A Multi-Parser Benchmark over Android Bug-Report Logs

João Alfredo Bessa[1], **Ricardo Miranda Filho**[1], **Rosiane de Freitas**[1]

[1]Instituto de Computação, Universidade Federal do Amazonas (IComp/UFAM)
Manaus - AM, Brasil

{joao.bessa,ricardo.filho,rosiane}@icomp.ufam.edu.br

Abstract. *Log parsing turns free-text log lines into event templates and is the first step of most log-mining pipelines, yet parser behavior on mobile-platform logs remains poorly characterized. We present* ***ABRLogParser****, a multi-parser benchmark that expands the ABRLog Android bug-report dataset by applying thirteen classical parsers to its eight regular-expression block types: 342 files, 14 devices, 32.0 million log lines. Every (file, parser) pair yields a released structured output and template summary, and each parser is characterized by content (how aggressively it abstracts lines into templates) and by execution (coverage, failure modes, runtime). Abstraction spans a factor of* 906 *across parsers and throughput about fiftyfold; parsers carrying the LogPAI server-log defaults degenerate into near-identity transforms on Android; and we map where each one breaks down on large inputs. The corpus has no ground-truth templates, so the artifact is a substrate for parser comparison, not an accuracy benchmark.*

Resumo. *A análise sintática de logs converte linhas em* templates *de eventos e é a primeira etapa da maioria dos* pipelines *de mineração, mas o comportamento dos* parsers *em logs móveis é pouco caracterizado. Apresentamos o* ***ABRLogParser****, um* benchmark *multi-parser que expande o conjunto ABRLog de* bug reports *Android aplicando treze* parsers *clássicos aos seus oito blocos extraídos por expressões regulares: 342 arquivos, 14 dispositivos e 32,0 milhões de linhas. Cada par (arquivo,* parser*) gera saída estruturada e resumo de* templates *publicados, e cada* parser *é caracterizado por conteúdo e por execução. A abstração varia por um fator de* 906 *entre* parsers *e a vazão cerca de cinquenta vezes; os padrões do LogPAI, ajustados para servidores, degeneram em transformações quase identidade no Android. Sem* templates *de referência, o artefato é um substrato para comparação de* parsers*, não um* benchmark *de acurácia.*

1. Introduction

System logs are the primary instrument for understanding software runtime behaviour, and converting their unstructured text into structured events, called *log parsing*, underpins downstream tasks such as anomaly detection, failure diagnosis and process mining [He et al. 2016]. A log parser maps each line onto an *event template*, keeping the constant text and replacing variable tokens by placeholders, e.g. `Fatal signal <*> in tid <*>`. Dozens of algorithms have been proposed and benchmarked, but almost exclusively on server and supercomputer logs [Zhu et al. 2019, Zhu et al. 2023]; their behaviour on *mobile* logs, which differ markedly in format and heterogeneity, is far less understood.

The ABRLog dataset [Miranda Filho et al. 2025] narrowed the data-availability gap for Android by releasing structured log blocks from 45 bug-report captures across 14 devices, together with reference templates from a *single* parser, Drain. That corpus already underpinned a parser-accuracy study [Bessa et al. 2025b] and a multi-device dataset paper [Bessa et al. 2025a]. ABRLogParser expands it directly: the same regex-extracted blocks, with the single Drain reference replaced by the output of thirteen parsers.

Parsing is not neutral preprocessing for knowledge discovery: the template vocabulary a parser emits is the alphabet every downstream miner operates on, so a parser that merges distinct events, or emits one template per line, fixes what can still be discovered afterwards. Three questions follow and organise the paper. **RQ1** (content): on identical inputs, how far do parsers differ in how aggressively they abstract Android lines into templates? **RQ2** (execution): which parsers survive the size and heterogeneity of real bug-report blocks, and how do they fail? **RQ3** (cost): how does parsing cost vary, and how far does it account for the coverage gaps of RQ2?

Answering them, the paper contributes a publicly released, containerised expansion that applies thirteen classical parsers to the eight regular-expression block types of ABRLog, yielding 3,852 structured tables with their template summaries, together with the per-parser characterisation along those three axes and an analysis of where and why each parser fails. Because the source logs are not annotated, no parsing-accuracy claims are made; every reported quantity is directly measurable in the released files.

2. Background and Related Work

Log parsing algorithms. Parsers are grouped by how they discover the constant/variable structure of messages [Zhu et al. 2019]: frequent-pattern mining (SLCT, LogCluster [Vaarandi and Pihelgas 2015], LFA), iterative partitioning (IPLoM [Makanju et al. 2009]), similarity clustering (LKE, LogMine [Hamooni et al. 2016], LenMa [Shima 2016], SHISO [Mizutani 2013]), heuristic abstraction (AEL [Jiang et al. 2008]), message signatures (LogSig [Tang et al. 2011]), parsing trees (Drain [He et al. 2017], Brain [Yu et al. 2023]), LCS streaming (Spell [Du and Li 2016]), n-gram dictionaries (Logram [Dai et al. 2022]) and evolutionary search (MoLFI [Messaoudi et al. 2018]). The LogPAI benchmark [Zhu et al. 2019] consolidated these implementations and evaluated them on Loghub [Zhu et al. 2023], dominated by server and HPC systems.

Android log datasets. Public corpora targeting Android remain scarce. ChiDroid [Karagiannis et al. 2023] collects logcat from two devices for a single log type; ABRLog [Miranda Filho et al. 2025] is, to date, the broadest, covering 24 block types from 14 devices of five manufacturers. None of these releases a *multi-parser* reference. Prior work on this corpus [Bessa et al. 2025b] measured parsing accuracy against reference templates on a smaller Android sample, reporting summary metrics only. This expansion differs on three axes and is weaker on a fourth: it covers the full 342-file corpus and all eight block types rather than a sample; it applies thirteen parsers rather than the single Drain reference shipped with ABRLog [Miranda Filho et al. 2025]; and it releases the complete parsed output instead of summary tables. What it gives up is accuracy, which an unannotated corpus cannot support.

Table 1. Source block types processed by every parser, continued in the right-hand half.

Block type	Files	Log lines	MB	Block type	Files	Log lines	MB
dmesg	45	25,658,982	1,974.4	system_properties	45	70,785	2.9
logcat	45	4,593,130	664.6	cpuinfo	45	7,636	0.5
batterystats	45	1,472,653	120.0	memory	45	1,991	0.1
wakelock	41	222,606	13.3	battery_capacity	31	31	0.0
Total	**342**	**32,027,814**	**2,775.8**				

3. The ABRLogParser Dataset

3.1. Source corpus

The inputs are the eight regular-expression block types of ABRLog [Miranda Filho et al. 2025]: `logcat`, `dmesg`, `batterystats`, `wakelock`, `system_properties`, `cpuinfo`, `memory` and `battery_capacity`, one plain-text file per block per capture, for 342 files over 45 captures from 14 devices. ABRLog defines 24 block types; the eight used here are those stored as line-oriented text, the form a parser consumes, and present in essentially every capture (31 to 45 of 45), which makes a per-parser comparison across the corpus meaningful; conclusions below are about these eight blocks, not about ABRLog as a whole. Table 1 summarizes their volume: 32.0 million lines in 2.7 GB, strongly right-skewed, with file sizes spanning four orders of magnitude from one-line battery summaries to multi-million-line kernel buffers. That spread is what stresses the parsers differently. The `dmesg` files are the complete kernel ring-buffer sections, the dominant contributor; the other seven block counts match the parent dataset exactly.

3.2. Parsers and pipeline

Thirteen parsers from the `logparser` collection were applied, namely AEL, Brain, Drain, IPLoM, LFA, LenMa, LogCluster, LogMine, LogSig, Logram, MoLFI, SHISO and Spell, spanning the families noted in Section 2. Five bundled implementations are absent, for two causes that differ in kind. DivLog and NuLog were never run: they need a GPU or an external service, an infrastructure constraint that says nothing about the logs. LKE, SLCT and ULP were run and produced no usable output, from a mix of library incompatibilities under the pinned image and memory exhaustion on these inputs. Only the former is an exclusion criterion; memory exhaustion here is itself a scalability result, of the same kind as the size-driven failures of Section 4.2. Since the per-parser cause was not isolated, we omit all three, and they should not be read as unable to parse Android logs. Parameters follow the original authors' and LogPAI defaults [Zhu et al. 2019], were held constant across all files, and are listed in the released scripts (e.g. Drain depth = 4; LogMine max_dist = 0.001; LenMa threshold = 0.9). Each (file, parser) pair runs in an isolated, timeout-supervised subprocess inside a pinned Docker image (Python 3.11), with per-job wall-clock time recorded. Successful runs emit a `*_structured.csv` (every line mapped to a template, with its parameters) and a `*_templates.csv` (each distinct template with its count); LogCluster emits only the former. Input-size caps and a per-job timeout bound the cost; capped cells were re-attempted without a bound in later passes (Section 4.2).

Table 2. Per-parser summary, ordered by coverage and continued in the right-hand half. Cov. is % of the 342 files; Templ. the median templates per file; Abstr. the median templates/lines (%); "s" and "Lines/s" the median wall-clock time and throughput on the benchmark subset. LogCluster emits no template file (—).

Parser	Cov.	Templ.	Abstr.	s	Lines/s	Parser	Cov.	Templ.	Abstr.	s	Lines/s
Brain	100.0	285	13.26	0.27	2,811	SHISO	84.2	9	1.23	0.95	839
LFA	100.0	132	5.13	0.26	3,186	LogMine	83.0	229	99.66	0.31	908
AEL	99.7	260	8.84	0.31	1,976	Logram	76.3	152	11.70	0.26	2,962
Drain	99.7	319	9.40	0.27	2,420	Spell	74.3	70	8.15	0.27	2,238
LogCluster	97.7	—	—	0.29	2,976	IPLoM	74.0	33	4.26	0.27	2,737
LogSig	84.5	5	0.11	0.29	2,918	MoLFI	68.7	26	7.50	1.54	61
LenMa	84.2	205	99.19	0.90	1,004						

4. Results and Discussion

The thirteen parsers are analyzed along the content axis (Section 4.1, RQ1), the execution axis (Section 4.2, RQ2) and the runtime axis (Section 4.3, RQ3). Because every parser processes the *identical* inputs, the differences below are attributable to the algorithms and their default configuration, not to input variation. Table 2 consolidates the per-parser medians.

4.1. Content: how parsers abstract Android logs

Figure 1 characterizes the templating behavior. For a file with L lines mapped to T templates, the *abstraction ratio* T/L measures how aggressively a parser collapses lines into patterns; a low ratio means strong aggregation. Two observations stand out (RQ1).

First, the per-file median template count (Fig. 1a) spans a factor of 64 across parsers on the very same inputs, from 5 templates for LogSig to 319 for Drain; the per-file distributions, being right-skewed, spread wider still. The tree-based and heuristic parsers (Drain, Brain, AEL) sit in a middle regime (260–319 median templates, abstraction 8.8–13.3%) that matches the compression a log parser is usually assumed to provide: many lines, far fewer distinct patterns. The message-signature and similarity-tree parsers (LogSig, SHISO) instead collapse files to a handful of templates (medians 5 and 9; abstraction 0.11% and 1.23%), merging very different messages under one pattern. Without reference templates we cannot say which regime is more faithful: aggressive merging is equally consistent with useful summarisation and with over-generalization, and nothing in the data separates them. LogCluster emits no template file, so this axis covers twelve of the thirteen parsers.

Second, and more cautionary, Fig. 1b shows two parsers, LenMa and LogMine, barely aggregating at all under their default configuration: their median abstraction ratios are 99.2% and 99.7%, almost one distinct template per line. For LogMine this is a direct consequence of the default `max_dist` = 0.001, which makes the clustering criterion so strict that nearly every line forms its own cluster; for LenMa the default similarity threshold of 0.9 has the same effect on the highly heterogeneous Android messages. The claim that this supports is narrow. Both values are LogPAI defaults tuned on server and HPC logs, so what the data show is that those defaults do not transfer to Android: a parser carried over with them can silently degenerate into a near-identity transform, yielding a

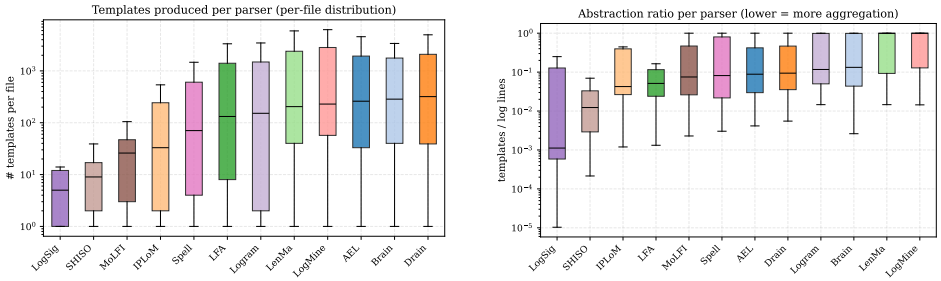

Figure 1. Per-file content behavior (log scale, outliers omitted): (a) templates per file, (b) abstraction ratio T/L. Every parser sees the identical source files, so the spread reflects the algorithms. LenMa and LogMine (rightmost in (b)) sit near 100%, i.e. almost no aggregation under their default configuration.

"parsed" output with no compression and destroying the event vocabulary a downstream miner consumes. Whether the degeneracy is correctable by tuning or intrinsic to the algorithms is *not* settled here, since every parser ran in one fixed configuration; a sweep over `max_dist` and the LenMa threshold would decide it, and we leave that to future work. The effect also explains these parsers' large total-template counts despite unremarkable per-file medians. Output expansion, the structured table's size over the input's, is by contrast uniform at 1.4–2.4×, since that table always repeats the original content and adds template and parameter columns.

4.2. Execution: coverage and failure modes

Not every pair (file, parser) yields output. Each of the 342×13 pairs ends as OK (a non-empty structured table exists), FAIL (the parser raised an error), TIMEOUT (killed at the per-job limit) or TOO_BIG (skipped by an input-size budget). Coverage ranges from 100% (Brain, LFA) to 68.7% (MoLFI); Fig. 2a makes the causes legible.

Three patterns account for most of the gaps. (i) The largest `dmesg` and `logcat` files are skipped (TOO_BIG) under the parsers whose cost grows super-linearly: MoLFI (107, its entire gap), and to a lesser extent LenMa, LogSig, Logram and SHISO (45–50 each). (ii) LogMine and Spell instead *time out* on the large inputs (58 and 83 cases). (iii) IPLoM fails on 89 of the 90 `dmesg` and `logcat` files by raising an `IndexError` on those formats, a deterministic, format-specific defect rather than a resource limit; the ninetieth file parses, so the defect is not strictly universal. The accounting is not complete for every parser: Table 2 implies 81 absent cells for Logram against 45–50 size-capped ones, so about thirty of its gaps are failures that Fig. 2a shows, but we do not itemize. Reattempted without any size budget, the capped cells overwhelmingly failed or timed out and only LogCluster recovered a few large logs, confirming that the caps tracked genuine algorithmic limits rather than arbitrary cut-offs. Robustness is therefore highly uneven: full coverage of large kernel buffers is available only from Brain, LFA, Drain, AEL and LogCluster.

4.3. Runtime and scalability

To compare cost fairly, wall-clock time was measured single-process (one worker, no concurrency) on every (file, parser) pair whose input is at most 0.1 MB, yielding 2,178

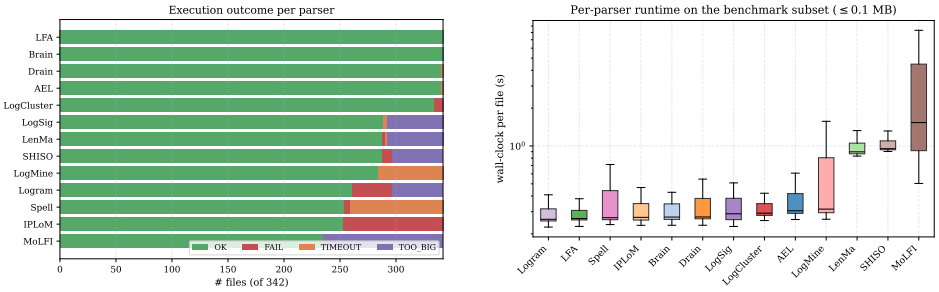

Figure 2. (a) Execution outcomes over all 342 files: gaps concentrate on the largest `dmesg`/`logcat` inputs for the costly parsers, and IPLoM fails on almost every file of those two formats. (b) Wall-clock per file on the controlled subset ($\leq$0.1 MB, single process, log scale); ten of the thirteen parsers sit on the spawn/import floor and only LenMa, SHISO and MoLFI separate from it. Coverage and median throughput are given numerically in Table 2.

successful measurements over a common set of files. Each time includes a fixed overhead of a few tenths of a second for process spawn and module import, visible as a floor in Fig. 2b.

That distribution is bimodal, but the fast mode must be read with care. Ten of the thirteen medians fall in 0.26–0.31 s, a range narrower than the spawn-and-import overhead itself: taking the fastest median as an upper bound on that floor, at most about 0.05 s of each is attributable to parsing. Differences inside this cluster are therefore not interpretable, and the throughputs derived from them are lower bounds set by interpreter startup rather than parser speed: those same ten parsers span 908 to 3,186 lines/s, a factor of 3.5 among medians indistinguishable in time. Only three parsers rise clearly above the floor: LenMa and SHISO at 0.90 s and 0.95 s ($\sim$1,000 and 840 lines/s), reflecting their pairwise message comparison, and MoLFI, whose evolutionary search costs a median 1.54 s even on these tiny files and yields 61 lines/s (Table 2). The defensible statement of the spread is thus a factor of about 52 between MoLFI and the fastest measured parser.

The link between cost and coverage is indirect, and our own numbers bound how it may be argued. MoLFI's 107 absent cells are TOO_BIG skips, not timeouts: those jobs never executed, so they cannot themselves evidence a timeout. What the runtime axis shows is that MoLFI is an order of magnitude slower than the rest already at 0.1 MB, which is why its size budget binds where it does; it is the unbudgeted re-attempt pass of Section 4.2 that connects cost to missing coverage. Large-file *capability* is therefore better read from the coverage column of Table 2 than from these small-input timings.

4.4. Limitations

The corpus carries no hand-annotated ground-truth templates, so no parsing-accuracy metric is reported. Each parser used a single fixed configuration, so the LenMa and LogMine degeneracies describe the LogPAI defaults, not the algorithms' best achievable behavior. Coverage is non-uniform by construction and the absent cells are not independent of the parser, so coverage and content are not measured on identical file sets. Timings are single runs, single-process, and include a spawn/import floor dominating all but the three

slowest parsers, so they support a coarse ordering only. LogCluster emits no template file, so the content axis rests on twelve parsers and the execution axis on thirteen. Finally, the `dmesg` files are the full kernel sections, larger than the regex-filtered `dmesg` of the parent dataset.

5. Concluding Remarks

This paper presented ABRLogParser, a thirteen-parser benchmark over the Android bug-report logs of ABRLog. Parser behavior varies by nearly three orders of magnitude in abstraction and about fiftyfold in throughput (RQ1, RQ3); the LogPAI server-log defaults make LenMa and LogMine degenerate into near-identity transforms on Android; and robustness on large kernel buffers is confined to a few parsers, with IPLoM failing on 89 of the 90 `dmesg`/`logcat` files (RQ2). The released tables let practitioners choose on the axes the data support, coverage and cost, and inspect abstraction directly; ranking by output *quality*, and settling whether the degeneracies are correctable by tuning, need a hand-annotated subset and a parameter sweep respectively, both left to future work. The dataset are released as the ABRLogParser Zenodo record [Bessa et al. 2026], an expansion of ABRLog Zenodo record [Miranda Filho et al. 2025].

Acknowledgements. This work was carried out with the support of the Coordination for the Improvement of Higher Education Personnel - Brazil (AUXPE-CAPES-PROEX) - Financing Code 001. Additionally, this work was partially funded by the Amazonas State Research Support Foundation - FAPEAM - through the PDPG-CAPES and POSGRAD 2025-2026, POSGRAD 2026-2027 projects, and by the SWPERFI RD&I Project, a partnership between the Federal University of Amazonas (UFAM) and Motorola Mobility LLC.

References

Bessa, J. A., Miranda Filho, R., Barreto, R., and de Freitas, R. (2025a). A multi-device Android mobile raw log dataset for intelligent analysis. In *XV Symposium on Computing Systems Engineering (SBESC)*, pages 1–6. IEEE.

Bessa, J. A., Miranda Filho, R., Barreto, R., and de Freitas, R. (2026). ABRLogParser: A multi-parser expansion of the ABRLog Android bug-report log dataset. `https://doi.org/10.5281/zenodo.21033598`.

Bessa, J. A., Miranda Filho, R., Souza, G., Barreto, R., and de Freitas, R. (2025b). Log parsers performance on raw logs from Android devices. *Journal of Internet Services and Applications (JISA)*, 16(1):105–116.

Dai, H., Li, H., Chen, C.-S., Shang, W., and Chen, T.-H. (2022). Logram: Efficient log parsing using n-gram dictionaries. *IEEE Trans. Software Engineering*, 48(3):879–892.

Du, M. and Li, F. (2016). Spell: Streaming parsing of system event logs. In *Proc. IEEE 16th Int. Conf. Data Mining (ICDM)*, pages 859–864.

Hamooni, H., Debnath, B., Xu, J., Zhang, H., Jiang, G., and Mueen, A. (2016). LogMine: Fast pattern recognition for log analytics. In *Proc. 25th ACM Int. Conf. Information and Knowledge Management (CIKM)*, pages 1573–1582.

He, P., Zhu, J., He, S., Li, J., and Lyu, M. R. (2016). An evaluation study on log parsing and its use in log mining. In *Proc. 46th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN)*, pages 654–661.

He, P., Zhu, J., Zheng, Z., and Lyu, M. R. (2017). Drain: An online log parsing approach with fixed depth tree. In *Proc. IEEE Int. Conf. Web Services (ICWS)*, pages 33–40.

Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. (2008). Abstracting execution logs to execution events for enterprise applications. In *Proc. 8th Int. Conf. Quality Software (QSIC)*, pages 181–186.

Karagiannis, S., Ntantogian, C., Magkos, E., et al. (2023). ChiDroid: A mobile Android application for log collection and security analysis in healthcare and IoMT. *Applied Sciences*, 13(5):3061.

Makanju, A. A. O., Zincir-Heywood, A. N., and Milios, E. E. (2009). Clustering event logs using iterative partitioning. In *Proc. 15th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD)*, pages 1255–1264.

Messaoudi, S., Panichella, A., Bianculli, D., Briand, L., and Sasnauskas, R. (2018). A search-based approach for accurate identification of log message formats. In *Proc. 26th IEEE/ACM Int. Conf. Program Comprehension (ICPC)*, pages 167–177.

Miranda Filho, R., Bessa, J. A., Barreto, R., and de Freitas, R. (2025). ABRLog: A multi-device Android bug-report log dataset. `https://doi.org/10.5281/zenodo.19401760`.

Mizutani, M. (2013). Incremental mining of system log format. In *Proc. IEEE Int. Conf. Services Computing (SCC)*, pages 595–602.

Shima, K. (2016). Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213*.

Tang, L., Li, T., and Perng, C.-S. (2011). LogSig: Generating system events from raw textual logs. In *Proc. 20th ACM Int. Conf. Information and Knowledge Management (CIKM)*, pages 785–794.

Vaarandi, R. and Pihelgas, M. (2015). LogCluster – a data clustering and pattern mining algorithm for event logs. In *Proc. 11th Int. Conf. Network and Service Management (CNSM)*, pages 1–7.

Yu, S., He, P., Chen, N., and Wu, Y. (2023). Brain: Log parsing with bidirectional parallel tree. *IEEE Trans. Services Computing*, 16(5):3224–3237.

Zhu, J., He, S., He, P., Liu, J., and Lyu, M. R. (2023). Loghub: A large collection of system log datasets for AI-driven log analytics. In *Proc. IEEE/ACM Int. Symp. Empirical Software Engineering and Measurement (ESEM)*.

Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., and Lyu, M. R. (2019). Tools and benchmarks for automated log parsing. In *Proc. IEEE/ACM 41st Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130.