PHOENIX: Protocol State Machine Recovery from Events in 5G Logs

Qingyuan Zheng[1], Ushasi Ghosh[1], Ali Mamaghani[1], Srinivas Shakkottai[2], and Dinesh Bharadia[1]
[1]University of California San Diego, CA, USA, [2]Texas A&M University, TX, USA

Abstract

Modern multi-vendor cellular systems generate massive volumes of debug logs, while limited source code access makes protocol behavior modeling and fault diagnosis challenging. Traditional KPI-based analysis fails to capture protocol semantics, and existing log-driven inference approaches either suffer from high computational overhead or rely on oversimplified prior knowledge. We propose PHOENIX, an automated framework that reconstructs component-level finite state machines (FSMs) directly from raw network logs. PHOENIX constructs causal graphs from event sequences, applies statistical pruning to suppress noise, and iteratively refines FSMs using acceptance thresholds to achieve high coverage and precision. Experiments on large-scale srsRAN debug logs demonstrate scalability to gigabyte–terabyte datasets and show that PHOENIX produces interpretable FSMs that enable visualization and facilitate causal reasoning for complex 5G protocol analysis.[1]

CCS Concepts

• **Networks** → **Network measurement**.

Keywords

Log Analysis, Finite State Machine Inference, 5G Protocols

ACM Reference Format:
Qingyuan Zheng[1], Ushasi Ghosh[1], Ali Mamaghani[1], Srinivas Shakkottai[2], and Dinesh Bharadia[1]. 2025. PHOENIX: Protocol State Machine Recovery from Events in 5G Logs. In *Proceedings of The 16th ACM Wireless of the Students, by the Students, and for the Students Workshop (Mobicom '25)*. ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern network systems generate massive volumes of log data and signaling messages during operation. These logs not only contain sufficient detail to infer protocol implementation specifics, but are also indispensable for key functions such as health monitoring and the time-consuming task of root cause failure analysis. However, in multi-vendor network architectures, understanding and evaluating these logs and message flows is highly challenging, as source code is typically unavailable and documentation rarely provides adequate protocol details. Consequently, analyzing logs has become one of the most practical means of understanding protocol realization. However, understanding the operational dynamics of 5G systems requires more than simply observing raw logs; it demands uncovering the hidden, protocol-driven state transitions that govern network behavior.

As illustrated in **??**, the essence of a communication system is defined by the state machines specified in its protocols. To reveal the concrete implementation of these protocols, this work proposes an automated pipeline that infers finite state machines (FSMs) from system operation logs, thereby reconstructing the underlying protocol logic defined in 3GPP standards. By making these latent state transitions explicit, the approach not only enhances the interpretability of system behavior but also enables downstream tasks such as performance diagnosis, fault localization, and adaptive optimization.

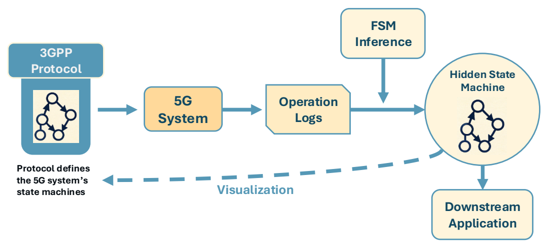

Figure 1: From 3GPP Protocols to Hidden State Machines: A Log-Driven Inference Paradigm.

Research on inferring FSMs from execution logs has been active for over a decade. MINT [6] effectively extracts system-level FSMs from small-scale logs, particularly in distributed systems such as HDFS. SCALER [4] extends MINT to component-level inference and heuristically merges these models into system-level FSMs, improving scalability and handling datasets of around 30,000 log entries. However, these methods struggle with the gigabyte- or terabyte-scale logs of communication systems. AutoModel [1] synthesizes FSMs from SoC communication traces by extracting event triplets (source, destination, message) from structured protocol interactions, segmenting them into independent threads via attributes such as module roles and transaction IDs, and enabling precise modular FSM construction. Validated on large-scale SoC datasets, it is conceptually applicable to communication networks with similar concurrency patterns. We adapt and extend this method for FSM modeling from 5G log data.

We propose **PHOENIX**, a fully automated pipeline capable of deriving component-level finite state machines directly from raw network logs. PHOENIX can be applied to any logging system that meets the following two conditions: (1) each log entry contains an explicit network component tag (e.g., RLC, PDCP, MAC); and (2) each log entry contains a user equipment identifier (e.g., UE ID, RNTI).

[1]Our code is in github link: https://github.com/ucsdwcsng/PHOENIX.git

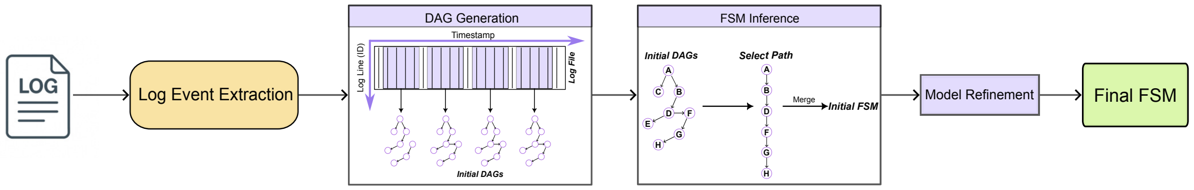

Figure 2: PHOENIX Overview.

To evaluate the effectiveness of PHOENIX, we analyzed logs from the widely used 5G over-the-air simulation platform SRSRAN. The results demonstrate that PHOENIX is a non-trivial log visualization tool that can support log-based root cause analysis and downstream LLM reasoning tasks.

Related Work

FSM-based inference methods have been widely adopted for protocol reimplementation. Tu et al. proposed 5GBaseChecker, which employs black-box automata learning to model the signaling messages of 5G baseband control-plane protocol implementations as FSMs[5]. It then identifies input sequences that trigger divergent outputs in the learned FSMs and uses these deviations to efficiently locate relevant security properties defined in the protocol specifications.

Another line of work leverages protocol specification documents directly to infer the state machines they define. Hermes[3], for instance, introduces a neural constituency parser and a protocol-oriented domain-specific language to extract transition components from specification texts, translate them into logical formulas, and compile them into FSMs, thereby structurally and precisely modeling protocol flows and enabling the automated formalization of cellular protocol specifications.

2 PHOENIX Design

The PHOENIX pipeline comprises two main modules: the **Log Event Extraction Module**, and the **FSM Inference Module**. **Log Event Extraction Module** leverages state-of-the-art log parsers to infer log templates and identify the complete set of atomic events. During parsing, PHOENIX first groups log entries by component tag and then sequences events from the same session or device using the UE identifier, ultimately producing a time-ordered series of structured events for state machine inference.**FSM Inference Module** adopts AutoModel's causal inference paradigm in an unsupervised, passive manner to generate per-component finite state models.

Log Event Extraction for FSM. In log-based causal inference, a "log event" is a structured abstraction of raw messages, consisting of a template that captures the fixed textual pattern (e.g., "UE attach request received"), a timestamp for temporal ordering, and parameter values extracted from variable segments (e.g., UE ID, sequence number, error code). Representing logs as *type + time + attributes* provides a consistent and analyzable input for causal inference.

To obtain such structured events, log parsing converts unstructured messages into templates by tokenizing text, normalizing variable elements (e.g., IP addresses, IDs), separating constant and variable components, and clustering similar structures based on token sequence or pattern similarity for large-scale aggregation. In this work, we adopt **Drain** [2] as the primary parsing method. Drain organizes templates into a fixed-depth parse tree: messages are routed by length, filtered by leading tokens, and matched at leaf nodes—updating matches with wildcards or creating new templates when unmatched. This one-pass streaming process enables high-throughput log parsing without offline training.

Once logs are structured, they can be naturally modeled as sequences of discrete events, which aligns with the formalism of an FSM. An FSM models the behavior of a discrete event system and, in its deterministic form, is defined as a tuple:$M = (Q, \Sigma, \delta, q_0)$, where Q is the finite set of states, Σ is the finite event alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, and $q_0 \in Q$ is the initial state. Given an event sequence $\sigma = e_1 e_2 \cdots e_n \in \Sigma^*$, the FSM generates a unique state path: $q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} q_n$, with $q_i = \delta(q_{i-1}, e_i)$.

In log analysis, each parsed log line corresponds to an event $e_i \in \Sigma$ with an associated timestamp, forming a trace σ that reflects the hidden execution paths of the underlying FSM. FSM inference from logs thus aims to recover the observable event set Σ, the hidden state set Q, the initial state q_0, and the transition relation δ.

FSM Inference. The FSM Inference Module reconstructs protocol behavior from raw log data $\mathcal{L}$ through a sequence of structured steps, fully parameterized by the confidence threshold θ, the acceptance threshold τ, and the maximum number of iterations T.

Directed Acyclic Graph (DAG) Construction: We first parse the raw log stream $\mathcal{L}$ into an ordered sequence of events, assigning each unique event a distinct identifier. For each event node s, we compute its **NodeSupport** $N(s)$, defined as the total number of occurrences of s in $\mathcal{L}$. For each ordered pair of adjacent events (a, b), we calculate the **EdgeSupport** $E(a, b)$, representing the number of times a is immediately followed by b.Two directional confidence measures are defined as $\mathrm{FC}(a, b) = E(a, b)/N(a), \mathrm{BC}(a, b) = E(a, b)/N(b)$, where $\mathrm{FC}(a, b)$ denotes the probability of b following a, and $\mathrm{BC}(a, b)$ denotes the probability of a preceding b. Using the set of nodes V and all observed edges, we construct a directed acyclic graph (DAG) $G = (V, E)$.

Graph Pruning: To eliminate noisy or weakly supported transitions, we compute for each edge $(a, b) \in E$ the average confidence score: $\mathrm{Conf}_{\mathrm{avg}}(a, b)$. If $\mathrm{Conf}_{\mathrm{avg}}(a, b) < \theta$, the edge is removed from G. This process retains only transitions with strong bidirectional statistical support, improving clarity and reducing complexity for subsequent FSM extraction.

Initial FSM Extraction: From each connected component of the pruned DAG, we extract the longest node-disjoint path P_i to maximize coverage of distinct nodes. These paths are concatenated, in logical or temporal order, to initialize the FSM $M = (Q, \Sigma, \delta, q_0)$, where Q is the set of states, Σ is the event alphabet corresponding to V, δ is the partial transition function, and q_0 is the designated initial state. This initial FSM captures the most frequent, high-confidence behavioral sequences, omitting transitions that were pruned or rarely observed.

Iterative FSM Refinement: The FSM is iteratively refined to improve coverage and acceptance. At iteration counter $t = 0$, we evaluate M on $\mathcal{L}$ to determine the set of unaccepted events $\mathcal{I}$ and the set of uncovered transitions $\mathcal{E}$. We maintain a score list $\mathcal{S}$ of candidate paths from the original DAG, excluding edges already present in M. At each iteration, we select the highest-scoring path $P_s \in \mathcal{S}$, integrate it into M, and update $\mathcal{S}$ by removing incorporated edges. This process is repeated, incrementing t after each iteration, until either the acceptance rate of M reaches the target threshold τ or the iteration count reaches T. The result is a comprehensive FSM whose structure and transitions align closely with the protocol behaviors observed in the log stream $\mathcal{L}$.

3 Case Study: Per Layer Graph Visualization

In this section, due to page limitations, we cannot present the complete state machine diagram, as the full model contains over 400 nodes. Instead, we provide selected key paths as a demonstration.

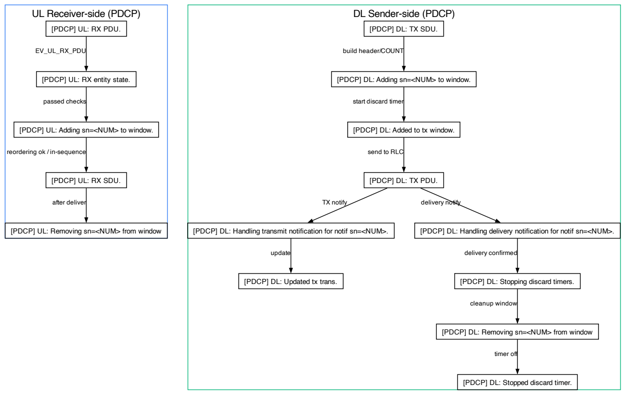
UL Receiver-side (PDCP)
[PDCP] UL: RX PDU.
EV_UL_RX_PDU
[PDCP] UL: RX entity state.
passed checks
[PDCP] UL: Adding sn=<NUM> to window.
reordering ok / in-sequence
[PDCP] UL: RX SDU.
after deliver
[PDCP] UL: Removing sn=<NUM> from window
DL Sender-side (PDCP)
[PDCP] DL: TX SDU.
build header/COUNT
[PDCP] DL: Adding sn=<NUM> to window.
start discard timer
[PDCP] DL: Added to tx window.
send to RLC
[PDCP] DL: TX PDU.
TX notify
delivery notify
[PDCP] DL: Handling transmit notification for notif sn=<NUM>.
[PDCP] DL: Handling delivery notification for notif sn=<NUM>.
update
[PDCP] DL: Updated tx trans.
delivery confirmed
[PDCP] DL: Stopping discard timers.
cleanup window
[PDCP] DL: Removing sn=<NUM> from window
timer off
[PDCP] DL: Stopped discard timer.

Figure 3: Inferred PDCP Critical Transfer

PDCP Visualization. As shown in Figure 3, the derived state machine captures the essential PDCP-layer procedures in both uplink (UL) reception and downlink (DL) transmission. On the UL receiver side, the process begins with the arrival of a PDCP PDU from the lower layer, which triggers updates to the PDCP entity's internal state, including integrity verification, duplicate detection, and sequence number (SN) management. Valid PDUs are inserted into the reception/reordering window, where out-of-order packets are buffered until the in-sequence delivery condition is met. Once the condition is satisfied, the corresponding SDU is forwarded to the upper layer, and the associated SN is removed from the window to advance the reordering frontier, thus ensuring reliable and ordered data delivery.

On the DL sender side, the procedure starts when an SDU arrives from the upper layer. The PDCP entity assigns a new SN, inserts the resulting PDU into the transmission window, and activates the discard timer to handle potential retransmission timeouts. The PDU is then handed over to the RLC layer for delivery. Subsequently, transmission and delivery notifications from the lower layer cause the PDCP entity to update its transmission state, acknowledge progress, stop the discard timer, and remove confirmed PDUs from the window.

RLC Visualization. Figure 4 illustrates the end-to-end process in RLC AM mode from the UL Receiver-side to the DL STATUS Sender-side. The receiver first receives and stores UL PDU segments, assembles them into complete SDUs, and, while advancing the reception window, checks whether a status report is required (e.g., triggered by a polling bit or status change). If so, a STATUS PDU is generated and transmitted when a DL MAC opportunity arises. Upon receiving the STATUS PDU, the peer responds with an ACK/NACK, enabling the sender to stop polling or retransmission and to perform window management and ACK processing, thereby completing the status acknowledgment and feedback loop.

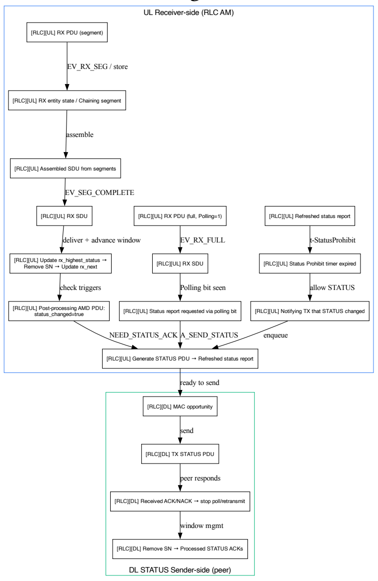
UL Receiver-side (RLC AM)
[RLC][UL] RX PDU (segment)
EV_RX_SEG / store
[RLC][UL] RX entity state / Chaining segment
assemble
[RLC][UL] Assembled SDU from segments
EV_SEG_COMPLETE
[RLC][UL] RX SDU
[RLC][UL] RX PDU (full, Polling=1)
[RLC][UL] Refreshed status report
deliver + advance window
EV_RX_FULL
t-StatusProhibit
[RLC][UL] Update rx_highest_status → Remove SN → Update rx_next
[RLC][UL] RX SDU
[RLC][UL] Status Prohibit timer expired
check triggers
Polling bit seen
allow STATUS
[RLC][UL] Post-processing AMD PDU: status_changed=true
[RLC][UL] Status report requested via polling bit
[RLC][UL] Notifying TX that STATUS changed
NEED_STATUS_ACK
A_SEND_STATUS
enqueue
[RLC][UL] Generate STATUS PDU → Refreshed status report
ready to send
[RLC][DL] MAC opportunity
send
[RLC][DL] TX STATUS PDU
peer responds
[RLC][DL] Received ACK/NACK → stop poll/retransmit
window mgmt
[RLC][DL] Remove SN → Processed STATUS ACKs
DL STATUS Sender-side (peer)

Figure 4: Inferred RLC Critical Transfer

References

[1] M. R. Ahmed, B. Nadimi, and H. Zheng. 2024. AutoModel: Automatic Synthesis of Models From Communication Traces of SoC Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 7 (Jul 2024), 2191–2204. doi:10.1109/TCAD.2024.3362598

[2] Peng He, Jianfeng Zhu, Zhi Zheng, and Min-sheng R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *Proceedings of the 2017 IEEE International Conference on Web Services (ICWS)*. IEEE, Honolulu, HI, USA, 33–40. doi:10.1109/ICWS.2017.13

[3] A. A. Ishtiaq, F. Farokhi, Z. Qian, Z. Zhao, J. R. Crandall, and Z. M. Mao. 2023. Hermes: Unlocking Security Analysis of Cellular Network Protocols by Synthesizing Finite State Machines from Natural Language Specifications. *arXiv preprint* arXiv:2310.04381 (2023). doi:10.48550/arXiv.2310.04381

[4] D. Shin, S. Messaoudi, D. Bianculli, A. Panichella, L. Briand, and R. Sasnauskas. 2020. Scalable Inference of System-level Models from Component Logs. arXiv preprint arXiv:1908.02329. Available at https://arxiv.org/abs/1908.02329.

[5] Kai Tu, Fei Cui, Runqing Ding, Lei Wang, Yuqing Zhang, and Zhiqiang Lin. 2024. Logic Gone Astray: A Security Analysis Framework for the Control Plane Protocols of 5G Basebands. In *Proceedings of the 33rd USENIX Security Symposium (SEC '24)*. USENIX Association, Article 172, 3063–3080 pages.

[6] Neil Walkinshaw, Robert Taylor, and James Derrick. 2013. Inferring Extended Finite State Machine Models from Software Executions. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, Koblenz, Germany, 301–310. doi:10.1109/WCRE.2013.6671305