Journal of Systems Architecture 168 (2025) 103578

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Log2Evt: Constructing high-level events for IoT Systems through log-code execution path correlation☆

Teng Li [a,b,c,d], Baichuan Zheng [a,b,c,d], Yebo Feng [e,h,*], Xiaowen Quan [f], Jiahua Xu [g,h], Yang Liu [a], Jianfeng Ma [a]

[a] School of Cyber Engineering, Xidian University, Xi'an 710071, China
[b] State Key Laboratory of Integrated Services Networks, Xidian University, Xi'an 710071, China
[c] Songshan Laboratory, Henan 450018, China
[d] Key Laboratory of Cyberspace Security, Ministry of Education, Xi'an 710071, China
[e] College of Computing and Data Science, Nanyang Technological University, Singapore 639798, Singapore
[f] WebRAY Tech (Beijing) Co., Ltd., Beijing 100000, China
[g] Department of Computer Science, University College London, London WC1E 6EA, UK
[h] Exponential Science, Cayman Islands

ARTICLE INFO

Keywords:
Internet of things
Smart society
Log analysis
Event converting
Graph theory

ABSTRACT

The detection of cyberattacks in IoT ecosystems requires comprehensive log auditing across distributed devices, yet the volume and heterogeneity of IoT logs exceed traditional analysis capabilities. Therefore, it is essential to narrow down the scope of forensics precisely and efficiently to target attack-related events. Existing schemes have the disadvantage of low accuracy and flexibility. We propose a novel approach that synthesizes high-level security events from low-level IoT logs by correlating firmware execution traces with runtime call stack contexts. Our approach implements lightweight monitoring probes at critical IoT workflow points and employs an IoT-optimized Common Ancestor algorithm for log sequence analysis. The experiments demonstrate a 15% improvement in accuracy compared to the rule-based matching scheme. Additionally, the results highlight the influence of the threshold parameter and show that the approach has minimal impact on program operation. The approach effectively addresses the challenges of protocol fragmentation and resource constraints in IoT environments, providing a foundation for robust security monitoring in smart city deployments.

1. Introduction

In IoT-enabled smart societies, the operational behavior of IoT devices, such as sensor activations, data transmissions, or firmware updates, manifests as high-level events composed of numerous low-level function executions. Each execution generates a stream of log messages that document state changes across distributed IoT networks. Constructing high-level events from raw IoT logs is essential for two reasons. First, these events contextualize device interactions into actionable security narratives, enabling analysts to trace multi-step attack chains, such as compromised sensors triggering cascading failures, which individual logs alone cannot reveal [1]. Second, they expose stealthy attack patterns, including slow-burn data exfiltration or spoofed device commands, that evade detection when logs are analyzed in isolation. However, IoT environments introduce unique challenges. Fig. 1 illustrates a fundamental challenge in log analysis where concurrent system activities cause log entries to become fragmented and interleaved. A single logical event, such as a user login, often comprises multiple suboperations, such as authentication and session initialization, with each generating a distinct log entry. In the figure, the single User-A Login event produces two separate log entries: User-A Login_1 and User-A Login_2. These entries are separated in the timeline by logs from other concurrent user activities. This interleaving complicates the reconstruction of a user's session. For example, the complete timeline for User-A involves a successful login that generates two separate log entries, a subsequent logout, and later, a separate, failed login attempt. Without sophisticated analysis, the fragmented nature of these logs makes it difficult to accurately trace the sequence of operations for any single user or process. Additionally, heterogeneous IoT hardware generates both structured logs, like timestamped sensor readings, and unstructured logs, such as free-text error reports from legacy devices. These concurrent operations scatter log entries, forcing analysts to prioritize specific messages and inadvertently overlook fragmented traces of malicious activity [2,3]. Attackers exploit these inconsistencies by dispersing malicious activity across mixed-format

☆ This article is part of a Special issue entitled: 'SPASS' published in Journal of Systems Architecture.

* Corresponding author at: College of Computing and Data Science, Nanyang Technological University, Singapore 639798, Singapore.
E-mail address: yebo.feng@ntu.edu.sg (Y. Feng).

https://doi.org/10.1016/j.sysarc.2025.103578
Received 31 March 2025; Received in revised form 27 August 2025; Accepted 8 September 2025
Available online 13 September 2025

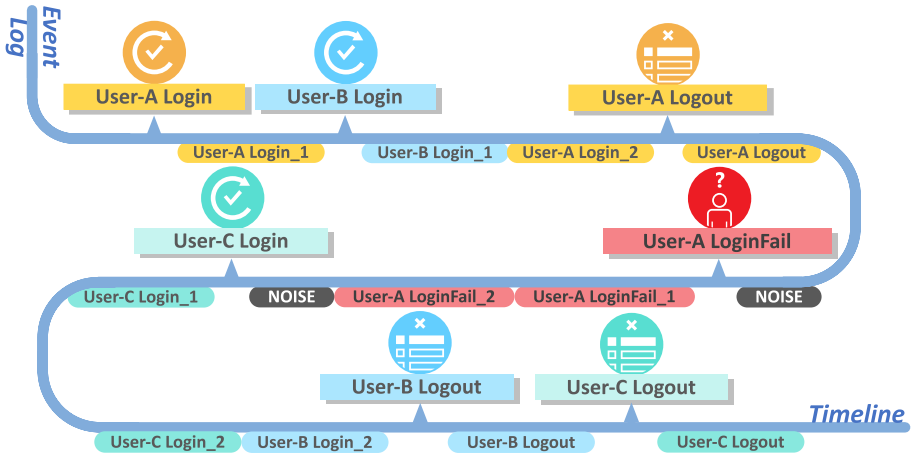


Fig. 1. Visualization of interleaved multi-user log sequences.

logs to evade rule-based detection [4–6]. Existing approaches for IoT log analysis [7–9] often prioritize sequential log ordering, neglecting asynchronous workflows where correlated events span multiple devices. By isolating event-specific log sequences, such as those tied to a single firmware update, auditors can reconstruct IoT-centric attack pathways, such as rogue nodes injecting falsified data. This need for efficiency is a well-recognized challenge, as many security mechanisms for functions like data sharing are too computationally intensive for resource-limited IoT devices [10]. Addressing these challenges requires lightweight techniques to map low-level IoT logs to high-level events without overwhelming resource-constrained devices, ensuring real-time intrusion detection in scalable smart societies.

However, transforming low-level log data generated by IoT devices into high-level operational events remains a critical challenge for effective intrusion detection in smart societies. IoT systems inherently involve large-scale deployments with heterogeneous devices, producing immense volumes of unstructured logs. Administrators struggle to define reliable detection rules [11,12] or keywords [13,14] due to the dynamic nature of IoT environments, where device diversity and unpredictable interactions amplify the risk of missing attack signatures. Compounding this issue, IoT devices often lack standardized logging formats, leading to inconsistencies that hinder automated analysis. Furthermore, concurrent operations across distributed IoT networks scatter log entries, forcing analysts to prioritize specific messages and inadvertently overlook fragmented traces of malicious activity. Systems can analyze IoT workflow execution paths, representing the chronological sequence of operations. By reconstructing these paths, systems can correlate isolated log entries into coherent, high-level events. This enables real-time detection of covert attacks. This focus on efficiency is crucial, as even fundamental tasks like processing network packets can create unpredictable workloads that challenge the real-time capabilities of embedded systems [15]. Importantly, this approach maintains minimal computational overhead, which is essential for resource-constrained IoT infrastructures.

In IoT-enabled environments, addressing log analysis challenges requires approaches capable of processing massive volumes of device-generated logs, even when fragmented across distributed networks. IoT-specific solutions must sift through disorganized logs to identify security-relevant entries, such as anomalous sensor readings or unauthorized firmware access attempts. Existing IoT-specific approaches, such as behavioral model extraction and source code analysis techniques, contribute to resolving these issues but have limitations. For example, process mining and supervised learning approaches model device behavior through operational logs [16,17], enabling administrators to effectively correlate log patterns with IoT device states [18]. However, in IoT contexts, these approaches rely heavily on accurate log text semantics, and errors in this analysis can lead to faulty models [19]. Conversely, source code analysis techniques map log messages to execution nodes in programs, helping administrators track connections between logs based on control flow. This enables deeper insights into temporal relationships between events but often requires extensive computational resources, making these approaches struggle with scalability in large IoT deployments [20]. These persistent challenges are rooted in the limitations of analysis techniques that depend on the simple chronological order of entries or text-based keyword matching, which remain inadequate for reconstructing a complete picture of events in complex IoT environments. Specifically, such methods struggle with the fact that devices often perform multiple, independent actions simultaneously, thereby obscuring the true cause-and-effect relationships between log entries. For instance, a smart lock might log user authenticated, followed by an interleaved, periodic battery status message, and then motor driver fault. An analyzer relying solely on temporality or keyword matching would likely fail to associate the authentication with the subsequent motor fault, thereby failing to identify the high-level event: a failed unlock attempt. This semantic fragmentation of logs presents an opportunity for adversaries to conceal malicious activities. Thus, there is a critical need for an approach that integrates the real-time responsiveness and flexibility of process mining with the structural efficacy and event flow tracking capabilities of source code analysis [21], while minimizing resource consumption and semantic ambiguities inherent to IoT systems [22].

To address these challenges in IoT systems, we propose Log2Evt, an advanced approach for IoT devices that tightly integrates log analysis with execution paths. Log2Evt offers three key advantages over existing approaches: (1) it accurately tracks logs across program function calls by monitoring real-time execution paths across IoT workflows, providing a complete picture of how logs are generated and passed between functions; (2) it minimizes reliance on complex semantic analysis by utilizing execution context from the program's call stack, improving the precision of log correlation without excessive computational overhead; and (3) it enables administrators to pinpoint log origins to specific IoT components, offering real-time access to device states and accelerating threat response through integration with IoT management platforms, all without disrupting operational efficiency. Revisiting the smart lock example, Log2Evt would identify that both the user authenticated and motor driver fault logs originate from the same high-level execution trace corresponding to an unlock operation. Consequently, it correctly correlates these entries into a single, cohesive event while disregarding the unrelated battery status log.

Log2Evt is optimized for IoT device log analysis by employing techniques specifically designed for resource-constrained and heterogeneous smart environments. It utilizes lightweight tracing tools to dynamically monitor IoT firmware and real-time operating system

(RTOS) functions, capturing execution states without requiring device recompilation. This capability is essential for embedded systems with limited computational flexibility. The approach first identifies logging sources specific to IoT workflows, such as sensor data pipelines or edge compute tasks, and determines whether logs originate from structured frameworks or direct hardware-level writes. Probes are then deployed on these functions or IoT-specific storage interfaces to record call stacks and log content during execution. For low-power devices, it minimizes overhead by selectively tracing mission-critical function. To address IoT's fragmented log sequences, Log2Evt maps logs and execution paths to a hierarchical tree structure, tagging nodes with IoT-centric metadata like device type or edge cluster affiliation. It combines the Common Ancestor algorithm with an optimized Tarjan-LCA variant, which efficiently identifies the Lowest Common Ancestor, to reconstruct event sequences from distributed IoT operations, such as correlating a compromised gateway's logs with downstream actuator anomalies. This approach enables granular auditing of cross-device attacks while maintaining compatibility with lightweight IoT protocols, ensuring efficient analysis even in bandwidth-constrained smart societies.

To evaluate this approach, we designed several experiments based on CentOS with SystemTap installed. In Section 4.3, Log2Evt workflow is demonstrated, and the impact of thresholds on efficacy is evaluated. In Section 4.4, the efficacy of this approach is improved by 15% compared to the rule-based matching approach. In Section 4.5, it is verified that the time and space consumption has low impact on program operation.

The main contributions of this paper are listed as follows:

- We develop a theoretical framework to model causal relationships in fragmented IoT logs through execution context awareness, explicitly tailored for IoT firmware and RTOS. By defining event boundaries via execution path ancestry in IoT workflows, rather than relying on timestamps or log semantics, our approach reconstructs high-level device operations from disordered logs, countering evasion tactics like device spoofing or distributed data injection attacks.
- We introduce an approach to dynamically correlate IoT device runtime behaviors with log generation processes—without modifying resource-constrained IoT hardware. This eliminates dependency on predefined templates, critical for heterogeneous IoT systems where structured and unstructured logs coexist.
- We design a multi-granular event abstraction framework to resolve ambiguities in IoT logs through hierarchical execution trace analysis. By leveraging shared ancestry in device firmware execution and adaptive thresholding, it bridges semantic gaps in IoT-specific logs while maintaining linear scalability for large-scale deployments.
- In the designed experiments, Log2Evt demonstrates 15% higher accuracy in event reconstruction and 93% lower deployment overhead compared to baselines, while establishing reproducible metrics for cross-approach comparisons in fragmented log analysis.

2. Related work

While recent advancements in process mining [23,24] and log clustering [25,26] have enhanced log analysis, these methods are often inadequate for addressing the unique challenges of IoT environments. Process mining techniques, for example, typically assume semantic consistency in log messages to discover behavioral models. This assumption is frequently violated in heterogeneous IoT ecosystems characterized by diverse and unstructured logging practices. Similarly, many modern log parsers require training on specific formats, which limits their applicability to new or proprietary device logs.

Log2Evt proposes a more fundamental approach by shifting the focus from log content to log origin. Unlike methods that contend with format diversity, our technique is inherently agnostic to log message content, establishing correlations based on the shared code execution ancestry of log entries. This provides a robust mechanism for event reconstruction across heterogeneous devices without relying on brittle, predefined templates. Furthermore, Log2Evt is designed specifically for resource-constrained settings. It employs a lightweight, on-device probe for data capture and offloads computationally intensive analysis to a backend system with greater resources. This distributed architecture contrasts sharply with approaches that impose an unsustainable analytical burden on low-power sensors or gateways, ensuring our method is well-suited for practical, large-scale IoT deployments.

Table 1 compares Log2Evt with state-of-the-art approaches in IoT log analysis, emphasizing its capabilities in addressing device heterogeneity and resource constraints. Log2Evt constructs IoT device execution trees by analyzing runtime call stack traces from IoT firmware and RTOS, using an adaptive Tree Conversion algorithm to map log correlations across distributed workflows. Granularity is dynamically tuned based on IoT device capabilities — for instance, prioritizing low-memory sensors versus high-performance edge servers — ensuring efficient log analysis without overburdening resource-limited nodes. Unlike existing approaches, Log2Evt operates without firmware modifications, enabling seamless deployment across diverse IoT environments, from embedded ARM microcontrollers to ZigBee gateways, through standard debugging interfaces.

2.1. Process mining

Process mining techniques are increasingly critical for analyzing IoT systems, where they discover, monitor, and optimize device workflows by extracting insights from distributed event logs [30–32] generated by sensors, actuators, and edge computing nodes. In IoT-enabled systems, the generated process models can be represented using structures such as Petri-nets [33], BPMN [34]. In the process mining, ProM [35], which provides a variety of algorithms to support process mining in the broadest sense, is widely used such as discovering processes, identifying bottlenecks, analyzing social networks, and verifying business rules.

To mine the software development process, Sebu et al. [23] extract hidden information about the process in the event process logs in software development using the ProM. Boxi Yu et al. [24] propose LightAD, an optimized architecture that balances training time, inference time, and performance scores through automated hyper-parameter tuning.

To mine behavior patterns, Michela Vespa et al. [36] present a framework that integrates probabilistic constraints with declarative process mining to address the inherent uncertainty in process execution, offering novel techniques for discovery, conformance checking, and monitoring, demonstrated through proof-of-concept prototypes and evaluated on real-life logs. Moreover, for the mining of user behavior, [37,38] use pre-defined high-level user operations as their references to derive process and user interface flow models. Liu et al. [29] propose a supervised learning approach that matches user behavior patterns and discovers user behavior models using existing process discovery approaches. Further, Zhihan Jiang et al. [39] propose LILAC, achieving superior efficiency compared to existing approaches, significantly reducing queries to language models by leveraging in-context learning and an adaptive parsing cache.

Uncertain data are found in process mining, which contains non-deterministic and random event attributes that may represent many possible real-life events. Pegoraro et al. [40] propose an approach to obtain a complete probability distribution over possible instantiations of uncertain attributes in tracking. Meanwhile, Aleksei Pismerov et al. [41] apply various embedding approaches to a dataset of event logs. By transitioning to log embeddings and applying clustering techniques, the efficiency of process mining is improved. Siyu Yu et al. [42] propose Log3T, a novel log parsing approach with generalization ability

Table 1
Comparison of approaches (see [27]).

Approaches	Dynamic adaptation capability		System integration characteristics		Semantic comprehension depth	
	Particle size adjustment	Fast application adaptation	Graph algorithm application	Root cause tracing	Loose match	Temporal sensitivity
[28]	×	×	×	×	×	×
[27]	×	×	×	×	✓	✓
[29]	×	×	✓	×	✓	✓
Log2Evt	✓	✓	✓	✓	✓	✓

to support new log types in incoming logs. Evaluation on 16 benchmark datasets shows Log3T outperforms state-of-the-art parsers in parsing accuracy and can automatically adapt to new log types in incoming logs.

2.2. Log clustering

Logging serves as a critical diagnostic tool in IoT systems, enabling efficient identification of device failures, network anomalies, and security breaches in resource-constrained smart environments. For IoT systems clustering techniques streamline fault detection by grouping related entries across distributed nodes. This addresses the inefficiency of keyword-based searches [43], which struggle to detect subtle patterns in encrypted LoRaWAN transmissions or correlate multi-device failures in industrial IoT deployments. By organizing logs from gateways, sensors, and actuators into contextually meaningful clusters, administrators can pinpoint issues like firmware crashes in low-power devices or synchronization errors in mesh networks, significantly reducing diagnostic latency while conserving computational resources.

Static clustering is single log row clustering that ignores the order and dependencies between rows. Each log row is assigned to the cluster representing the statement that generated it. Lin Yang et al. [25] propose PLELog, a practical semi-supervised log-based anomaly detection approach that leverages probabilistic label estimation and attention-based GRU neural networks to efficiently identify system anomalies by incorporating the strengths of both supervised and unsupervised approaches. Siyu Yu et al. [44] propose Brain, a novel stable log parsing approach inspired by the observation that the longest common pattern among logs is likely to be part of the log template. AUTOLOG [45] employs program analysis to generate comprehensive runtime log sequences without actual system execution, enabling log-based anomaly detectors to achieve improved performance over existing datasets. LogPrompt [46] utilizes large language models (LLMs) and advanced prompt strategies to address the limitations of existing approaches in terms of interpretability and adaptability to new domains.

Xu Junjielong et al. [26] introduce a novel automatic logging framework that utilizes the in-context learning paradigm of large language models. This approach demonstrates superior logging accuracy while significantly reducing the computational resources typically required for model tuning.

Dynamic clustering is the process of assigning log lines to classes that refer to their original events. DivLog [47] leverages diverse log samples and constructing prompts with appropriate examples, addressing limitations of traditional parsers that rely on heuristics or limited training data. Zeyang Ma et al. [48] investigate the effectiveness of large language models, specifically Flan-T5-small, Flan-T5-base, LLaMA-7B, and ChatGLM-6B, in improving log parsing efficacy over traditional approaches, highlighting the model size and training size impact, and discussing the mixed results of pre-training on log parsing performance.

2.3. Userspace tracing in IoT device development

IoT developers face unique challenges when debugging resource-constrained embedded systems, particularly when working with RTOS and low-power wireless protocol stacks. Symbolic debuggers provide some help, but the task is still complex and challenging. Other than breakpoints and tracing, these tools provide little advanced help. Over the past decade, many tracing tools have emerged in the software stack, and the debugging process can be largely automated if explicit support is provided for these tasks.

Kprobes [49] dynamically modifies the kernel code on the x86 architecture to provide the ability to insert custom detection based on breakpoints or based on dynamic jumps. Breakpoints in both of these approaches can have some performance impact, and neither of these approaches guarantees access to local variables in the middle of functions that the compiler has optimized. SystemTap [50] is based on Kprobes and Linux Kernel Markers to provide a scriptable language for creating probes. Adel Belkhiri et al. [51] introduce a comprehensive tracing-based framework for DPDK applications to collect and analyze performance data, enabling practitioners to diagnose performance anomalies and optimize application efficiency with minimal overhead.

3. Methodology

3.1. Overview

In Log2Evt, we specialize in extracting security-critical events from IoT devices by analyzing firmware execution paths, treating log messages as contextual markers within constrained embedded environments. As shown in Fig. 2, our methodology employs three IoT-optimized phases. In the source location phase, we deploy lightweight probes by analyzing interactions between IoT firmware and logging mechanisms: for logging frameworks, we identify framework-specific interface functions, while for direct file logging, we monitor Virtual File System (VFS) write operations and authenticate target logs through inode verification. The dataflow trajectory tracing phase then bridges static logging points with dynamic execution context using SystemTap as a real-time collector, which deploys Kprobes to capture complete call stacks during function execution, preserving critical runtime information often lost in offline logging systems. This dual capture of structured log data and concurrent execution paths enables precise event attribution. Finally, the event chain integration phase transforms raw trajectories into semantic events through hierarchical tree construction and relational analysis, where our segmentation algorithm clusters correlated logs while filtering noise, effectively elevating low-level log sequences to comprehensible system events through structural pattern recognition and multi-level aggregation. This pipeline ensures event reconstruction maintains both execution context and operational semantics through continuous instrumentation-to-interpretation synchronization.

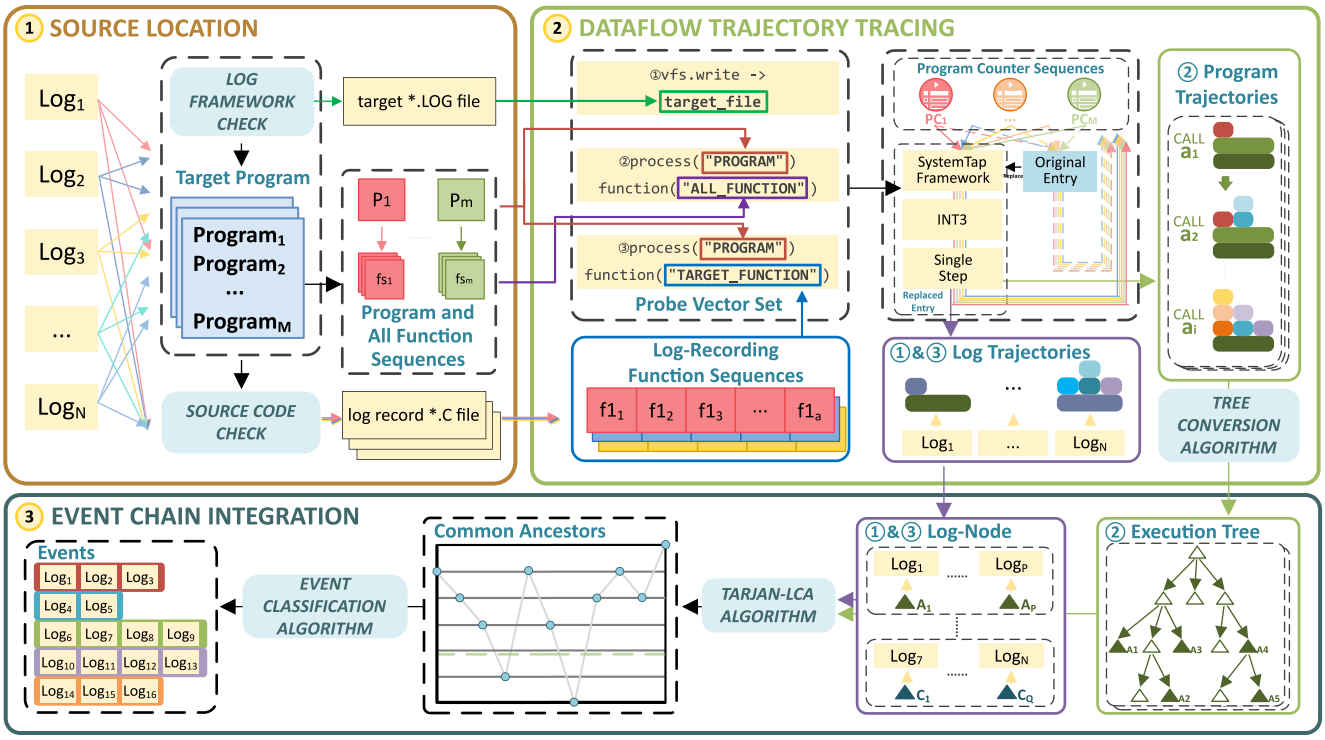


Fig. 2. Overview of Log2Evt.

3.2. Source location

A fundamental challenge in log-based auditing is accurately attributing log messages to their originating execution contexts — essential for reconstructing meaningful system events from fragmented logs. Traditional approaches typically rely on static program analysis or predefined logging rules, which present two inherent limitations: they cannot adapt to dynamically generated logs in modern componentized systems with varying logging mechanisms across software layers, and they fail to capture the causal relationships between log entries and their triggering execution paths, particularly when logs are routed through intermediate services. These shortcomings compromise audit accuracy.

Our key insight is that all log generation must ultimately interface with system-level I/O operations, regardless of the logging mechanism used — whether direct file writes or framework-mediated outputs. By instrumenting this critical junction, we establish a unified monitoring layer that preserves execution context without requiring prior knowledge of application-specific logging implementations.

As illustrated in Fig. 2, our implementation operationalizes this insight through dual monitoring strategies. For applications writing directly to files, we intercept VFS operations to trace back to the originating program's call stack. When logs are routed through frameworks like Syslog, we instead target the final interface functions that bridge applications to the logging service. This bifurcated approach ensures comprehensive coverage while maintaining system integrity — monitoring points are selected at the last common boundary before log data leaves application control, thus capturing authentic execution contexts without perturbing normal operations.

The initial step involves identifying the specific programs requiring monitoring. Log data can originate from diverse sources, including but not limited to application operations, system activities, network communications, database transactions, and security events. These logs are typically captured through various monitoring mechanisms such as system audit tools, network packet capture utilities, application instrumentation, and dedicated logging frameworks. These log entries commonly contain essential process identifiers, and execution path information, which form the basis for subsequent analysis.

For programs that generate log files directly, the data writing process typically occurs through VFS. VFS provides a standardized interface that enables applications to perform file operations. As Fig. 3 illustrates, when a program needs to write or read data, it first initiates the operation in user mode and passes the data to VFS. The approach then transitions to kernel mode, where VFS executes sys_write/sys_read functions to transmit the data to the underlying file system. Finally, the data is written to disk storage.

Since log messages are passed to VFS through chained function calls, by monitoring the write function of VFS, we can directly obtain the corresponding call stack at the time of writing when the program executes the write.

In the alternative logging approach, messages are routed through logging frameworks like Syslog on Linux systems. Syslog employs User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) protocols for message transmission. Unlike direct file writing where call stacks can be traced through VFS, applications utilizing Syslog first transmit messages via socket connections to the Syslog daemon, which then writes them to files through VFS. This indirect routing prevents direct VFS monitoring from capturing the originating program stack. When attempting to trace function call chains as in the previous scenario, VFS monitoring can only detect the logging service's activities, failing to capture the message forwarding process from the source application to the logging service, as illustrated in Fig. 4.

Programs typically implement an interface function to interact with the logging framework. This function processes all logging events according to predefined configurations and formatting rules, then forwards them to the framework by invoking public library functions such as **syslog()**.

In the scenario described in Section 1, PAM (Pluggable Authentication Modules) manages remote login authentication through SSH protocol for remote shell access. PAM's logging output is directed to

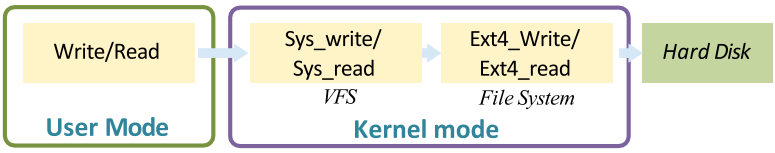


Fig. 3. Function call path of VFS.

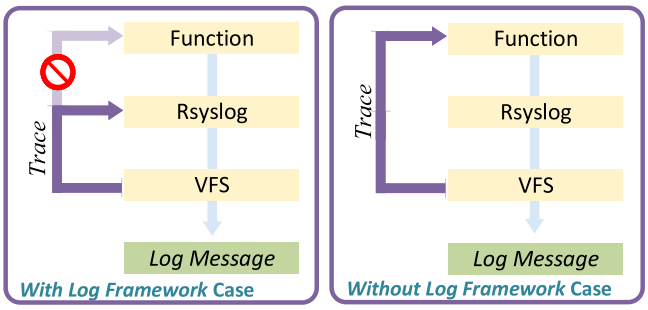


Fig. 4. Traceability of the log service.

the *SECURE* log file located in */var/log/* via the Rsyslog framework. As illustrated in Fig. 5, when processing the log message requirement "user ingroup nopasswdlogin" not met by user "niki", the call trace terminates at **pam_vsyslog()**. Since this function serves as PAM's final logging interface, monitoring it enables accurate capture of both message content and corresponding stack traces.

This monitoring approach can be extended to other applications utilizing logging frameworks, provided system developers can identify the interface functions responsible for framework communication during program execution.

3.3. Dataflow trajectory tracing

Our fundamental insight recognizes that meaningful event reconstruction requires continuous yet lightweight capture of execution trajectories, representing the specific, ordered sequence of functions, methods, or key code segments executed by the program leading up to a particular state or event. This capture must be synchronized with log generation. Rather than instrumenting entire codebases or relying on sampled profiles, we strategically trace control flow at the narrow interface between program logic and log emission mechanisms.

Our implementation realizes this through Kprobes — a dynamic kernel instrumentation mechanism — orchestrated via SystemTap's abstraction layer. Unlike static tracing frameworks that require persistent code modifications, Kprobes enable hot-pluggable monitoring points inserted at runtime. This is crucial for auditing closed-source components or systems with frequent updates, where traditional recompilation-based approaches are inadequate. SystemTap enhances this capability through its domain-specific language, allowing declarative specification of trace points while managing low-level complexities such as register preservation and context switching.

The tracing methodology deliberately avoids full-stack recording, instead focusing on two key dimensions:

Vertical Context: Captured through user/kernel-space stack unification, preserving call hierarchy from application logic to system call execution.

Horizontal Correlation: Achieved via temporal synchronization between log writes and their preceding execution trajectories.

3.3.1. Kprobes

Log2Evt implements dynamic program monitoring and custom instruction execution through the SystemTap framework, with Kprobes serving as the primary instrumentation mechanism. The approach enables users to establish monitoring points by strategically placing Kprobes and associating them with custom handler functions. Upon reaching a Kprobe point during kernel execution, the approach temporarily diverts control flow to execute the corresponding handler function before returning to the normal execution path.

Kprobes provides kernel instrumentation mechanisms that enable the insertion of hooks into kernel functions, allowing user-defined code execution at these instrumentation points. As illustrated in Fig. 6, the Kprobes mechanism operates through the following sequence:

When registering a Kprobe point, the approach first creates a backup of the target instruction at the monitored location. It then replaces the original instruction's entry point with a breakpoint instruction.

Upon CPU execution reaching the breakpoint instruction, a trap is triggered. The trap handler preserves the current CPU register state and invokes the designated trap processing function. This function establishes the Kprobes execution context and calls the user-registered pre_handler callback, passing both the struct Kprobes structure address and preserved CPU register data as parameters.

Kprobes executes the previously backed-up instruction in single-step mode, followed by invoking the user-registered post_handler function. The execution path then returns to the normal instruction stream following the probe point.

Leveraging Kprobes, Log2Evt implements monitoring points at strategic locations within target program functions that precede log-writing operations. When log-writing occurs, Log2Evt captures both the program's current call stack trace and the corresponding log content, enabling comprehensive runtime analysis.

3.3.2. SystemTap

Log2Evt utilizes SystemTap as an abstraction layer over Kprobes to streamline the monitoring implementation. SystemTap provides a high-level scripting interface that enables developers to create sophisticated analysis and monitoring scripts for problem diagnosis. As illustrated in Fig. 7, SystemTap translates administrator-provided scripts into Kprobe implementations through a multi-stage process. When a user submits a script file, SystemTap first generates an Abstract Syntax Tree (AST) through parsing. During the elaboration phase, it resolves symbolic references, transforms the code into C, and compiles it into a loadable kernel module. Finally, SystemTap loads the module into the kernel, initiates system monitoring, and establishes data channels to relay collected information to userspace.

3.3.3. Set Kprobes

SystemTap exposes the **print_ubacktrace()** function to capture and output the current user-space task's call stack trace. Upon Kprobe trigger events, this function enables immediate stack trace extraction at the instrumentation point.

The analysis requires continuous stack monitoring to detect and record all runtime call stack modifications. The implementation places Kprobes at function entry points, executing instrumentation code upon each function invocation. When triggered, these probes capture the current call stack, providing a comprehensive view of the program's call hierarchy.

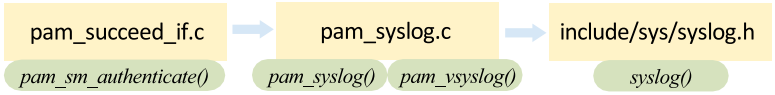


Fig. 5. Function call path of a login log.

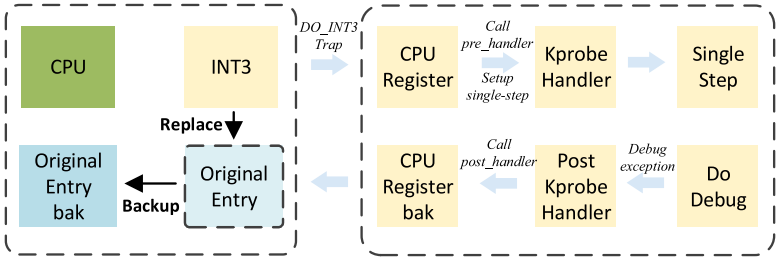


Fig. 6. Workflow of Kprobes.

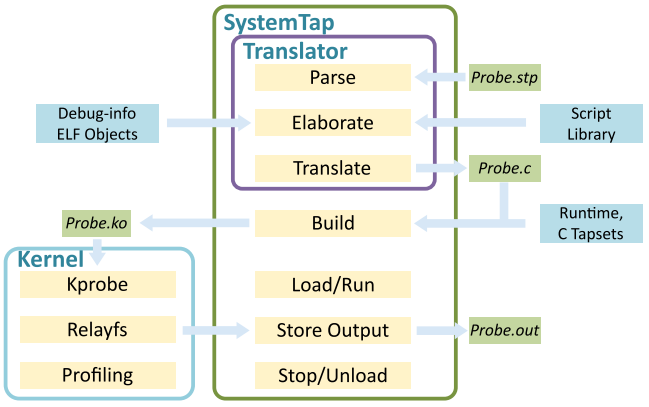


Fig. 7. Workflow of SystemTap.

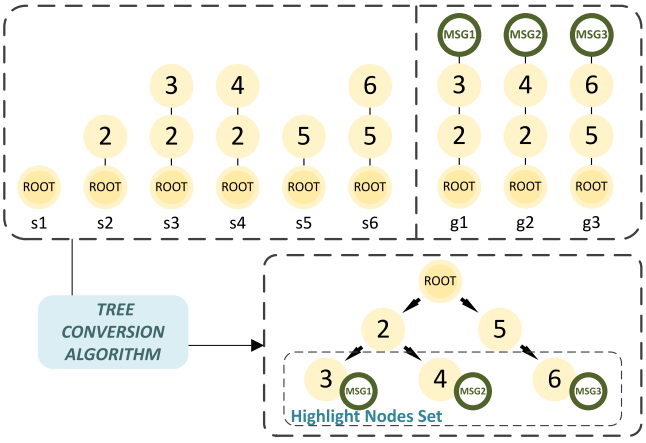


Fig. 8. An example of *Tree conversion*.

Formally, let C denote an atomic function call operation, and let $s = \{C_1, C_2, \ldots, C_m\}$ represent an ordered trajectory sequence of function executions, comprising m sequential calls. We define O_a as the primary operation that executes **print_ubacktrace()** and emits trajectory s. Building upon O_a, we further define O_b as an augmented operation that encapsulates both the base functionality and extends it to capture parametric log text, subsequently generating the log-path binary tuple $g = \{s, message\}$.

Let $E = \{C_1, C_2, \ldots, C_n\}$ be the original program execution sequence, where n is the length of this sequence. Log2Evt inserts the operation O_a, O_b into all general functions and log recording functions of the target program, then E becomes $E' = \{C_1, O_a, C_2, O_a, \ldots, O_a, C_n\}$.

As detailed in the "Log Framework Case" in Fig. 4, when programs utilize logging frameworks, O_b is instrumented at log recording function entry points. The O_b operation executes prior to message propagation through the logging framework.

For the "Without Log Framework Case" described in Fig. 4, when programs perform direct file-based logging, Log2Evt implements VFS activity monitoring, constraining its scope to the target file through inode-based filtering.

Log2Evt implements SystemTap-based Kprobe instrumentation on VFS write operations. The probe activation is constrained to specific file operations through inode-based filtering, ensuring targeted monitoring.

Notably, despite the function call path traversing from the target program to the VFS (illustrated in Fig. 3), memory address analysis enables precise delineation between the two execution contexts. This boundary identification facilitates accurate determination of the recording function's memory address.

3.4. Event chain integration

Reconstructing high-fidelity system events from interleaved logs and fragmented execution traces represents a fundamental challenge in log auditing. Traditional graph-based correlation methods struggle with two key limitations: they impose rigid parent–child dependencies ill-suited for modern parallelized systems where logs may originate from concurrent threads, and they scale poorly with execution depth. These shortcomings enable attackers to evade detection by distributing attack footprints across shallow but broad execution branches.

Our insight pivots on three principles for robust event reconstruction: hierarchical causality preservation, adaptive granularity control, and linear-time scalability. Events manifest as clusters of causally related logs, bounded by shared execution ancestry rather than temporal adjacency. Segmentation thresholds must dynamically adapt to execution context depth to counter adversarial dispersion tactics. Practical auditing demands sub-quadratic complexity regardless of system scale or attack sophistication.

The proposed approach operationalizes these principles through a novel fusion of tree-based ancestry analysis and adaptive thresholding. Unlike prior tree construction methods that require complete execution traces, our differential tree conversion incrementally builds hierarchical models from partial trajectories — a critical adaptation for auditing long-running systems. By anchoring event boundaries to the lowest common ancestors of log-correlated nodes, we inherently account for parallel execution paths while maintaining deterministic segmentation.

Following trajectory acquisition, Log2Evt transforms the captured function call sequences into hierarchical tree structures through node

clustering. This transformation leverages the Common Ancestor algorithm, which comprises three components: Tree Conversion, Tarjan-LCA (Lowest Common Ancestor), and Event Segmentation algorithms.

Algorithm 1: Execution Path Tree Construction and Log Node Highlighting

Input: A sequence of function call trajectories $S \leftarrow \{s_1, s_2, \ldots, s_n\}$; A sequence of (log-path, log-message) groups $G \leftarrow \{g_1, \ldots, g_k\}$;
Output: Program execution path tree T; Highlight log node set $\alpha \leftarrow \{N_1, \ldots, N_p\}$

// Initialize tree and supporting structures
1 Tree $T \leftarrow$ new Tree();
2 Node N;
3 Set $\alpha \leftarrow \emptyset$;
// Build tree structure from call trajectories
4 **foreach** *path* $s_i \leftarrow \{C_i^1, C_i^2, \ldots, C_i^m\}$ *in* S **do**
5 Variable $j \leftarrow 1$ **if** $i > 1$ **then**
// If not the first path, find common prefix with s_{i-1}
6 **while** $j \leq length(s_{i-1})$ *and* $j \leq m$ *and* C_i^j *points to* C_{i-1}^j **do**
7 $j++$; // Calls match, extend common prefix
8 $N \leftarrow T$.locateNode($\{C_i^1, \ldots, C_i^{j-1}\}$) ; // Move N to end of common prefix in T
9 **while** $j \leq m$ **do**
// Add remaining calls from s_i as new nodes
10 $N \leftarrow N$.addSon(C_i^j) ;
11 j++ ;
// Attach log messages and identify highlight nodes
12 **foreach** *group* $g_{idx} \leftarrow (path_{idx}, message_{idx})$ *in* G **do**
13 $N \leftarrow T$.findNode(path_{idx}) ; // Find node in T for this log's path
14 **if** $N \neq null$ **then**
// If path exists in tree
15 N.appendLogMessage(message_{idx}) ;
16 α.add(N)
17 **return** T, α

3.4.1. Tree conversion algorithm

Given a function call trajectory sequence $S = \{s_1, s_2, \ldots, s_n\}$ and its corresponding binary log-path grouping sequence $G = \{g_1, g_2, \ldots, g_k\}$, Log2Evt constructs a hierarchical tree structure where nodes responsible for log message generation are distinctly marked, as illustrated in Fig. 8.

To address this challenge, we perform a traversal of sequences S and G to construct the program execution tree. Let C_i^j denote the jth function invocation within trajectory s_i. The process, formalized in **Algorithm** 1, encompasses three primary phases. First, the differential analysis phase compares adjacent trajectories to identify their first divergent node (lines 4–7). Subsequently, the tree construction phase incorporates these identified differences into the tree structure through child node insertion operations (lines 8–11). Finally, during the node annotation phase, log-related nodes are identified using the log-path binary grouping, marked accordingly, and augmented with their corresponding log messages (lines 12–16).

Multiple tree matches for a function call path s, which is tied to a log entry in group g, arise when identical call sequences execute repeatedly during a program's runtime. Such repetitions occur, for example, in loops or through frequently called subroutines. Each execution, while structurally identical, represents a distinct temporal instance.

Algorithm 2: Tarjan-LCA

Input: execution path tree T; Highlight log node set $\alpha \leftarrow \{N_1, N_2, \ldots, N_n\}$
Output: LCA node set $\beta \leftarrow \{M_1, M_2, \ldots, M_{n-1}\}$

// Initialize data structures
1 Query *query*;
2 Bool Array *marked*;
// Definition of union-find helper functions
3 **def** *Find(u)*:
// Finds representative of the set containing u with path compression
4 return u points to ancestor[u] ? u : (ancestor[u] ← Find(ancestor[u])) ; // If u is not its own ancestor, recursively find and update
5 **def** *Union(u, v)*:
// Merges the sets containing u and v
6 return ancestor[Find(u)] ← Find(v) ; // Set ancestor of u's representative to v's representative
// Main Tarjan-LCA algorithm
7 **def** *Tarjan_LCA(u)*:
8 ancestor[Find(u)] ← u ;
9 **for** $v \in u.son$ **do**
10 Tarjan_LCA(v) ; // Recursively call LCA for child v
11 Union(u, v) ; // Merge the set of v into the set of u
12 ancestor[Find(u)] ← u ; // Ensure u remains the representative of the merged set
13 marked[u] ← True ;
14 **foreach** $N_i \in \alpha$ **do**
// Iterate through highlight nodes to form query pairs (N_i, N_{i+1})
15 **if** $N_{i+1} \leftarrow u$ *and* *marked*$[N_i] \leftarrow$ *True* **then**
// If u is one node of a query (N_i, u) and N_i is already marked
16 $M_i \leftarrow$ ancestor[N_i] ; // The LCA of (N_i, u) is the current ancestor of N_i
// Call main algorithm to start the process
17 Tarjan_LCA(T.root) ; // Begin LCA computation from the root of the tree T

To resolve this, we employ chronological ordering. This means a log entry is matched to the instance of path s in the execution tree T that corresponds to the log's actual time of generation. This time is typically determined by log timestamps or the log's sequential order. The rationale is to ensure each log entry is linked to its unique, temporally accurate execution context within T. This precise mapping is vital for maintaining the integrity of event reconstruction, particularly for analyzing the ordered operations common in AIoT systems. Our approach presumes sufficient temporal information is available for such deterministic mapping.

3.4.2. Tarjan-LCA algorithm

Following the transformation of function call trajectory sequence S and log-path binary groups G into tree structure T with highlighted node-set α, event integration proceeds through analysis of common ancestor maximization between adjacent highlighted nodes. The determination of maximum common ancestors reduces to the Lowest Common Ancestor (LCA) problem, where the LCA depth represents the maximum value. Given that each function transition corresponds to a

unique tree node, resulting in substantial tree dimensionality, Log2Evt implements the Tarjan-LCA algorithm to compute the LCA node-set $\beta = \{M_1, M_2, \ldots, M_{n-1}\}$, where M_j represents the LCA of adjacent nodes N_j and N_{j+1}. Leveraging Concurrent Set operations, the computation of β achieves time complexity $O(N + Q)$, where Q denotes the total query count.

Algorithm 3: Event Segmentation

Input: LCA node set $\beta \leftarrow \{M_1, M_2, \ldots, M_{n-1}\}$; Program execution path tree T; Highlight log node set $\alpha \leftarrow \{N_1, N_2, \ldots, N_n\}$; Threshold ratio RA

Output: Event sequences $\{E_1, E_2, \ldots, E_{count}\}$

```
// Initialize variables and array
1  Variable count, threshold ;
2  Array depth ;
3  count ← 0 ;
4  threshold ← 0 ;
// Calculate the depth of each LCA node and the initial threshold
5  foreach M_i in β do
       // Iterate through each LCA node
6      depth[i] ← T.depth(M_i) ; // Get depth of LCA node M_i in tree T
7      threshold ← threshold + depth[i] ;
8  threshold ← threshold * RA ; // Calculate depth and get final threshold
// Initialize the first event sequence with the first highlight node
9  E_0.insert(N_1) ;
// Segment events based on depth and threshold
10 foreach depth[i] in depth do
       // Iterate through calculated depths of M_i
11     if depth[i] > threshold then
           // If LCA depth is greater than threshold, start a new event
12         count ← count + 1 ;
13     E_count.insert(N_{i+1}) ; // Add the next highlight node N_{i+1} to the current event E_count
```

In **Algorithm** 2, we define functions Find and Union (lines 3–6) that form the foundation of the Tarjan-LCA algorithm. For each vertex u encountered during depth-first search traversal, we consider a subtree of T rooted at u (line 8). The Tarjan_LCA function initializes by designating u as the ancestor of its disjoint set. For each adjacent vertex v connected to u, the algorithm recursively invokes Tarjan_LCA, followed by a Union operation on u and v, subsequently re-establishing u as the set's ancestor (lines 9–13). Upon completing the traversal of u's descendants, the algorithm marks u as processed and queries the LCA utilizing the marked vertex array (lines 14–16). Finally, the highlighted node-set α is enqueued into the query queue, and the Tarjan-LCA algorithm initiates its execution from the root of the program execution tree T (line 17).

3.4.3. Event segmentation algorithm

The process of segmenting raw log sequences into meaningful, high-level events begins by analyzing the structural relationships between consecutive log entries. Following the application of Algorithm 2, we first determine the Lowest Common Ancestor (LCA) for every pair of adjacent highlighted nodes in the sequence. The depth of this LCA within the program's execution path tree is a critical piece of information; it serves as a proxy for the semantic relatedness between the two nodes. A shallow LCA, corresponding to a smaller depth value, indicates that the nodes share a recent ancestor and are likely part of the same localized operation. Conversely, a deep LCA suggests their common ancestor is far up the execution hierarchy, implying the nodes belong to distinct functional contexts.

With this structural information established, **Algorithm** 3 leverages these LCA depths to partition the log stream. The fundamental premise of this algorithm is that a significant increase in the LCA depth between consecutive log nodes signals a transition from one high-level event to another. To formalize this, the algorithm first computes a dynamic partitioning threshold. As detailed in lines 5–7, it iterates through the set of all LCA nodes, β, summing their respective depths. This sum is then scaled by a user-defined ratio, RA, to produce the final $threshold$. This RA parameter provides crucial flexibility, allowing the sensitivity of the segmentation to be tuned; a higher RA will result in coarser, larger events, while a lower value will produce more fine-grained event boundaries.

The segmentation procedure itself begins by initializing the first event sequence, E_0, with the very first highlighted log node, N_1 (line 8). Subsequently, the algorithm iteratively traverses the pre-computed LCA depths (lines 9–13). For each step in the iteration, it compares the LCA depth between the current node and the next against the calculated $threshold$. If the depth value is less than or equal to the threshold, it signifies contextual continuity, and the subsequent node, N_{i+1}, is appended to the current event, E_{count}. However, if the depth exceeds the threshold (line 10), it indicates a contextual break. In this case, the algorithm concludes the current event and initiates a new one by incrementing the event counter, $count$ (line 11). The node N_{i+1} is then assigned as the first entry in this newly created event sequence. This process is repeated until all highlighted nodes have been assigned, resulting in a complete partitioning of the log sequence into a set of discrete, semantically coherent events.

4. Evaluation of Log2Evt

To evaluate the effectiveness of our proposed approach, we conducted comprehensive experimental analysis across four primary modules:

- *Theoretical analysis:* We evaluate Log2Evt against two baseline approaches. Our system achieves $\mathcal{O}(N)$ complexity versus baselines' $\mathcal{O}(ND)$, providing enhanced flexibility through program execution tree analysis.
- *Functional experiments:* We demonstrate the operational workflow of Log2Evt using an SSH login case study, visualizing call stack sequences through flame graph analysis [52]. Additionally, we assess the sensitivity of the threshold parameter RA on system performance.
- *Efficacy experiments:* We evaluate the system's performance through comparative analysis using established metrics including Purity, Rand Index, and Fowlkes–Mallows Index, benchmarking against behavior pattern alignment-based and rule-based matching approaches.
- *Time and space occupation experiments:* We examine both theoretical and empirical performance characteristics of Log2Evt, including computational complexity analysis and empirical time consumption measurements.

We present the experiment results from Sections 4.1 to 4.5, respectively.

4.1. Theoretical analysis

This section presents a comparative analysis between our proposed technique Log2Evt and two established baseline models. We systematically evaluate these models to provide a comprehensive assessment of their performance and methodological characteristics.

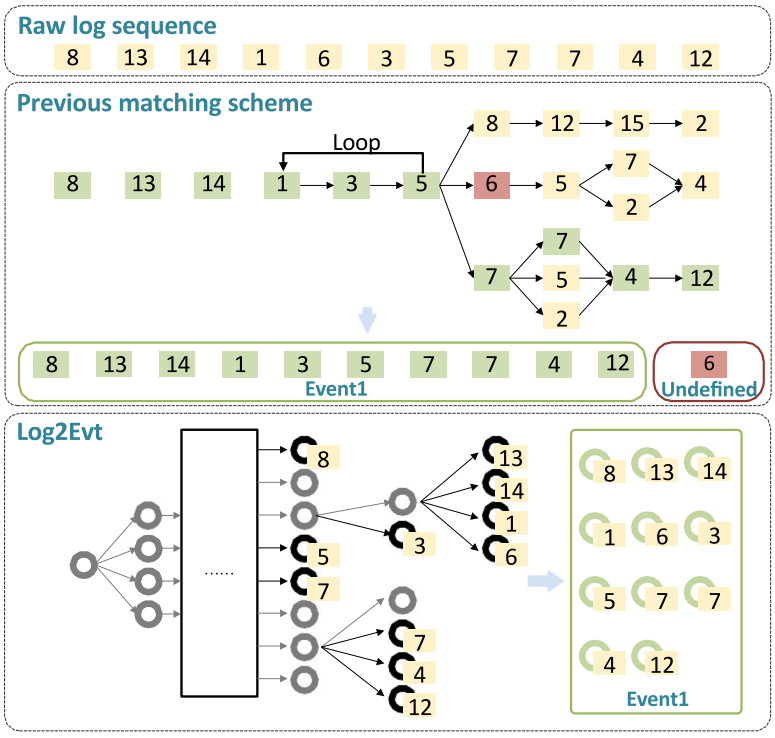


Fig. 9. Comparison of Log2Evt and the previous matching approach.

Table 2
Comparison in terms of complexity. (N: sequence length of log messages, D: number of detectable event types.)

Approaches	Space complexity			Time complexity
	Pre-process	Post-process	Sum	
Log2Evt	–	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
[29]	$\mathcal{O}(D)$	$\mathcal{O}(N\,D)$	$\mathcal{O}(N\,D+D)$	$\mathcal{O}(N\,D)$
[28]	$\mathcal{O}(D)$	$\mathcal{O}(N\,D)$	$\mathcal{O}(N\,D+D)$	$\mathcal{O}(N\,D)$

- Liu et al. [29] propose a hierarchical approach that begins with activity mining to transform low-level event logs into abstract logs. These abstract logs capture user actions, which are subsequently analyzed using established process discovery techniques to derive user behavior models.
- Khan et al. [28] present an object-oriented framework for event log analysis. Their methodology employs association rule mining on object-based event log representations. The extracted rules are then transformed into temporal event sequences, with causal inference techniques applied for validation.

Both Liu et al. [29] and Khan et al. [28] employ rule-based matching techniques to establish mappings between logs and events. Liu et al. further enhance efficacy and efficiency through the application of Petri nets. In contrast, our proposed approach, Log2Evt, augments the analysis by incorporating function call information captured during program execution, in addition to the log message content. Table 2 presents a comparative analysis of Log2Evt and the baseline models in terms of time and space complexity, where N represents the sequence length of log messages and D denotes the number of detectable event types.

Regarding preprocessing requirements, Liu et al. [29] and Khan et al. [28] both necessitate predefined matching templates, resulting in space complexity that scales linearly with the number of detectable events. In contrast, our approach, Log2Evt, determines log message relationships by analyzing common ancestor patterns in the program execution tree, eliminating the need for explicit matching templates.

The sublinear-time design of Log2Evt provides inherent scalability advantages as log volumes grow. Nevertheless, highly distributed IoT deployments with concurrent device operations present distinctive challenges. Maintaining real-time analysis capabilities becomes increasingly demanding when coordinating execution paths across large-scale heterogeneous infrastructures, where resource-constrained devices coexist with high-capacity nodes. To preserve responsiveness in such environments, future implementations could incorporate distributed coordination mechanisms. Edge gateways might perform preliminary event reconstruction before forwarding consolidated metadata to centralized systems. This layered strategy would retain the core methodology while adapting to the hierarchical nature of operational IoT networks.

While Liu et al. [29] and Khan et al. [28] focus exclusively on log text analysis, Log2Evt encompasses both log text and function call transitions, resulting in higher computational complexity and time requirements. However, our experimental evaluation demonstrates that this additional computational overhead is justified by the substantial improvements in detection efficacy compared to both baseline approaches.

From a theoretical perspective, Log2Evt demonstrates enhanced flexibility through its integration of dynamic debugging techniques, contrasting with the rule-based event matching approaches employed by [28,29]. As illustrated in Fig. 9, Log2Evt achieves event clustering by analyzing the positional relationships of log output functions within the program execution tree, eliminating the need to consider boundary conditions and variant operations.

The implications of this theoretical framework extend beyond its analytical capabilities to its practical performance and scalability. The architectural design, centered on a strategic separation of duties between on-device data gathering and backend analysis, is not merely a conceptual choice but a foundational element that ensures efficiency in resource-constrained IoT environments.

This performance profile is best understood by examining its core components. On the end devices, the Log2Evt component functions as a minimalist observer, with its sole responsibility being to note when specific software activities occur and to report these observations.

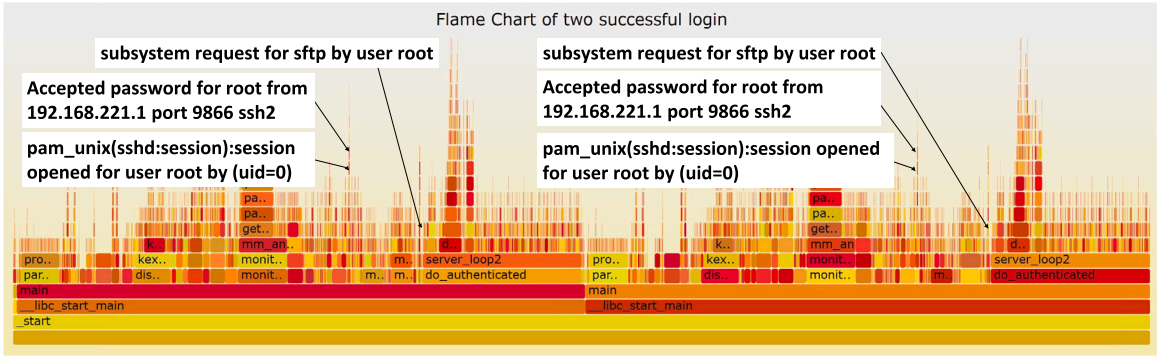


Fig. 10. Flame graph of two login events.

Table 3
Log datasets characteristics.

Log source	Entry count	Message type ground truth
Loghub (OpenSSH)	655,146	Obtained from He et al. [53][a]
Loghub (Linux)	25,567	Obtained from He et al. [53][a]
Linux Application Log Dataset	21,760	Available online[b]

[a] Loghub, https://github.com/logpai/loghub.
[b] Linux Application Log Dataset, https://github.com/czz19981215/Linux-Application-Log-Dataset.

By offloading all complex calculations and data interpretation, the processing overhead is kept exceptionally low, preventing interference with a device's primary functions. The memory footprint is similarly minimal and, crucially, constant, as the system does not need to store a growing history of events on the device itself. This makes the approach theoretically sound for even the most stringent memory limitations.

In contrast, all computationally intensive tasks are handled by the backend analysis engine. While operating in a resource-rich environment, the algorithms for event reconstruction and pattern identification are still chosen for efficiency to ensure that the analysis process remains timely even as data volume grows. This architectural separation is also what makes the framework inherently scalable. The on-device component operates in isolation, allowing the system to support a growing number of devices without creating a cascading load. The greater data throughput from a larger network is managed by the backend's ability to be scaled horizontally, for instance, by distributing the analytical workload across a cluster of servers.

Therefore, the principles established in our theoretical analysis provide a robust blueprint for a system that is not only sound in its logic but also viable in practice. This strong theoretical grounding in efficiency and scalability sets the stage for the empirical validation of these performance characteristics in the subsequent sections.

4.2. Experiment setting

The experimental evaluations were conducted on a computing platform with the following specifications: CentOS Linux 7 (kernel version 3.10.0–123.el7.x86_64), SystemTap version 4.0/86, Intel Core i7-6800K CPU, and 16 GB DDR4 RAM.

Given that Log2Evt necessitates access to runtime program information, evaluation using pre-existing log datasets is not feasible. Instead, we utilized a custom log dataset, as detailed in Table 3, for our analysis. The log samples from the Linux Application Log Dataset were employed to benchmark our proposed approach against established baselines, as detailed in Table 4.

Log2Evt performs event sequence segmentation on log datasets, which, along with the baseline approaches, can be conceptualized as clustering methodologies. The effectiveness of these approaches is evaluated using standard clustering metrics: Purity, Rand Index, and Fowlkes–Mallows Index.

Purity is a straightforward and interpretable metric widely employed for evaluating clustering performance. This measure quantifies clustering efficacy by first identifying the most predominant class within each cluster, then summing the count of these dominant class members across all clusters, and finally normalizing by the total number of data points. Formally, given a set of clusters M and a set of classes D, both partitioning N data points, *Purity* can be mathematically expressed as: $Purity = \frac{1}{N}\sum_{m\in M}\max_{d\in D}|m\cap d|$.

The Rand Index (R) quantifies the similarity between two data clustering configurations. Consider a set $S = \{o_1, \ldots, o_n\}$ containing n elements, with two distinct partitions: $X = \{X_1, \ldots, X_r\}$ dividing S into r subsets, and $Y = \{Y_1, \ldots, Y_s\}$ dividing S into s subsets. The following definitions establish the foundation for calculating this index:

- True Positives (TP): Number of element pairs in S that are clustered together in both X and Y.
- True Negatives (TN): Number of element pairs in S that are separated in both X and Y.
- False Positives (FP): Number of element pairs in S that are clustered together in X but separated in Y.
- False Negatives (FN): Number of element pairs in S that are separated in X but clustered together in Y.

The Rand Index (R) is defined as: $R = (TP+TN)/(TP+FP+TN+FN)$,where $TP+TN$ represents the total number of concordant element pairs between configurations X and Y, while $FP+FN$ represents the discordant pairs.

The Fowlkes-Mallows Index (FM) is an external evaluation metric used to quantify the similarity between two clustering configurations or between a clustering and a benchmark classification. This index is particularly useful in assessing confusion matrices. A higher FM value indicates greater similarity between the compared clusterings or between the clustering and the benchmark. The FM index is defined as the geometric mean of precision and recall: $FM = \sqrt{PPV \cdot TPR} = \sqrt{\frac{TP}{TP+FP}\cdot\frac{TP}{TP+FN}}$, where TPR is the true positive rate, also called sensitivity or recall, and PPV is the positive predictive rate, also known as precision.

4.3. Functional experiments

The SSH protocol is widely used as a secure remote login protocol, and its experiments are representative and of high research value. In this section, the experiments evaluate the feasibility of Log2Evt and the impact of threshold RA on efficacy by analyzing successive SSH login sequences.

In the feasibility experiment, two logins were made to the host via the WinSCP software at 22:04:13 and 22:04:26. The log file recording

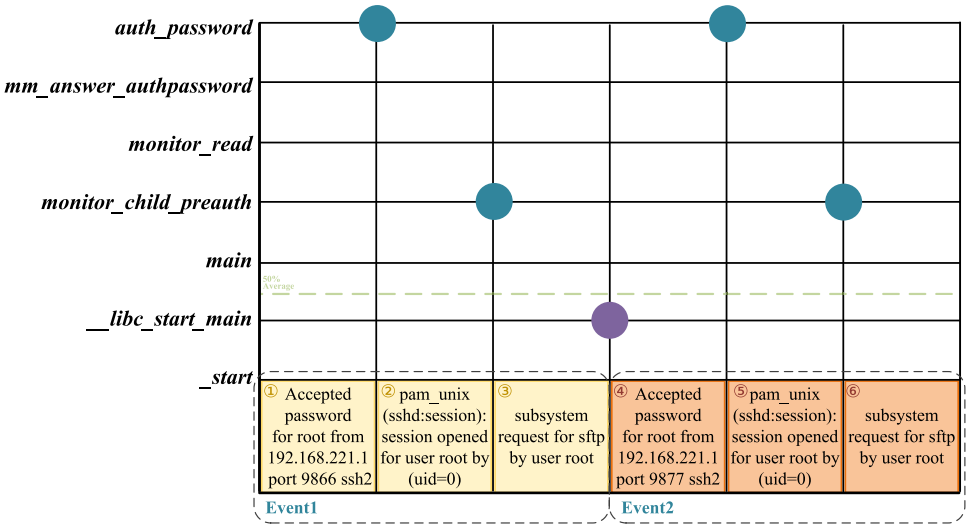


Fig. 11. The common ancestors of the target log set.

Table 4
Tested components and log sources.

Test components	Log path	Description
PAM (1.1.8)	*/var/log/secure*	User authentication library suite
Samba (4.10.16)	*/var/log/samba/*	SMB protocol implementation
Vsftpd (3.0.2)	*/var/log/vsftpd.log*	FTP server for Unix-like systems
Networkmanager (0.9.9.1)	*/var/log/message*	Network interface configuration daemon

Table 5
Logs of two successful logins.

Time	Content
22:04:13	Accepted password for root from 192.168.221.1 port 9866 ...
22:04:13	pam_unix(sshd:session): session opened for ...
22:04:13	subsystem request for sftp by user root
22:04:26	Accepted password for root from 192.168.221.1 port 9877 ...
22:04:26	pam_unix(sshd:session): session opened for ...
22:04:26	subsystem request for sftp by user root

the login behavior is located in */var/log/secure*, and the logs of the two successful logins are shown in Table 5.

SSH login behavior is done through PAM (Pluggable Authentication Modules), and PAM completes logging by calling the rsyslog logging framework. From the logging framework call relationship, it is necessary to monitor the **pam_vsyslog()** function located in *linux-pam-master/libpam/pam_syslog.c* during the log source location phase, and its function prototype is "**void pam_vsyslog (const pam_handle_t *pamh, int priority, const char *fmt, va_list args)**".

After converting the obtained set of program's call stacks into a tree structure by **Algorithm** 1, the sequence of function call traces corresponding to the logs is mapped into highlighted nodes. To better represent the program's operation, the structure is visualized using the flame graph.

The flame graph is a visual profiler that creates an interactive SVG that shows a collection of stacked traces. It has the following features:

- The stack is represented as a column of boxes, where each box represents a function (stack frame)
- The y-axis shows the stack depth from the root node at the bottom to the leaf node at the top, the top box represents the function on the CPU when the stack trace is collected, and the function below the function is its parent.
- The x-axis spans the stack collection, indicating the passage of time.

As shown in Fig. 10, the left and right sides have the same pattern, corresponding to the same login behavior twice, and the resulting logs are mapped to the same set of stack frames in both sets. This visualization shows that the program performs the same process twice and outputs the same log messages when performing two login operations.

Subsequently, after calculation of highlighted nodes by **Algorithm** 2, as shown in Fig. 11, the numbers of common ancestors between adjacent highlighted nodes on this tree is calculated, which are 7, 4, 2, 7, 4. The respective depths of these highlighted nodes are 1, 4, 6, 1, 4. Taking $RA = 0.35$, by **Algorithm** 3, the threshold is calculated to be 5.6, so it is decided that logs 3 and 4 do not belong to the same event.*Log messages* 1, 2, 3 are classified to event 1, and *log messages* 4, 5, 6 are classified to event 2. The output results are consistent with the operation flow and prove the approach's feasibility.

4.4. Efficacy experiments

In the execution path analysis conducted by Log2Evt, **Algorithm** 3 employs a critical threshold ratio, denoted as RA, for the purpose of event classification. To establish the classification criterion, we first identify β as the set of LCA nodes relevant to the log messages under consideration. A threshold value is then computed. This computation involves multiplying the ratio RA by the sum of the depths of all nodes M_i within the set β. This specific calculation is formally expressed in from equation: $Threshold = RA * \sum_{M_i \in \beta} depth(M_i)$.

The threshold ratio RA represents a critical trade-off. A low RA value imposes a stringent, or shallow, depth threshold, which causes the algorithm to partition log sequences more frequently. While this produces fine-grained event sequences, it also introduces the risk of over-segmentation, wherein a single logical operation is erroneously fragmented. Conversely, a high RA value establishes a lenient threshold, leading to the creation of broader, coarse-grained events. Although this approach provides a high-level overview, it risks under-segmentation, where distinct activities are improperly merged. As demonstrated in our experiments, the optimal RA value must be

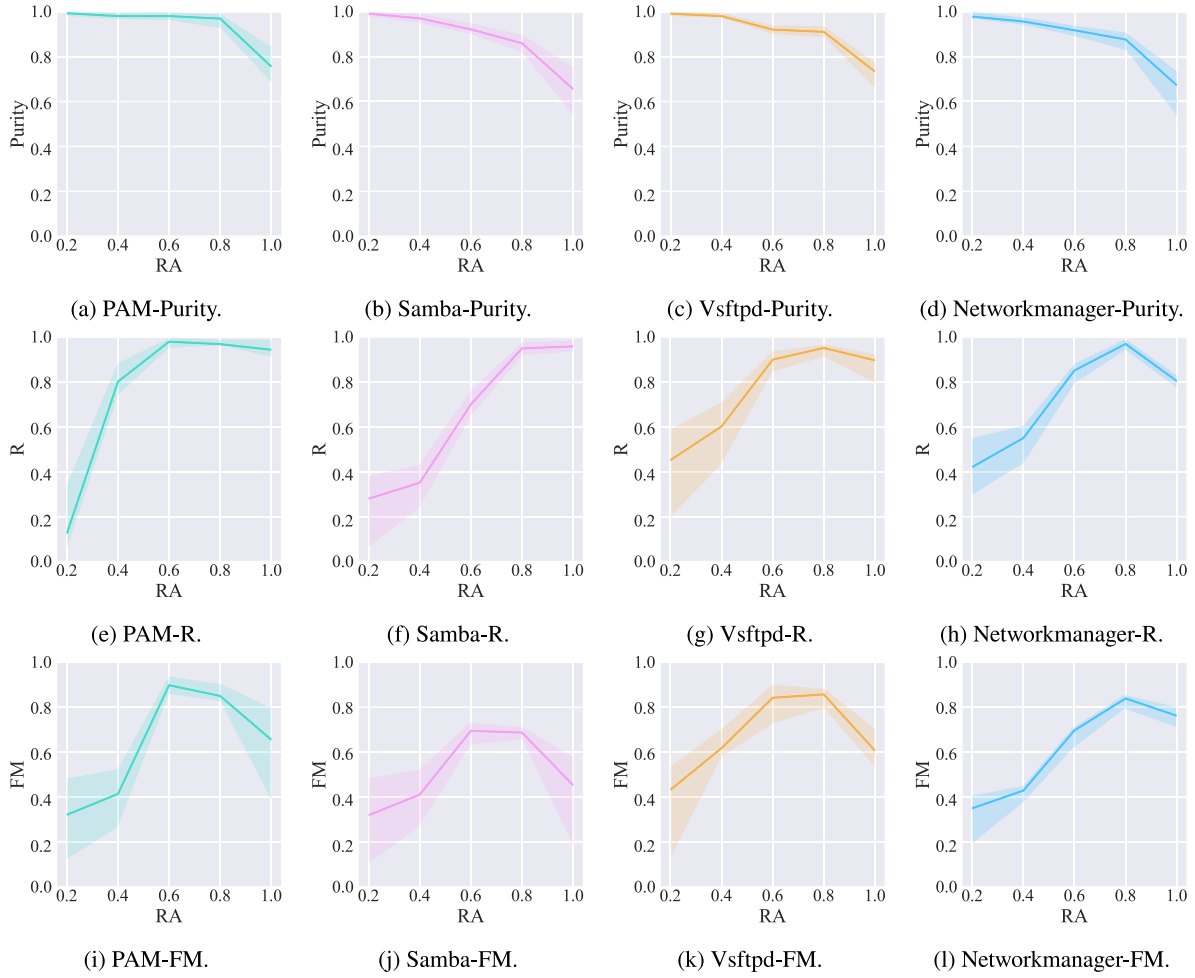


Fig. 12. Effect of RA on Log2Evt's performance across multiple log datasets and evaluation metrics.

selected to balance these extremes, aligning the event granularity with the application's inherent logical structure.

While the RA parameter governs the outcome, the system's performance is also dependent on the efficiency of the LCA computation. This represents a key design choice rather than a tunable hyperparameter. A naive, brute-force LCA calculation would render the system unscalable. Our deliberate selection of the Tarjan-LCA algorithm, a highly optimized offline method with near-linear time complexity, is foundational to our approach. This choice ensures that the event integration phase remains efficient enough for large-scale log analysis, a decision that prioritizes performance without altering the correctness of the final event classification.

In Fig. 11, $RA = 0.35$ is used as a sample, but in practice, the RA values for obtaining the best results are different for different log samples. As an important variable directly related to the threshold, the variation of RA can significantly affect the score.

In this experiment, the manually divided event sequence is used as the standard answer compared to the analysis results. This experiment compares the $Purity$, R and FM as the index with the following approaches: behavior pattern alignment-based matching approach represented by [29], and the rule-based matching approach represented by [28].

This experiment evaluates [28,29] and Log2Evt based on $Purity$, R and FM. The results are shown in Table 6. Log2Evt has a certain degree of improvement in the above metrics compared with the alignment-based matching approach [29] and rule-based matching approach [28].

In the threshold experiments, we evaluate the efficacy of Log2Evt at RA equals to 0.2, 0.4, 0.6, 0.8 and 1.0, respectively, by evaluating the criteria $Purity$, R, and FM based on this log sample. The results are shown in Fig. 12. As illustrated in the figure, all three metrics demonstrate notable sensitivity to variations in the RA threshold. Specifically, $Purity$ exhibits a gradual decline as RA increases beyond 0.4, while R shows relatively stable performance until RA reaches 0.6, after which it experiences a sharp decrease. The FM score, being a harmonic mean of precision and recall, follows a similar trend to $Purity$ but with more pronounced deterioration at higher RA values. From the experiment results, the choice of RA greatly impacts the performance of Log2Evt, and the best efficacy is achieved with RA around 0.4 for this log sample.

4.5. Time and space occupation experiments

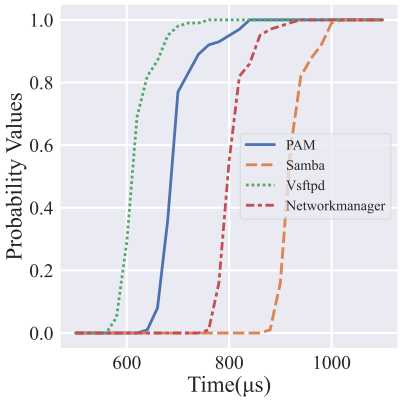
The time required to perform an operation is one of the metrics to measure the effectiveness of an algorithm. In addition, Log2Evt records all the program traces to the memory for analysis. So the space usage of Log2Evt needs to be tested to evaluate the feasibility of its use.

To evaluate the time consumption and space occupation of Log2Evt a long sequence of operations with repeatable and strictly consistent characteristics is applied to the log samples, and the results are averaged. These cases' time consumption and space occupation are evaluated using this approach following the actual application scenario.

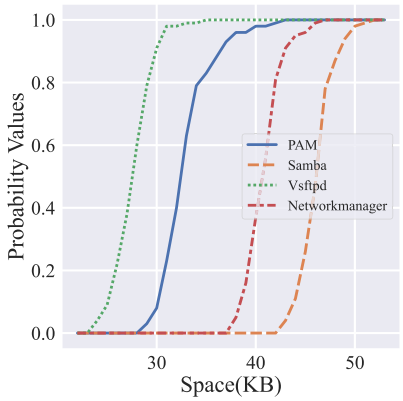
In this section of the experiment, the same operations are executed 1000 times for each application scenario, evaluating the time complexity and space usage by the algorithm run time and the size of the recorded program trace file. In order for multiple login attempts

Table 6
Efficacy experiment results.

Condition	PAM			Samba			Vsftpd			Networkmanager		
	Purity	*R*	*FM*	*Purity*	*R*	*FM*	*Purity*	*R*	*FM*	*Purity*	*R*	*FM*
Log2Evt	1.00	1.00	1.00	0.90	0.95	0.75	0.95	0.99	0.92	0.85	0.81	0.80
[29]	0.91	0.93	0.88	0.81	0.97	0.77	0.89	0.99	0.83	0.39	0.55	0.36
[28]	0.86	0.91	0.85	0.74	0.95	0.65	0.87	0.99	0.78	0.34	0.54	0.33



(a) Consumption of time.



(b) Consumption of space.

Fig. 13. CDFs of time and space consumption for a single event.

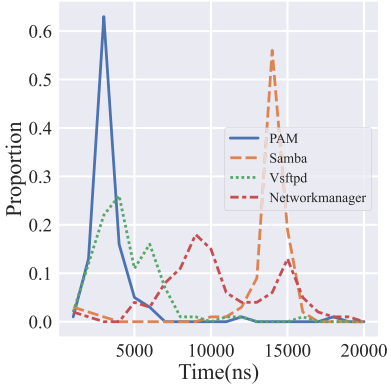


Fig. 14. Time to record call stacks.

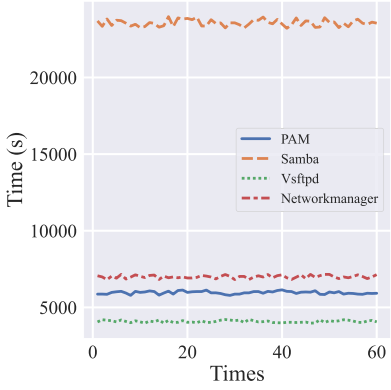


Fig. 15. Time to configure probes.

to not be blocked, for PAM, the *MaxAuthTries* configuration file in */etc/ssh/sshd_config* needs to be changed to 1000.

The experimental results are shown in Fig. 13. For all test samples, the average space occupied by the logged data for each operation is between 25 KB and 50 KB, and the algorithm execution time is between 500 μs and 1000 μs.

At the same time, since Log2Evt needs to configure the sequence of probes in the target application, it needs to consider the time to configure the sequence in the program and the impact of probes on the program execution performance.

For the evaluation of the time to record program's call stacks, we place timers in each probe and print out the CPU time taken by each probe in the logged data. As shown in Fig. 14, the time taken for each execution of the code in the probe is mainly distributed between 2500 ns and 17 000 ns, with PAM taking the least time and Samba taking the most time, which is related to the length of the printed stack.

For evaluating the time to configure the probe, we place a timer at the beginning of the bash script and the beginning of the SystemTap to count the time from executing the script to configuring all probes. As shown in Fig. 15, the time used to configure probes is mainly distributed between 3000 ms and 25 000 ms, among which Vsftpd takes the least time and Samba takes the most time, which is related to the number of functions of the program.

Based on the experimental results presented in Fig. 16, we conducted a systematic analysis of temporal and spatial characteristics across four application scenarios (PAM, Samba, Vsftpd, and Networkmanager) with controlled operational repetitions (1000–4000 executions). The boxplots reveal statistically significant performance distributions across two primary dimensions: temporal performance and spatial efficiency.

The algorithm demonstrates sub-millisecond latency across all scenarios, with interquartile ranges indicating stable temporal predictability. PAM exhibits the most consistent temporal distribution, suggesting optimized performance for authentication operations. Conversely, Samba displays wider dispersion, potentially due to its complex file-sharing protocol stack. Notably, temporal scaling remains sublinear as operational repetitions increase from 1000 to 4000, confirming the algorithm's efficient time complexity.

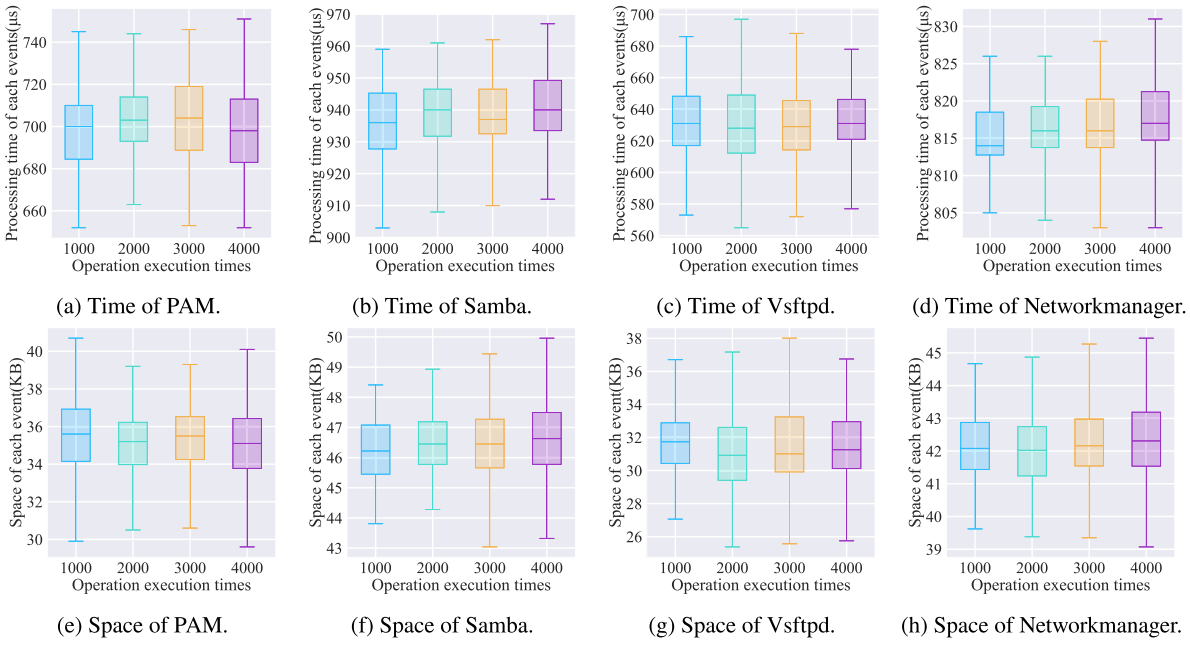


Fig. 16. Boxplots of time and space consumption for events with different times.

Memory consumption per operation remains constrained within 25–50 KB, exhibiting near-constant growth characteristics. Networkmanager demonstrates the most stable allocation pattern, while Vsftpd shows moderate variance. The 95th percentile of spatial consumption never exceeds 50 KB, even under maximum operational load, validating the design's memory-efficient architecture.

5. Discussion

The integration of execution path tracing with IoT log analysis, as proposed in Log2Evt, introduces a novel paradigm for enhancing observability in resource-constrained smart systems. By correlating firmware runtime context with device logs, this approach bridges the gap between fragmented sensor telemetry and actionable security insights. While evaluations demonstrate efficacy in lab-based IoT testbeds, scalability challenges and IoT-specific deployment barriers require deeper scrutiny.

5.1. Internal validity in IoT contexts

IoT hardware and RTOS heterogeneity: The reliance on dynamic tracing tools like eBPF introduces dependencies on specific microcontroller architectures and real-time operating systems . Proprietary firmware in industrial IoT devices or locked-down consumer gadgets may block runtime instrumentation, limiting real-time path tracing. For example, low-power LoRaWAN sensors with stripped debugging interfaces cannot capture call stack traces during energy-saving sleep modes.

Concurrency limitations in distributed IoT workflows: The method assumes execution context coherence within single-device firmware. However, IoT operations like multi-sensor data fusion or edge-cloud synchronization involve asynchronous, distributed workflows. In such cases, call stacks from a Raspberry Pi edge node may fail to propagate context to associated LoRaWAN gateways, causing log correlation gaps during cross-device attacks.

Resilience to Log Noise and Adversarial Manipulation: Log2Evt demonstrates inherent robustness against common forms of log noise and adversarial manipulation. Its resilience stems from the core principle of correlating logs with their actual, dynamically captured code execution paths. Intentionally injected log noise, if not generated by a legitimate and instrumented code path, will fail to correlate and is thus naturally filtered out. Forging logs with misleading call stacks presents a more sophisticated challenge, but this would require an attacker to gain sufficient privileges to manipulate the low-level tracing mechanism itself, at which point the system is likely already fully compromised. While a high volume of noise could degrade performance before being filtered, the accuracy of event reconstruction remains high against attacks that cannot control the underlying execution environment.

5.2. External validity in IoT deployments

Resource constraints and performance balance: Despite efficiency optimizations, implementing full execution path tracking on embedded microcontrollers may significantly increase computational load, threatening real-time guarantees in industrial control systems. For battery-powered IoT nodes, the trade-off between tracking granularity and energy consumption becomes an unavoidable design contradiction, requiring dynamic balance between data accuracy and device longevity.

Real-world IoT log complexity: Evaluations used sanitized logs from homogeneous device fleets, but operational IoT environments mix structured and unstructured logs. Environmental interference and third-party black-box components introduce noise unaccounted for in current models, potentially masking stealthy attacks like firmware downgrade exploits.

Robustness to data obfuscation: The validity of the Log2Evt framework extends to environments with encrypted logs or restricted tracing because its core mechanism relies on the structural fingerprint of the code execution path, not solely on log content. While this approach preserves the ability to reconstruct a skeletal event narrative, it inherently faces limitations, namely a loss of semantic richness from unreadable messages and potential ambiguity if the execution trace itself is incomplete.

Applicability to Real-Time Detection: The Log2Evt framework is conceptually adaptable to runtime anomaly detection because its core data collection method, which relies on dynamic kernel instrumentation, captures execution paths in real time. This provides the necessary live data stream for on-the-fly analysis. However, the computational expense of the full event reconstruction and relational analysis pipeline

introduces a significant latency trade-off. In its current form, this overhead makes the complete process better suited for deep postmortem forensics rather than immediate, low-latency threat detection.

5.3. Future work

Our future research will advance the Log2Evt framework by focusing on three interconnected areas: enhancing its utility in privacy-restricted settings, strengthening its resilience against sophisticated attacks, and adapting it for real-time threat detection.

First, we will address the challenge of analysis in privacy-restricted environments by developing a more resilient hybrid framework. We plan to enhance Log2Evt by fusing the available structural data with other system telemetry, such as network flows, and employing probabilistic models to handle ambiguity. Furthermore, we will investigate the application of advanced privacy-preserving technologies, including homomorphic encryption, to enable direct analysis of encrypted logs and traces, thereby maintaining data confidentiality throughout the event reconstruction process.

Building on this, to further harden Log2Evt against sophisticated adversarial attacks, our future work will focus on developing a robust integrity validation framework. We plan to move beyond simple correlation by implementing anomaly detection directly on the execution traces themselves, building models of legitimate call stack patterns to identify forged or unusual paths. This will be complemented by a cross-validation system that correlates events constructed by Log2Evt with data from independent sources, such as network intrusion detection systems, to verify their authenticity. This multi-layered approach will enable the system to not only detect but also actively flag and isolate sophisticated adversarial manipulations.

Finally, to make these enhanced security capabilities practical for immediate threat response, we will focus on adapting Log2Evt for real-time anomaly detection by designing a tiered analysis framework. This involves developing a lightweight, real-time triage component for instant threat flagging based on execution path signatures, complemented by a near-real-time deep analysis layer for more complex event reconstruction. We plan to build this system using stream processing engines for efficient data handling and will integrate online machine learning algorithms to dynamically model normal system behavior and accurately detect deviations.

By addressing these IoT-centric limitations — through adaptive tracing for heterogeneous hardware and noise-resilient correlation algorithms — future work could enable robust intrusion detection across smart cities, healthcare IoT, and Industry 4.0 deployments.

6. Conclusion

Cyberspace is increasingly targeted by stealthy attacks, particularly in IoT systems where adversaries exploit vulnerabilities in interconnected smart devices. To address the forensic challenges posed by massive, fragmented logs from distributed sensors, gateways, and edge nodes, we propose Log2Evt, which can construct high-level events from low-level log messages, helping users locating attack-related log entries more quickly and accurately.

Log2Evt is shown to have higher efficacy in specifying test cases compared to the matching-based representation approaches. However, depending on the design of the program, the classification approach based on the maximum number of common ancestors has the theoretical possibility of outputting low efficacy results. Due to the debugging mechanism, Log2Evt is only applicable to programs with DebugInfo. For future work, we will improve the data structure in the Common Ancestor algorithm, for example, using a hashing algorithm to increase computational efficiency and robustness in the face of large-scale systems. In addition, inspired by the excellent performance of Kprobes and the SystemTap framework, we will investigate its application in the field of network security in more depth.

CRediT authorship contribution statement

Teng Li: Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Baichuan Zheng:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Yebo Feng:** Writing – review & editing, Methodology, Conceptualization. **Xiaowen Quan:** Conceptualization. **Jiahua Xu:** Writing – review & editing, Supervision, Methodology. **Yang Liu:** Conceptualization. **Jianfeng Ma:** Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is funded by the National Key Research and Development Program of China (2023YFB2904000), Natural Science Basic Research Program of Shaanxi (No. 2025JC-JCQN-073), National Natural Science Foundation of China under Grant (No. 62272370), Young Elite Scientists Sponsorship Program by CAST (2022QNRC001), the China 111Project (No. B16037), Qinchuangyuan Scientist + Engineer Team Program of Shaanxi (No. 2024QCY-KXJ-149), Songshan Laboratory (No. 241110210200), Open Foundation of Key Laboratory of Cyberspace Security, Ministry of Education of China (No. KLCS20240405) and the Fundamental Research Funds for the Central Universities, China (QTZX23071), the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008), the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme, and Ripple under its University Blockchain Research Initiative (UBRI) [54].

References

[1] Wei Qiao, Yebo Feng, Teng Li, Zhuo Ma, Yulong Shen, JianFeng Ma, Yang Liu, Slot: Provenance-driven APT detection through graph reinforcement learning, 2024, arXiv preprint arXiv:2410.17910.

[2] Shahbaz Siddiqui, Sufian Hameed, Syed Attique Shah, Abdul Kareem Khan, Adel Aneiba, Smart contract-based security architecture for collaborative services in municipal smart cities, J. Syst. Archit. 135 (2023) 102802, URL https://www.sciencedirect.com/science/article/pii/S1383762122002879.

[3] Hongtao Yu, Suhui Liu, Liquan Chen, Yuan Gao, Blockchain-enabled one-to-many searchable encryption supporting designated server and multi-keywords for Cloud-IoMT, J. Syst. Archit. 149 (2024) 103103, URL https://www.sciencedirect.com/science/article/pii/S1383762124000407.

[4] Panjun Sun, Yi Wan, Zongda Wu, Zhaoxi Fang, Qi Li, A survey on privacy and security issues in IoT-based environments: Technologies, protection measures and future directions, Comput. Secur. 148 (2025) 104097.

[5] Tao Zhang, Fanyu Kong, Dongshang Deng, Xiangyun Tang, Xuangou Wu, Changqiao Xu, Liehuang Zhu, Jiqiang Liu, Bo Ai, Zhu Han, et al., Moving target defense meets artificial intelligence-driven network: A comprehensive survey, IEEE Internet Things J. (2025).

[6] Yangzixing Lv, Wei Shi, Weiyong Zhang, Hui Lu, Zhihong Tian, Do not trust the clouds easily: The insecurity of content security policy based on object storage, IEEE Internet Things J. 10 (12) (2023) 10462–10470.

[7] Yukyung Lee, Jina Kim, Pilsung Kang, Lanobert: System log anomaly detection based on bert masked language model, Appl. Soft Comput. 146 (2023) 110689.

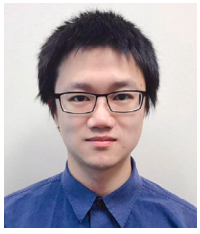
[8] Lin Yang, Junjie Chen, Shutao Gao, Zhihao Gong, Hongyu Zhang, Yue Kang, Huaan Li, Try with simpler-an evaluation of improved principal component analysis in log-based anomaly detection, ACM Trans. Softw. Eng. Methodol. 33 (5) (2024) 1–27.

[9] Zijun Cheng, Qiujian Lv, Jinyuan Liang, Yan Wang, Degang Sun, Thomas Pasquier, Xueyuan Han, Kairos: Practical intrusion detection and investigation using whole-system provenance, in: 2024 IEEE Symposium on Security and Privacy, SP, IEEE, 2024, pp. 3533–3551.

[10] Qingqing Xie, Fatong Zhu, Xia Feng, Blockchain-enabled data sharing for IoT: A lightweight, secure and searchable scheme, J. Syst. Archit. 154 (2024) 103230, URL https://www.sciencedirect.com/science/article/pii/S138376212400167X.
[11] Kexiong Fei, Jiang Zhou, Yucan Zhou, Xiaoyan Gu, Haihui Fan, Bo Li, Weiping Wang, Yong Chen, LaAeb: A comprehensive log-text analysis based approach for insider threat detection, Comput. Secur. 148 (2025) 104126.
[12] Yiren Chen, Mengjiao Cui, Ding Wang, Yiyang Cao, Peian Yang, Bo Jiang, Zhigang Lu, Baoxu Liu, A survey of large language models for cyber threat detection, Comput. Secur. (2024) 104016.
[13] Guojun Chu, Jingyu Wang, Qi Qi, Haifeng Sun, Zirui Zhuang, Bo He, Yuhan Jing, Lei Zhang, Jianxin Liao, Anomaly detection on interleaved log data with semantic association mining on log-entity graph, IEEE Trans. Softw. Eng. (2025).
[14] Teng Li, Shengkai Zhang, Yebo Feng, Jiahua Xu, Zhuo Ma, Yulong Shen, Jianfeng Ma, Heuristic-based parsing system for big data log, in: GLOBECOM 2024 - 2024 IEEE Global Communications Conference, 2024, pp. 2329–2334.
[15] Ilja Behnke, Christoph Blumschein, Robert Danicki, Philipp Wiesner, Lauritz Thamsen, Odej Kao, Towards a real-time IoT: Approaches for incoming packet processing in cyber–physical systems, J. Syst. Archit. 140 (2023) 102891, URL https://www.sciencedirect.com/science/article/pii/S138376212300070X.
[16] Chen Zhi, Liye Cheng, Meilin Liu, Xinkui Zhao, Yueshen Xu, Shuiguang Deng, LLM-powered zero-shot online log parsing, in: 2024 IEEE International Conference on Web Services, ICWS, IEEE, 2024, pp. 877–887.
[17] Aoxiao Zhong, Dengyao Mo, Guiyang Liu, Jinbu Liu, Qingda Lu, Qi Zhou, Jiesheng Wu, Quanzheng Li, Qingsong Wen, Logparser-llm: Advancing efficient log parsing with large language models, in: Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2024, pp. 4559–4570.
[18] Daniel Schuster, Francesca Zerbato, Sebastiaan J van Zelst, Wil MP van der Aalst, Defining and visualizing process execution variants from partially ordered event data, Inform. Sci. 657 (2024) 119958.
[19] Tianzhu Zhang, Han Qiu, Gabriele Castellano, Myriana Rifai, Chung Shue Chen, Fabio Pianese, System log parsing: A survey, IEEE Trans. Knowl. Data Eng. 35 (8) (2023) 8596–8614.
[20] Haitian Yang, Degang Sun, Yan Wang, Weiqing Huang, DSGN: Log-based anomaly diagnosis with dynamic semantic gate networks, Inform. Sci. 680 (2024) 121174.
[21] Hui Lu, Chengjie Jin, Xiaohan Helu, Xiaojiang Du, Mohsen Guizani, Zhihong Tian, DeepAutoD: Research on distributed machine learning oriented scalable mobile communication security unpacking system, IEEE Trans. Netw. Sci. Eng. 9 (4) (2022) 2052–2065.
[22] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, Michael R Lyu, A large-scale evaluation for log parsing techniques: How far are we? in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 223–234.
[23] Maria Laura Sebu, Horia Ciocarlie, Applied process mining in software development, in: 2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics, SACI, IEEE, 2014, pp. 55–60.
[24] Boxi Yu, Jiayi Yao, Qiuai Fu, Zhiqing Zhong, Haotian Xie, Yaoliang Wu, Yuchi Ma, Pinjia He, Deep learning or classical machine learning? An empirical study on log-based anomaly detection, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.
[25] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, Wenbin Zhang, Semi-supervised log-based anomaly detection via probabilistic label estimation, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, IEEE, 2021, pp. 1448–1460.
[26] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al., UniLog: Automatic logging via LLM and in-context learning, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–12.
[27] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, Zhongzhi Luan, Hitanomaly: Hierarchical transformers for anomaly detection in system log, IEEE Trans. Netw. Serv. Manag. 17 (4) (2020) 2064–2076.
[28] Saad Khan, Simon Parkinson, Discovering and utilising expert knowledge from security event logs, J. Inf. Secur. Appl. 48 (2019) 102375.
[29] Cong Liu, Shi Wang, Shangce Gao, Feng Zhang, Jiujun Cheng, User behavior discovery from low-level software execution log, IEEJ Trans. Electr. Electron. Eng. 13 (11) (2018) 1624–1632.
[30] Edyta Brzychczy, Milda Aleknonytė-Resch, Dominik Janssen, Agnes Koschmider, Process mining on sensor data: a review of related works, Knowl. Inf. Syst. (2025) 1–34.
[31] Octavio Loyola-González, Process mining: software comparison, trends, and challenges, Int. J. Data Sci. Anal. 15 (4) (2023) 407–420.
[32] Shameer K. Pradhan, Mieke Jans, Niels Martin, Getting the data in shape for your process mining analysis: An in-depth analysis of the pre-analysis stage, ACM Comput. Surv. (2025).
[33] Guanjun Liu, Petri Nets: Theoretical Models and Analysis Methods for Concurrent Systems, Springer Nature, 2022.
[34] Anna A Kalenkova, Wil MP van der Aalst, Irina A Lomazova, Vladimir A Rubin, Process mining using BPMN: relating event logs and process models, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, 2016, pp. 123–123.
[35] Ezequiel O. Ramos, Rogério Rossi, Process mining applied in a software project development with SCRUM and prom, Eur. J. Eng. Technol. Res. 8 (5) (2023) 17–24.
[36] Michela Vespa, Elena Bellodi, Federico Chesani, Daniela Loreti, Paola Mello, Evelina Lamma, Anna Ciampolini, Marco Gavanelli, Riccardo Zese, Probabilistic traces in declarative process mining, in: International Conference of the Italian Association for Artificial Intelligence, Springer, 2024, pp. 330–345.
[37] Yintong Huo, Yuxin Su, Cheryl Lee, Michael R. Lyu, SemParser: A semantic parser for log analytics, in: 2023 IEEE/ACM 45th International Conference on Software Engineering, ICSE, IEEE, 2023, pp. 881–893.
[38] Weibin Meng, Federico Zaiter, Yuzhe Zhang, Ying Liu, Shenglin Zhang, Shimin Tao, Yichen Zhu, Tao Han, Yongpeng Zhao, En Wang, Yuzhi Zhang, Dan Pei, LogSummary: Unstructured log summarization for software systems, IEEE Trans. Netw. Serv. Manag. 20 (3) (2023) 3803–3815.
[39] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, Michael R Lyu, Lilac: Log parsing using llms with adaptive parsing cache, Proc. the ACM Softw. Eng. 1 (FSE) (2024) 137–160.
[40] Marco Pegoraro, Bianka Bakullari, Merih Seran Uysal, Wil MP van der Aalst, Probability estimation of uncertain process trace realizations, in: International Conference on Process Mining, Springer, Cham, 2022, pp. 21–33.
[41] Aleksei Pismerov, Maxim Pikalov, Applying embedding methods to process mining, in: Proceedings of the 2022 5th International Conference on Algorithms, Computing and Artificial Intelligence, 2022, pp. 1–5.
[42] Siyu Yu, Yifan Wu, Zhijing Li, Pinjia He, Ningjiang Chen, Changjian Liu, Log parsing with generalization ability under new log types, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 425–437.
[43] Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, Pengfei Chen, Logshrink: Effective log compression by leveraging commonality and variability of log data, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–12.
[44] Siyu Yu, Pinjia He, Ningjiang Chen, Yifan Wu, Brain: Log parsing with bidirectional parallel tree, IEEE Trans. Serv. Comput. 16 (5) (2023) 3224–3237.
[45] Yintong Huo, Yichen Li, Yuxin Su, Pinjia He, Zifan Xie, Michael R Lyu, Autolog: A log sequence synthesis framework for anomaly detection, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2023, pp. 497–509.
[46] Yilun Liu, Shimin Tao, Weibin Meng, Jingyu Wang, Wenbing Ma, Yuhang Chen, Yanqing Zhao, Hao Yang, Yanfei Jiang, Interpretable online log analysis using large language models with prompt strategies, in: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, 2024, pp. 35–46.
[47] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, Pinjia He, DivLog: Log parsing with prompt enhanced in-context learning, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–12.
[48] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, Shaowei Wang, Llmparser: An exploratory study on using large language models for log parsing, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
[49] Jim Keniston, Kernel probes, 2022, https://docs.kernel.org/trace/kprobes.html. (Accessed January 2025).
[50] Frank Ch. Eigler, Systemtap, 2005, https://sourceware.org/systemtap/. (Accessed January 2025).
[51] Adel Belkhiri, Martin Pepin, Mike Bly, Michel Dagenais, Performance analysis of DPDK-based applications through tracing, J. Parallel Distrib. Comput. 173 (2023) 1–19.
[52] Brendan Gregg, The flame graph, Commun. ACM 59 (6) (2016) 48–57.
[53] Shilin He, Jieming Zhu, Pinjia He, Michael R. Lyu, Loghub: a large collection of system log datasets towards automated log analytics, 2020, arXiv preprint arXiv:2008.06448.
[54] Yebo Feng, Jiahua Xu, Lauren Weymouth, University blockchain research initiative (UBRI): Boosting blockchain education and research, IEEE Potentials 41 (6) (2022) 19–25.

Teng Li received the B.S. degree in School of Computer Science and Technology from Xidian University, China in 2013, and Ph.D. degree in School of Computer Science and Technology from Xidian University, China in 2018. He is currently a Professor at the School of Cyber Engineering, Xidian University, China. His current research interests include wireless and networks, distributed systems and intelligent terminals with focus on security and privacy issues.

Baichuan Zheng received his B.S. degree in School of Cyber Engineering from Xidian University, China in 2024, where he is pursuing the M.S. degree. His current research interests include IoT security, log analysis, network security, and federated learning.



Yebo Feng is a research fellow in the College of Computing and Data Science (CCDS) at Nanyang Technological University (NTU). He received his Ph.D. degree in Computer Science from the University of Oregon (UO) in 2023. His research interests include network security, blockchain security, and anomaly detection. He is the recipient of the Best Paper Award of 2019 IEEE CNS, Gurdeep Pall Graduate Student Fellowship of UO, and Ripple Research Fellowship. He has served as the reviewer of IEEE TDSC, IEEE TIFS, ACM TKDD, IEEE JSAC, IEEE COMST, etc. Furthermore, he has been a member of the program committees for international conferences including SDM, CIKM, and CYBER, and has also served on the Artifact Evaluation (AE) committees for USENIX OSDI and USENIX ATC.

Xiaowen Quan received the M.S. degree in software engineering from Tsinghua University. His research interests include application security and network measurement.

Jiahua Xu is Associate Professor in Financial Computing, and Programme Director of the MSc Emerging Digital Technologies at UCL. She is also affiliated to the UCL Centre for Blockchain Technologies. Her research focuses on blockchain economics and decentralized finance. She has published in Usenix Security, ACM IMC, FC, IEEE ICDCS and IEEE ICBC. She has reviewed for Advances in Complex Systems, Computer Networks, Transactions on the Web and Cities.

Yang Liu received the B.S. degree in computer science and technology from Xidian University, in 2017. He is now an associate professor at school of Cyber Engineering, Xidian University. His research interests cover formal analysis of authentication protocols and deep learning neural network in cyber security.

Jianfeng Ma received the B.S. degree in computer science from Shaanxi Normal University in 1982, and M. S. degree in computer science from Xidian University in 1992, and the Ph. D. degree in computer science from Xidian University in 1995. Currently he is the directer of Department of Cyber engineering and a professor in School of Cyber Engineering, Xidian University. He has published over 150 journal and conference papers. His research interests include information security, cryptography, and network security.