footprint collection method compensates for inadequate debugging mechanisms. For another, we trigger data events and action events only when reproducing crashing test cases, which is an independent period after fuzzing and does not negatively impact the fuzzing process.

Hardware tracing: On real hardware devices, analysts can also use a tracing unit to collect the runtime information. For example, with the support of the Embedded Trace Macrocell (ETM [42]) feature, analysts are able to obtain the execution trace [43]. However, these features require hardware settings to be enabled and do not exhibit priorities against our event-based footprint collection in terms of quantity and quality. In this work, FIRMLOCATOR utilizes the event-based footprint collection also for the sake of scalability and financial cost.

Execution trace length: Given a new crashing test case, analysts initially lack knowledge of the precise location where the fault resides. Intuitively, to accurately identify the root cause that could be far from the crash site, a longer trace would be preferable. However, as the number of instructions to analyze grows, the cost of the analysis time increases rapidly. When dealing with execution traces of tens of thousands of instructions, the total analysis period could even stretch to several days. Therefore, it is important for postmortem-based fault localization methods to determine the appropriate length of the execution

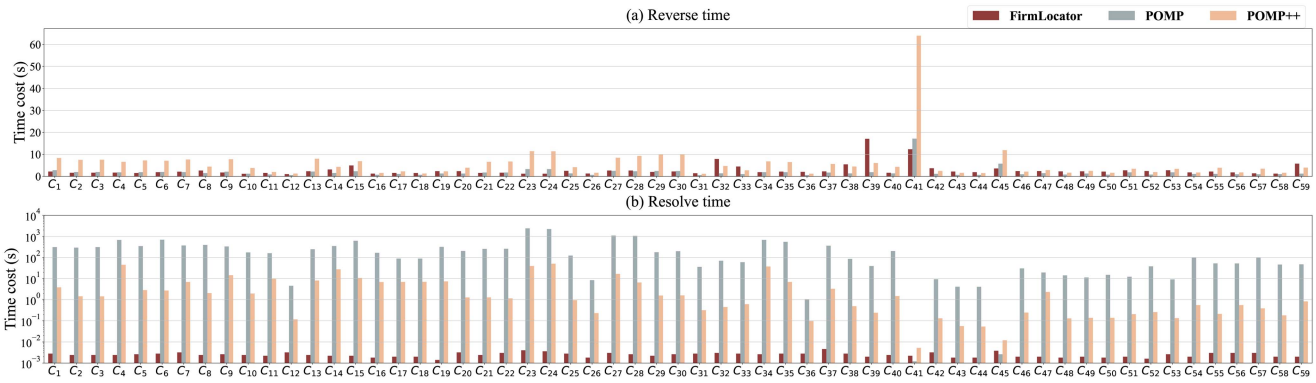

Fig. 11. The time overhead breakdown of the reverse execution on the last 10,000 instructions in the execution trace among FIRMLOCATOR, POMP and POMP++. (a) Reverse time shows use-define chain construction time cost. (b) Resolve time denotes the time consumed for memory alias resolution, shown in logarithmic scale.

trace to analyze. FIRMLOCATOR is able to complete analysis under the full execution trace setting in a reasonable time budget, demonstrating its efficiency and capability.

VIII. LIMITATIONS AND FUTURE WORK

While FIRMLOCATOR has demonstrated great performance in identifying the root causes of firmware crashes, there are some limitations that have not been fully addressed. In this section, we discuss the limitations of FIRMLOCATOR and possible avenues for future work.

A. Limitations

Fuzzing constraints: FIRMLOCATOR operates as a post-fuzzing analysis framework and is inherently dependent on the capabilities of the underlying fuzzer. As existing firmware fuzzers cannot perfectly emulate all embedded platforms and peripheral behaviors, the applicability of FIRMLOCATOR is constrained by the limitations of fuzzing coverage and platform support. This restricts the range and diversity of firmware binaries that can be effectively analyzed.

Report usability: To ensure scalability, FIRMLOCATOR targets raw firmware binaries stripped of debug symbols. FIRMLOCATOR reports the ranked instructions to analysts, but does not mean to get rid of all the manual investigation in the root cause analysis work. However, FIRMLOCATOR highlights key instructions related to the crash from tens of thousands of instructions in the execution trace, providing a crucial starting point for further manual analysis.

Generality limitations: FIRMLOCATOR currently adopts heuristic strategies and partial instruction support for prioritizing suspicious instructions during root cause analysis. The two proposed ranking strategies—loop-taint suppression and history-write prioritization—are designed based on common patterns observed in our evaluation dataset and may not generalize to all firmware scenarios. Similarly, FIRMLOCATOR supports a subset of 43 ARM Cortex-M instructions that are frequently used and have a significant influence on data and control flow. Given the vast number of instructions in the ARM Cortex-M ISA, this limited coverage may restrict FIRMLOCATOR's applicability when analyzing binaries that make use of uncommon or complex instructions. Both the heuristic strategies and instruction coverage are practical trade-offs between generality and efficiency, and expanding them to achieve broader applicability remains a direction for future improvement.

B. Future Work

While FIRMLOCATOR relies on an off-the-shelf fuzzer to collect the necessary runtime information, the data requirements could potentially be satisfied through other forms of dynamic program analysis as well. Therefore, our work is able to enhance other dynamic root cause analysis methods.

We acknowledge that FIRMLOCATOR encounters some analysis failures, which we believe could be mitigated by sophisticated ranking strategies and runtime information retrieval methods. However, since they are incremental improvements upon FIRMLOCATOR, we leave them as future work.

Besides, FIRMLOCATOR currently analyzes code semantics at the granularity of instructions. This leads to a limited understanding of the intentions of the whole program. A promising future direction would be integrating FIRMLOCATOR with advanced techniques such as large language models (LLMs) to generate more comprehensive, natural language descriptions of the identified root causes.

IX. RELATED WORK

A. Embedded Firmware Fuzzing

The embedded firmware runs on a variety of hardware platforms and interacts with multiple complicated peripherals. Some early works conduct black-box fuzzing directly on real hardware devices [44], [45], [46], but this approach lacks comprehensive coverage guidance. μAFL [6] and SyzTrust [47] utilized a hardware debugger to collect runtime information with the ETM feature. Other works adopt a hardware-in-the-loop [7], [8], [48], [49] approach to forward hardware requests to real peripherals. However, these methods require frequent context switching and state synchronization between the emulator and hardware, leading to significant performance overhead. Recent works adopt the rehosting technique for better scalability [5],

[32], [33], [34], [50], [51], [52], [53]. The key idea of rehosting is modeling peripherals and providing firmware with valid inputs, which mainly helps firmware pass state checks during the initialization stage. These rehosting works mainly focus on enabling fuzzing on IoT firmware, rather than tailoring and optimizing backend fuzzers (e.g., AFL [54]) for enhancing the fuzzing process effectiveness.

B. Automated Fault Localization

The manual investigation towards software crash analysis is a time-consuming and tedious process, which has motivated the development of automated fault localization techniques.

Spectrum-based approaches analyze crashing and non-crashing test cases to rank suspicious instructions based on their occurrence frequencies [21], [55], [56], [57], [58], [59]. While effective, these methods are limited in their analysis granularity. More advanced techniques explore different execution paths near the initial crash and assign scores to program entities [11], [12], [13], [60], [61], [62]. However, the heavy time cost of analyzing each individual test case makes them impractical for large-scale firmware fuzzing. Postmortem-based methods leverage post-crash artifacts, such as execution traces and memory dumps, to reverse analyze the program and identify the minimum set of instructions potentially responsible for the crash [9], [14], [15], [16], [37], [63], [64]. CrashLocator [64] located faulty functions using the crash stack information in crash reports. RETracer [16] reversely analyzed the code and the stack to find out how a bad value propagates. Besides, learning-based methods are also widely studied for a better understanding of crashes [17], [18], [19], [65], [66], [67], [68]. They proposed neural network architectures to analyze and incorporate the program context into suspicious score calculation. Recently, with the rise of LLMs, program analysis has achieved vigorous advances [20], [69].

X. Conclusion

In this paper, we present FirmLocator, a practical and efficient postmortem fault localization framework tailored for ARM embedded firmware. By introducing an event-based footprint collection mechanism, FirmLocator precisely resolves memory aliasing issues that traditionally hinder reverse execution and taint analysis in raw firmware analysis. We further propose a novel two-phase ranking strategy that prioritizes root-cause-relevant instructions, significantly reducing the manual burden on analysts.

FirmLocator represents a crucial step toward bridging the gap between fuzzing and vulnerability remediation—the "last mile" in the firmware vulnerability discovery pipeline. We evaluate FirmLocator on 59 crashing test cases across 19 firmware images, achieving a root cause identification success rate of 98.3% within the top 10 instructions. Compared to state-of-the-art works, FirmLocator demonstrates its superiority in 20.3% improvement in full execution trace analysis capability, polynomial-level acceleration in overall efficiency and 62.7% higher success rate within the top 10 instructions in effectiveness.

References

[1] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Comput. Surv.*, vol. 54, no. 1, pp. 1–36, 2021.

[2] HP, "HP wolf security threat insights report," 2023, Accessed: Jun. 2024. [Online]. Available: https://threatresearch.ext.hp.com/wp-content/uploads/2024/02/HP_Wolf_Security_Threat_Insights_Report_Q4_2023.pdf

[3] IBM, "Cost of a data breach report," 2023, Accessed: Jun. 2024. [Online]. Available: https://www.ibm.com/reports/data-breach

[4] Viasat, "KA-SAT network cyber attack overview," 2022, Accessed: Jun. 2024. [Online]. Available: https://news.viasat.com/blog/corporate/ka-sat-network-cyber-attack-overview

[5] T. Scharnowski et al., "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 1239–1256.

[6] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "μafl: Non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1–12.

[7] J. Zaddach et al., "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–16.

[8] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.*, 2018, pp. 1–11.

[9] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem program analysis with hardware-enhanced post-crash artifacts," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 17–32.

[10] D. Mu et al., "An in-depth analysis of duplicated Linux kernel bug reports," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2022, pp. 1–18.

[11] T. Blazytko et al., "AURORA: Statistical crash analysis for automated root cause explanation," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 235–252.

[12] Y. Park, H. Lee, J. Jung, H. Koo, and H. K. Kim, "Benzene: A practical root cause analysis system with an under-constrained state mutation," in *Proc. IEEE Symp. Secur. Privacy*, 2024, pp. 74–74.

[13] D. Xu et al., "Racing on the negative force: Efficient vulnerability root-cause analysis through reinforcement learning on counterexamples," in *Proc. 33rd USENIX Secur. Symp.*, 2024, pp. 4229–4246.

[14] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1787–1804.

[15] W. Cui et al., "REPT: Reverse debugging of failures in deployed software," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 17–32.

[16] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, "RETracer: Triaging crashes by reverse execution from partial memory dumps," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 820–831.

[17] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," in *Proc. ACM Programm. Lang.*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[18] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 273–283.

[19] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 169–180.

[20] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *Proc. 46th IEEE/ACM Int. Conf. Softw. Eng.*, 2024, pp. 1–12.

[21] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using DStar (D*)," in *Proc. IEEE Sixth Int. Conf. Softw. Secur. Rel.*, 2012, pp. 21–30.

[22] Q. I. Sarhan and Á. Beszédes, "A survey of challenges in spectrum-based software fault localization," *IEEE Access*, vol. 10, pp. 10618–10639, 2022.

[23] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proc. 10th ACM SIGSOFT Symp. Found. Softw. Eng.*, New York, NY, USA, 2002, pp. 1–10, doi: 10.1145/587051.587053.

[24] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proc. Int. Symp. Softw. Testing Anal.*, New York, NY, USA, 2011, pp. 199–209, doi: 10.1145/2001420.2001445.

[25] B. Chang et al., "FirmRCA: Towards post-fuzzing analysis on arm embedded firmware with efficient event-based fault localization," in *Proc. 46th IEEE Symp. Secur. Privacy*, 2025, pp. 3783–3800.

[26] Arm Limited, "ARMv7-M Architecture Reference Manual," ARM Limited E.e ed., Feb. 2027. Accessed: Jun. 2024. [Online]. Available: https://developer.arm.com/documentation/ddi0403/ee/
[27] Hoedur, "Files used for reproducing Hoedur's experiments," 2023, Accessed: Jun. 2025. [Online]. Available: https://github.com/fuzzware-fuzzer/hoedur-experiments/tree/main
[28] GDMA, "Files used for reproducing GDMA's experiments," 2025, Accessed: Jun. 2025. [Online]. Available: https://github.com/fuzzware-fuzzer/gdma-experiments/tree/main
[29] Fuzzware, "Files used for reproducing fuzzware's experiments," 2022, Accessed: Jun. 2024. [Online]. Available: https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/main
[30] MultiFuzz, "Files used for reproducing multifuzz's experiments," 2024, Accessed: Jan. 2026. [Online]. Available: https://github.com/MultiFuzz/MultiFuzz-benchmarks/tree/main
[31] SafireFuzz, "Files used for reproducing safirefuzz's experiments," 2023, Accessed: Jan. 2026. [Online]. Available: https://github.com/pr0me/safirefuzz-experiments/tree/main
[32] A. Fasano et al., "SoK: Enabling security analyses of embedded systems via rehosting," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2021, pp. 687–701.
[33] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz, "Hoedur: Embedded firmware fuzzing using multi-stream inputs," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 2885–2902.
[34] M. Chesser, S. Nepal, and D. C. Ranasinghe, "Icicle: A re-designed emulator for grey-box firmware fuzzing," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 76–88.
[35] U. Engine, "The ultimate CPU emulator," 2015, Accessed: Jun. 2024. [Online]. Available: https://www.unicorn-engine.org/
[36] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers Principles, Techniques & Tools*. London, U.K.: Pearson Education, 2007.
[37] D. Mu et al., "POMP++: Facilitating postmortem program diagnosis with value-set analysis," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1929–1942, Sep. 2021.
[38] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, and H. Yin, "Patchscope: Memory object centric patch diffing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 149–165.
[39] Capstone Engine, "The ultimate disassembler," 2015, Accessed: Jun. 2024. [Online]. Available: https://www.capstone-engine.org/
[40] Intel, "Collecting intel processor trace (intel pt) in intel system debugger," 2018. Accessed: Jun. 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/videos/collecting-processor-trace-in-intel-system-debugger.html?wapkw=intel%20pt
[41] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 165–176.
[42] ARM, "Embedded trace macrocell, etmv1.0 to etmv3.5, architecture specification," 2011. Accessed: Jun. 2024. [Online]. Available: https://documentation-service.arm.com/static/5f90158b4966cd7c95fd5b5e
[43] Y. Zhang et al., "Alligator in vest: A practical failure-diagnosis framework via arm hardware features," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 917–928.
[44] K. Koscher et al., "Experimental security analysis of a modern automobile," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 447–462.
[45] C. Mulliner, N. Golde, and J.-P. Seifert, "SMS of death: From analyzing to attacking mobile phones on a large scale," in *Proc. 20th USENIX Secur. Symp.*, 2011, pp. 24–24.
[46] J. Chen et al., "IOTFUZZER: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.
[47] Q. Wang et al., "Syztrust: State-aware fuzzing on trusted OS designed for IoT devices," in *Proc. IEEE Symp. Secur. Privacy*, 2024, pp. 70–70.
[48] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *Proc. 9th USENIX Workshop Offensive Technol.*, 2015, pp. 1–10.
[49] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 291–307.
[50] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1099–1114.
[51] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1237–1254.
[52] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 2007–2024.
[53] A. A. Clements et al., "HALucinator: Firmware re-hosting through abstraction layer emulation," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1201–1218.
[54] M. Zalewski, "American fuzzy lop," 2010, Accessed: Jun. 2024. [Online]. Available: https://lcamtuf.coredump.cx/afl/
[55] A. d. S. Meyer, A. A. F. Garcia, A. P. d. Souza, and C. L. d. Souza Jr, "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L)," *Genet. Mol. Biol.*, vol. 27, pp. 83–91, 2004.
[56] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2005, pp. 273–282.
[57] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proc. 12th Pacific Rim Int. Symp. Dependable Comput.*, 2006, pp. 39–46.
[58] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. Testing: Acad. Ind. Conf. Pract. Res. Techn.*, 2007, pp. 89–98.
[59] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 1–32, 2011.
[60] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 5–15.
[61] X. Zhang et al., "Default: Mutual information-based crash triage for massive crashes," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 635–646.
[62] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 286–295, 2005.
[63] D. Mu et al., "RENN: Efficient reverse execution with neural-network-assisted alias analysis," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 924–935.
[64] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *Proc. 2014 Int. Symp. Softw. Testing Anal.*, 2014, pp. 204–214.
[65] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 661–673.
[66] Z. Zhang, Y. Lei, X. Mao, M. Yan, X. Xia, and D. Lo, "Context-aware neural fault localization," *IEEE Trans. Softw. Eng.*, vol. 49, no. 7, pp. 3939–3954, Jul. 2023.
[67] Y. Lou et al., "Boosting coverage-based fault localization via graph-based representation learning," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 664–676.
[68] S. Wang et al., "Beep: Fine-grained fix localization by learning to predict buggy code elements," 2021, *arXiv:2111.07739*.
[69] X. Shang et al., "How far have we gone in stripped binary code understanding using large language models," 2024, *arXiv:2404.09836*.

Boyu Chang received the BS degree from the College of Computer Science and Technology, Zhejiang University, in 2022. He is currently working toward the PhD degree with Zhejiang University under the supervision of Prof. Shouling Ji. His research interests include software system security and binary analysis.

Binbin Zhao (Member, IEEE) received the PhD degree in electrical and computer engineering from the Georgia Institute of Technology. He is a ZJU 100 young professor with the College of Computer Science and Technology, Zhejiang University. He was a research faculty member with the School of Electrical and Computer Engineering. His research interests include IoT security, fuzzing, and data-driven security.

Bo Xu is currently working toward the graduate degree with the School of Cyberspace Security, Huazhong University of Science and Technology. His research interests include system fuzzing, software security, and malware detection models.

Qiao Zhang is currently working toward the PhD degree with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. His research interests include system security and AI for security.

Peiyu Liu (Member, IEEE) received the PhD degree from the School of Computer Science and Technology, Zhejiang University, Hangzhou, China. He is currently a postdoctoral fellow with the College of Control Science and Engineering, Zhejiang University. His current research interests include systems security and industrial control system security.

Qinge Xie received the BS degree in computer science from the Zhejiang University of Technology, in 2017, the MS degree in computer science from Fudan University, in 2020, and the PhD degree in computer science from the Georgia Institute of Technology, in 2025. Her research interests include web security and privacy, network security, and Internet measurements. She has published in leading conferences and journals in the field of computer security and networking, such as USENIX Security Symposium and *IEEE Transactions on Mobile Computing*.

Guozhu Meng (Senior Member, IEEE) received the BE and ME degrees from Tianjin University, China, in 2009 and 2012, respectively, and the PhD degree from Nanyang Technological University, Singapore, in 2017. He is currently an associate professor with the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include mobile security, vulnerability detection, Big Data analysis, and program analysis. He is the recipient of ACM SIGSAC China Rising Star and Beijing Nova Program.

Shouling Ji (Member, IEEE) received the BS (with Honors) and MS degrees in computer science from Heilongjiang University, the first PhD degree in electrical and computer engineering from the Georgia Institute of Technology, and the second PhD degree in computer science from Georgia State University. He is a Qiushi distinguished professor with the College of Computer Science and Technology, Zhejiang University and an adjunct research faculty with the School of Electrical and Computer Engineering, Georgia Institute of Technology. His current research interests include data-driven security and privacy, AI security, and software and system security. He is a member of ACM, a senior member of CCF, and was the membership chair of the IEEE Student Branch, Georgia State University (2012-2013).