Industry Practices and Event Logging: Assessment of a Critical Software Development Process

Antonio Pecchia*†, Marcello Cinque*†, Gabriella Carrozza‡, Domenico Cotroneo*†
*Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione
Universitá degli Studi di Napoli Federico II – Via Claudio 21, 80125, Naples, Italy
†Critiware S.r.l. – Via Carlo Poerio 89/A, 80121, Naples, Italy
‡ Selex ES S.p.A - A Finmeccanica Company – Piazza Monte Grappa 4, 00195, Rome, Italy
Email: {antonio.pecchia, macinque, cotroneo}@unina.it, gabriella.carrozza@selex-es.com

***Abstract*—Practitioners widely recognize the importance of event logging for a variety of tasks, such as accounting, system measurements and troubleshooting. Nevertheless, in spite of the importance of the tasks based on the logs collected under real workload conditions, event logging lacks systematic design and implementation practices. The implementation of the logging mechanism strongly relies on the human expertise.**

This paper proposes a measurement study of event logging practices in a critical industrial domain. We assess a software development process at Selex ES, a leading Finmeccanica company in electronic and information solutions for critical systems. Our study combines source code analysis, inspection of around 2.3 millions log entries, and direct feedback from the development team to gain process-wide insights ranging from programming practices, logging objectives and issues impacting log analysis. The findings of our study were extremely valuable to prioritize event logging reengineering tasks at Selex ES.

***Index Terms*—Source code analysis, event logging, development process, coding practices, industry domain.**

TABLE I. The Terminology Adopted in the Paper

Event log	A sequence of log entries. An event log is usually stored in a textual file or a database.
Log entry	A line in the event log. A log entry reports a numeric record and/or describes an event of interest detected during the execution of a given program. A log entry is usually characterized by a timestamp and a text message.
Logging point	An instruction that generates a log entry. A logging point is implemented by means of a generic output function, e.g., `printf`, `cout`, or a standard logging library, e.g., `syslog`, or a proprietary support. Examples of logging points taken from the reference development process are reported in Fig. 1 (e.g., *line 10* and *20*).
Logging mechanism	The set of (i) logging points and (ii) activation code of the logging points (if any), implemented by a given software platform.
Event logging	The practices underlying the implementation of a logging mechanism.

I. Introduction

Event logging is a well-established technique to collect numeric and textual data regarding the behavior of a computer system. Collected data are usually stored in a set of files, known as *event logs*. We clarify the terminology adopted in the paper in Table I, beforehand: while event logging-related concepts are widely accepted, we are not aware of the existence of standard definitions. Over the past decades both academic and industrial organizations have recognized that event logging is valuable for a variety of purposes, such as accounting, system measurements and troubleshooting, because it represents one of the few means to gain insights into the behavior of a computer system under real workload conditions [1], [2]. More important, event logs play a key role to conduct dependability-related tasks in a variety of application domains, such as error and failure characterization, debugging, performance modeling, anomaly detection, and security analysis [3], [4], [5], [6], [7]. In spite of the importance of the tasks based on logs, event logging is recognized to lack systematic design and implementation practices.

The implementation of the **logging mechanism** is an empirical procedure, which strongly relies on the human expertise. Fig. 1 shows two sample code snippets taken from the software being developed in the reference process. The logging point at *line 10* aims to detect an error occurring during the initialization of a data structure. The logging point is activated whenever `nResult!=0`: the activation is achieved by means of the `if` construct. The code snippet #2 reports four logging points, i.e., `BL_INFO` (*line 20, 23, 26, 29*), that are used to trace the program execution. The logging points are executed every time the `initTables` method is invoked, with no specific control structure activating the logging points. The logging points shown by Fig. 1 are heterogeneous in terms of *objectives*, i.e., the reasons why the logging point has been introduced, *placement* and *coding* of the activation constructs. These aspects regarding the logging mechanism are usually left to late stages of the software development process and are biased by the skills and past experience of individual programmers. As a result, several studies highlight that event logging is a subjective task that might lead to unstructured and inaccurate log entries at runtime [8], [9], [10], [11]. The analysis of the logging mechanism is crucial to gain insights into the practices underlying event logging.

This paper proposes a **measurement study** of event logging practices in a critical software development process at Selex ES[1]. Our study encompasses an industrial software platform in the transportation domain that involves a community of around

[1]Selex ES is a leading Finmeccanica company in electronic and information solutions for critical systems (`http://www.selex-es.com/`).

```
Code snippet #1
[omitted]
INIT_CORE();
// creation of the communication mailbox.
CREATEMAILBOX(&TxMbx, 1, &nResult);
if (nResult != 0)
{
    char * msg = "Impossibile to
      generate a mailbox in Register.";
    printf("\n%s",msg);
    //log_message(msg,strlen(msg));
    RaiseException(-1,-1,0,0);
}
[omitted]

Code snippet #2
[omitted]
void ConflictAlert::initTables()
{
    BL_INFO( Info_Initialization,
      "AircraftTypeTableValidity() -- START !!!");
    AircraftTypeTable::getInstance().init();
    BL_INFO( Info_Initialization,
      "AircraftTypeTableValidity() -- OK !!!");

    BL_INFO( Info_Initialization,
      "ConfigurationTableValidity() -- START !!!");
    ConfigurationTable::getInstance().init();
    BL_INFO( Info_Initialization,
      "ConfigurationTableValidity() -- OK !!!");
[omitted]
```

Fig. 1: Examples of logging points from the considered software platform.

100 programmers, testers, and integrators. The platform considered in this study consists of around 60 Computer Software Configuration Items (CSCI)s belonging to three product lines, i.e., *middleware*, *business logic*, and *human-machine interface*. Event logging is a key feature in the reference scenario because it is used to investigate failures occurring both during prerelease activities and system operations [12]. Our study combines source code analysis, inspection of around 2.3 millions log entries, and direct feedback from the development team to gain process-wide insights ranging from programming practices, logging objectives and issues impacting the analysis of collected logs. We conducted a quantitative assessment that allowed providing an answer to the following **research questions**:

- **RQ1**: *How do developers log?* We analyzed the procedures that are used to implement the logging points of the considered software. The analysis has been supplemented by a closer source code inspection, which revealed the programming patterns that are used to implement the activation code of the logging points. Our analysis shows that, in spite of the adoption of different logging supports, the implementation of the logging mechanism of different product lines is surprisingly similar (Findings 1 to 4).
- **RQ2**: *Why do developers log?* Event logging in the considered critical industrial domain serves three major objectives, i.e., (i) dump of the program state, (ii) program execution tracing, and (iii) event reporting. It is worth noting that the same logging procedures are used to generate entries serving different objectives. Moreover, the objectives pursued by means of event logs significantly change across different product lines (Finding 5, 6).
- **RQ3**: *How do industry practices impact event logging?* It should be observed that the first release of the platform considered in this study occurred around 20 years ago. During this timeframe, the logging mechanism has evolved based on the specific needs observed in a given product line. Different findings of our study show that industry practices have impacted event logging along several directions (Finding 1, 2, 6, 7, 8). For instance, event logging is not strictly regulated by company-wide practices, resulting into a variety of log collection mechanisms and formats that coexist in the same software product. Selex ES is currently devoting a strong effort to standardize event logging. At the time being, some of the issues pointed out by our study have been fixed by the system developers.

The rest of the paper is organized as follows. Section II provides background information on the platform considered in this study and introduces related work on event logging guidelines and assessment. Section III discusses the analysis of the logging mechanism. Section IV characterizes some event log samples generated by the reference system: the characterization made it possible to understand logging objectives and issues. Section V introduces the main industrial challenges toward a company-wide harmonization of event logging practices. Section VI highlights the threats to the validity of our study, while Section VII concludes the work.

II. Background and Related Work

The reference system is briefly described in the following Section. Moreover, we discuss the role of event log analysis within highly critical industrial domains and introduce related work addressing event logging guidelines and measurement studies.

A. Reference System

The platform considered in this study consists of around 60 Computer Software Configuration Items (CSCI)s organized into three different product lines, i.e., *middleware* (MW), *business logic* (BL), and *human-machine interface* (HMI). Fig. 2 shows the product lines. The software systems at Selex ES are developed with a component-based approach. Components are also referred to as CSCIs, i.e., an aggregation of software that satisfies an end user function and is designated for

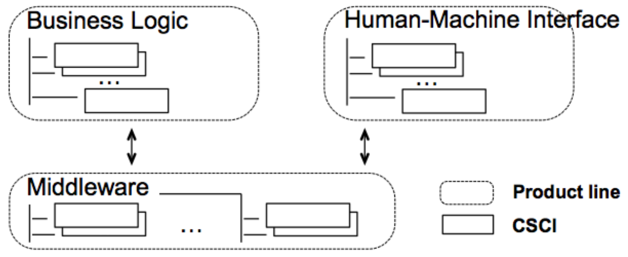


Fig. 2: Product lines implemented by the reference system.

separate configuration management, such as defined by [13]. The products and tools provided by the platform implement a variety of transportation control capabilities. The business logic implements tools supporting tasks, such as vehicle monitoring, trajectory calculation, scheduling and sequencing of arrivals. The human-machine interface allows the operators to visualize vehicle-related data, e.g., radar information, trajectory of a given vehicle, and to issue commands to the control system. The middleware level supports communication and data exchange among the CSCIs. The actual names of the CSCIs are not disclosed in Fig. 2 due to confidentiality reasons.

B. Event Logging Guidelines, Improvement and Assessment Studies

Event logging is a key feature within highly critical industrial domains. For example, the authors in [14] propose a failure detection and diagnosis framework for component-based distributed embedded systems, such as automotive. The framework consists of a logging layer that allows collecting a stream of system events; events are then fed to a diagnosis layer. Similarly, [15] proposes a log analysis framework for the Mars Science Laboratory flight software [16]. The proposal is based on the extraction of events from application text logs. The extraction is implemented through regular expressions and it aims to infer a well-structured log that can be analyzed with appropriate patterns to detect failures.

In spite of the efforts on processing and analysis techniques, logs are known to suffer of a variety of issues. Several frameworks aim to supplement the implementation of the logging mechanism by proposing uniform **log collection supports and formats**. For example, the IBM Common Event Infrastructure [17] offers an API for the creation, transmission, persistence and distribution of the log entries. Another popular framework is Apache log4x [18] . The framework is available for C++, PHP, Java and .NET applications, and it can be configured in terms of syntax of log messages, e.g., to support automatic parsing of the log entries, and destination. The Microsoft Event Log [19] protocol is another example of log-collection system; while syslog [20] has become a *de-facto standard*, which establishes both a format and a protocol to centralize the log entries. These frameworks represent a valuable support to collect, parse, and filter the event logs; however, the placement of the logging points is completely left to developers.

Recent work provides **guidelines for improving and systematizing** the implementation of the logging points. The authors in [21] propose an approach to visualize console logs: the approach allows identifying missing logging points in the source code of a given software platform. Similarly, [22] proposes to enhance the logging mechanism by adding information, e.g., data values, to ease the diagnosis task in case of failures. The authors design a *log-enhancer* tool to introduce such an information at the logging points. The Aspect-Oriented Programming (AOP) paradigm [23] is a valuable support to systematize the placement of the logging points. For example, aspects can be used to produce a log entry at each runtime exception in order to support application transparent exception reporting. In particular, in [24] it is argued that aspect-oriented exception management and logging can lead to more reliable software. The work [25] proposes a set of rules that formalize the placement of the logging points at *service-* and *interaction-*level. The logging rules proposed in the study make it possible to significantly improve the failure detection capability of event logs. The authors in [26] introduce a set of recommendations to improve the expressiveness of logs. Among the others, the authors suggest to incorporate numbering schemes and classes to categorize the information in the log and to make explicit the type of the values in the log entries.

In spite of the significant body of proposals and literature addressing the improvement of the logging mechanism, we are aware of few **measurement studies** that address event logging practices. The authors in [27] analyze the logging mechanism of three software platforms and provide insights into the coding practices underlying the logging points. The analysis shows strong similarities among the logging mechanisms of different platforms. The work [28] analyzes the density of the logging points in the source code and the changes made to log messages in the revision history, with the aim of investigating the logging practices in open-source software projects. Very recently, the work [29] provides insights into the logging practices of an industrial domain by means of code analysis and direct interviews.

We conduct a measurement study on event logging practices in an industrial domain; however, differently from previous work, our study combines source code analysis and inspection of the log entries generated during the system execution, to gain a variety insights ranging from programming practices to logging objectives. To the best of our knowledge, this is the first contribution taking this analysis approach. Furthermore, our measurements contribute to improve the findings and to establish new knowledge in an area, i.e., the analysis of the logging practices within industrial domains, which is still under explored.

III. Analysis of the Logging Mechanism

We gain insights into the logging mechanism of the reference transportation control platform by means of source code analysis. Once all the logging points have been identified, our analysis establishes the nature of the control structures that activates the logging points at runtime.

A. Distribution of the Logging Procedures

Again, a logging point is an instruction that, once triggered, generates an entry in the event log during the program execution. The source code analysis encompasses 920,809 lines of C and C++ code implementing the product lines described in Section II-A. We first identify all the **procedures**, e.g., generic output functions, proprietary class methods and home-made functions, that are used for logging purposes in the reference system. The identification has been conducted by interacting with the development team, referring to available project documentation and by means of source code inspection. As

TABLE II. Lines of Code (LOC), Logging Points (LP) and Density of the Reference Software Platform

CSCIs	LOC		Logging Points (LP)	*Density*
	C	*C++*		
All CSCIs	204,951	715,858	30,138	3.27%
CSCI #1 (MW)	9,391	23,470	466	1.42%
CSCI #2 (BL)	0	59,458	2,744	4.62%
CSCI #3 (HMI)	94,482	2,267	2,710	2.80%

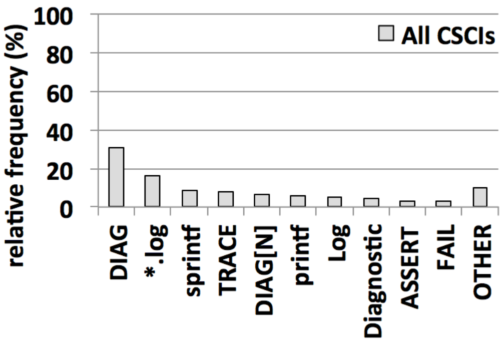


Fig. 3: Relative frequency of the top occurring logging procedures.

reported by the first row of Table II, we identified total 30,138 logging points. The density of the logging mechanism, i.e., $(LP/LOC) \cdot 100$, is around 3.27%. This finding indicates that event logging is a pervasive feature in the software platform considered in this study, i.e., one logging point around every 30 LOC. It is worth noting that a similar logging density has been observed by [28] in the context of open-source software.

We identified around 25 procedures that are used to implement the logging points across the three software product lines. Let LP_p denote the number of logging points implemented by means of a given procedure, e.g., `printf` or `DIAG`. Fig. 3 reports the relative frequency of the top 10 occurring logging procedures with respect to the total number of logging points, i.e., $(LP_p/LP) \cdot 100$. `DIAG` accounts for total around 31% logging points. Logging achieved by means of generic output functions (i.e., `sprintf`, `printf`) accounts for total around 14% of cases. The remaining 15 procedures represent less that 10% of cases (`OTHER` bar, i.e., the rightmost of Fig. 3).

Finding 1: Event logging is a widely adopted practice in a critical industrial domain. We estimated that around 3.27% of the overall source code of the software considered in this study consists of logging points. The logging mechanism relies on a variety of procedures; however, only a small subset of the procedures is extensively used to implement the logging mechanism.

We reduce the grain of our analysis at CSCI-level. Due to space limitations, we discuss the results obtained for the CSCI representing the top contributor to a given product line (similar findings have been observed in the other cases). Table II (rows 2-4) reports LOC, LP and density of each top CSCI along with the product line it belongs. It is worth noting that

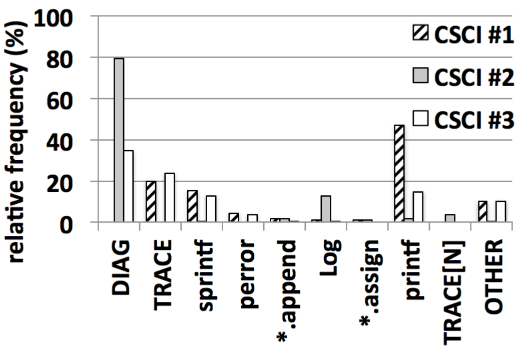


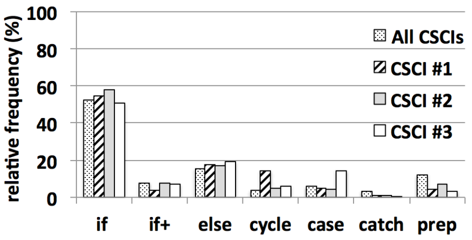
Fig. 4: Relative frequency of the top occurring logging procedures by CSCI.

the three CSCIs account for total 189,068 LOC, i.e., around 21% of the entire source code available in this study. Fig. 4 reports the relative frequency of the most recurring logging procedures by CSCI. It can be observed that the distribution of the logging procedures is significantly different across the CSCIs. For example, around 80% of the logging points of the CSCI #2 are implemented by `DIAG`; similarly, the CSCI #1 strongly relies on one procedure, i.e., `printf`. The density of the logging code ranges from a minimum of 1.42% (CSCI #1) to maximum of 4.62% (CSCI #2), as indicated by Table II.

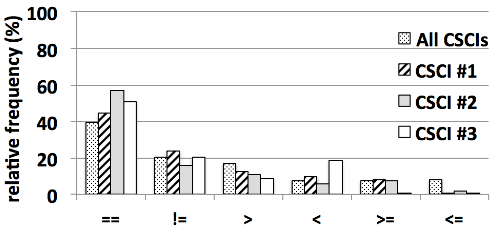
Finding 2: Almost all the logging points in a CSCI are implemented by means of a very limited number of distinct procedures; moreover, developers across different CSCIs adopt different logging procedures. Direct interviews with the programmers' community at Selex ES confirmed that the developers in a product line share common rules regarding the implementation of the logging points; however, no strict logging rules exist across different product lines.

B. Implementation of the Logging Mechanism

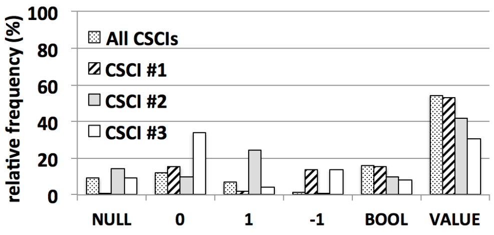
We investigate the nature and the distribution of the programming constructs that activate the logging points. Let us clarify this concept by recalling the code snippet #1 reported in Fig. 1, which has been discussed in Section I. The logging point at *line 10* is activated by the `if` control structure. The variable `nResult` is tested against the flag value `0`: the logging point is executed whenever the test condition `nResult!=0` is met during the program execution. We developed a parser to automate the analysis of the code that activates the logging points. The parser identifies all the logging points beforehand. The logging points are represented by the procedures that have been discussed in the previous Section III-A. Starting from the location of a logging point, the parser initiates a backward source code inspection to establish the control structure that is responsible for the activation of the logging point at runtime. The parser consists of a set of `bash` scripts developed under the Linux OS and adopts regular expressions to identify the logging points within the code of a given platform. The technicalities underlying the parser are not discussed here due to space limitations.



(a) Control structures.



(b) Test operator.



(c) Test flag.

Fig. 5: Relative frequency of the control structures, test flags and operators implementing the logging mechanism adopted by the system considered in the study. The histograms report measurements conducted both at system-wide level (i.e., All CSCIs) and CSCI level.

The parser assigns each logging point to one out of the control structure reported by the horizontal axis of Fig. 5a. It should be observed that 3,528 out of the total 30,138 logging points reported in Table II, i.e., around 11.7%, could not be assigned by the parser to any of the control structures indicated by Fig. 5a. For example, it can happen that no control structure activates a given logging point (such as the code snippet #2 reported in Fig. 1). The logging points that have been not classified by the parser are not included in the following statistics. Fig. 5a reports the relative frequency of each control structure measured both at system- and CSCI-level. For example, the leftmost bar of Fig. 5a indicates that 52.2% logging points across all the CSCIs are activated by the `if` construct (i.e., 13,890 out of total 26,610 logging points that were classified by the parser). Fig. 5a reports the frequency measured across all the CSICs and by the CSCI representing the top contributor to a product line. On average (i.e., estimated across all the CSCIs), around 60% of logging points are triggered by means of the `if` construct. This percentage includes both the `if` category (i.e., the condition tested by the `if` consists of a single logical clause) and the `if+` category (i.e., the condition consists of several logical clauses combined with `AND`/`OR` operators); however, `if+` is ways less common when compared to `if`. The logging points activated during the execution of an `else` branch represent another significant percentage of cases, i.e., 15.3% measured across all the CSCIs. The logging points activated by preprocessor directives (i.e., `prep` category in Fig. 5a), account for total around 12.1% of cases. The analysis conducted at CSCI-level confirms the findings observed at system-wide level. For example, the relative frequency of the `if` constructs ranges between a minimum of 50,6% (CSCI #3) and a maximum of 58.0% (CSCI #2).

Finding 3: The most adopted coding pattern that is used to activate the logging points is *if(condition) then log_error()*. We observed that around 60% of logging points are activated by this construct according to the data available in this study. The events of interest are detected by testing one or more variables collected during the program execution against flag values. A log entry is generated whenever the condition tested by the `if` is met.

We analyze the conditions that are tested to activate the logging points. To this aim, the focus is on the `if` encompassing a single clause, which is the most frequent coding pattern. In particular, we consider the *operator* and the *flag* (i.e., the value a variable collected during the program execution is tested to) that are used to implement the condition tested by the `if`. For example, the checking `if (m_nTimeHorizonMax<0)`, consists of the $<$ operator and the 0 flag; similarly, `if (StatoCorrente == WAITTORUN)` is composed by the == operator and the `WAITTORUN` flag. Both the examples have been extracted from the source code available in the study. Fig. 5b reports the relative frequency of the **operators** that are used to perform the comparison between a variable and the flag value. The histograms indicate that == is the most adopted operator. We observed very similar operator distributions across the reference CSCIs. Fig. 5c reports the frequency of the **flag values**. Fig. 5c reports both commonly adopted flags (such as `null`, `0`, and `boolean`) and the `value` category, which includes all the remaining test flags (e.g., `WAITTORUN` taken from the above example). The test conditions implemented by the `if` construct mainly encompass the boolean and value category. Again, percentages observed across different CSCIs are surprisingly similar.

Example #1: State dump

```
[omitted]
CString sLine;
DIAG("-----------------------------------------------------------------------------------------");
sLine.Format("VehicleId: %s - DEP: %s - DES: %s - EOBT: %s - VEHICLEDATE: %s - OWNER: %s",
            pFpl->m_VehicleId, pFpl->m_Dep, pFpl->m_Des, pFpl->m_EOBT, pFpl->m_VehicleDate.Format("%b %d %Y %H:%M"), pFpl->m_OwnerSectorId);
DIAG(sLine);
[omitted]
```

Log entries:

```
10 11:46:55.927 [CSCI_#2-006] -----------------------------------------------------------------------------------------
10 11:46:55.927 [CSCI_#2-006] VehicleId: * - DEP: LTBA - DES: OMDW - EOBT: 1040 - VEHICLEDATE: Apr 10 2014 00:00 - OWNER: STA
```

Example #2: Execution trace

```
[omitted]
case NOTUPDVOLO:    // messaggio con Des=3.
{
    #ifdef _DEBUG
        TRACE0("ACDPathManager: <------- Ricevuto messaggio NOTUPDVOLO.\n");
    #endif
    // diagnostica.

    DIAG("ACDITF: NOTUPDVOLO received.");
[omitted]
```

Log entries:

```
10 11:46:55.660 [CSCI_#2-006] ACDITF: NOTUPDVOLO received.
```

Example #3: Event reporting

```
[omitted]
// Verifico se e' presente un segment violation

#if CONFLICT_CONFIRMATION_LOGIC_PERIODICALLY != 1
if (sv_man.sv_list.emptylist())
{
        DIAG("sv_man.sv_list is EMPTY, Non Ho Segmenti in Conflitto");
        return;
}
[omitted]
```

Log entries:

```
07 00:02:42.067 [CSCI_#2-000] sv_manśv_list is EMPTY, Non Ho Segmenti in Conflitto
```

Fig. 6: Examples of code snippets reporting different usages of the logging points.

Finding 4: In spite of the adoption of a variety of logging procedures and the lack of company-wide rules among the development teams, the implementation of the logging mechanism is surprisingly similar across different CSCIs. Event logging is a strongly developer-dependent feature; however, the programmers belonging to different product lines exhibit a well-established coding attitude.

IV. Event Logs Characterization

The source code analysis has been supplemented with the inspection of the event logs collected during the execution of the target system. A closer event log inspection provided significant insights into the logging objectives and log analysis drawbacks that are caused by the lack of standardized practices among software developers in the reference process.

A. Available Dataset and Analysis Method

The available dataset encompasses sample event logs generated during regular and failure runs of the system. The samples have been collected at the real system production site and account for total around 2.3 millions log entries. The second column of Table III reports the total number of entries collected by CSCI. Again, we consider the top contributor to each product line. Event logs contain a large amount of entries, which are extremely hard to be manually inspected. However, in spite of the number of entries, information provided by the event logs can be collapsed to a very limited number of **patterns**. Let us clarify this concept by means of an example. The entries

```
03/06/2014 20:40:21 - fpt track id 1071 is not linked
03/06/2014 20:40:21 - fpt track id 1072 is not linked
```

report two simplified lines taken from the log of the CSCI #3 (HMI product line). The entries share the following pattern once the variable fields have been replaced with a token:

```
DATE HH:MM:SS - fpt track id NUMBER is not linked.
```

We identify the patterns in log with the aim of narrowing down the number of lines to be inspected. The log entries consist of variable and constant fields. The variable fields are represented by terms that identify manipulated objects or states of the program, such as IP and memory addresses, user names, and dates. Constant fields represent words that do not change across the log entries referring to the same pattern: for this reason, we de-parameterized the textual content of the event logs, i.e., the variable fields are replaced with a generic token (e.g., `DATE`, `NUMBER`, in the above example). The approach adopted in this study has been already used by several works in the area of log analysis, such as [9], [30], [31]. This procedure reduces the number of distinct messages, as most of the entries in the log differ because of the variable fields. The rightmost column of Table III reports the number of patterns observed in the event logs out of the original number of entries.

TABLE III. Breakdown of the Total Number of Log Entries and Patterns by CSCI

	Log entries	*Patterns*
CSCI #1 (MW)	1,150,527	69
CSCI #2 (BL)	1,183,840	124
CSCI #3 (HMI)	20,243	66
Total	2,354,610	259

B. Event Logging Purposes

All the patterns identified by means of de-parameterization have been manually inspected to gain insights into the analysis objective pursued by a given entry in the log. We found out that event logging in the reference industrial domain serves three major purposes:

- **State dump**: this type of pattern reports the value of critical variables or entire data structures in the event log. The entries belonging to this category usually report small, if not none, textual information content.
- **Execution tracing**: this pattern notifies that a given source code location has been reached during the program execution. For example, the log entries belonging to this category are used to notify the beginning and the termination of a function call (such as the code snippet #2 reported by Fig. 1) or the execution of branch.

- **Event reporting**: the log entries belonging to this pattern provide textual information regarding the occurrence of an event of interest. Typical usages include *information* events, i.e., regular situations that are pointed out during the program execution (e.g., commands issued to the system, startup and shutdown of a software module) and *error* events (e.g., the occurrence of an error condition in the program, the exhaustion of a critical resource).

Each pattern encountered in the logs has been assigned to one of the above described categories. When needed, we directly communicated with the system developers to disambiguate the patterns and to understand the type of analysis objective they were intended to support.

Finding 5: Event logging serves three major purposes in the reference domain, i.e., state dump, execution tracing and event reporting.

Event logs are extensively used to support debug and system integration activities in the reference process: the manual inspection of the execution trace in the log allows developers to reduce the amount of code to be reviewed for troubleshooting purposes. Fig. 6 reports a sample code snippet for each objective pointed out by our study. For each snippet, we report (i) the logging point and (ii) a sample entry that was generated at runtime in the log (some fields have been partially obfuscated due to confidentiality reasons). For instance, the example #2 shows a logging point, which notifies that the program executed the `NOTUPDVOLO` case; similarly, the example #3 reports a logging point that checks and notifies the state of data structure. It should be observed that the same logging procedure, i.e., `DIAG` in Fig. 6, was used by the system developers to implement a logging point irrespectively of its analysis purpose.

The event logs generated by different product lines focus on rather different analysis purposes. Fig. 7 reports the relative frequency of the log patterns by purpose and product line. For example, the leftmost bar of Fig. 7 indicates that around 42% of patterns observed for the CSCI #1 (which belongs to the middleware product line) serve state dump purposes. We observed that the patterns generated by the HMI product line are almost all used for event reporting. All

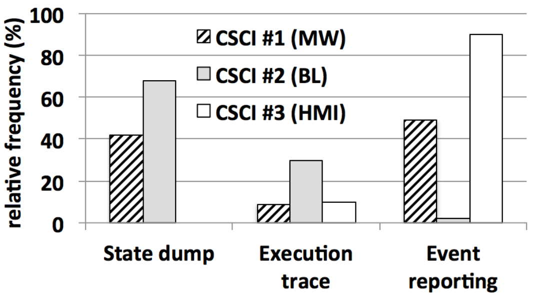


Fig. 7: Relative frequency of the log patters available in the study, by purpose and product line.

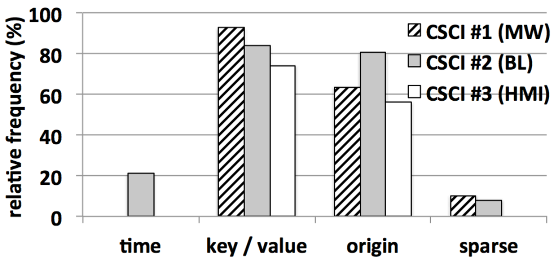


Fig. 8: Relative frequency of the log patters by format issue considered in this study. Please observe that, given a CSCI, the bars might not sum up to 100 because the same log entry might be affected by multiple issues at the same time.

the product lines make a significant use of execution trace capabilities, while state dump is almost absent in CSCI #3 based on the data available in the study.

Finding 6: The event logs generated by the CSCIs that belong to different product lines pursue different analysis objectives. Event logs have been evolving into different directions according to the specific needs and internal analysis procedures adopted by a given development team.

C. Issues Impacting Log Analysis

The manual inspection of each pattern made it possible to gain insights into format and representation issues of available logs. Table IV shows several log entries generated by each product line. Each entry is identified by a numeric id that is reported in the leftmost column of Table IV. Some limitations observed in the data are discussed in the following. For each case, we focus on analysis drawbacks and implications.

Finding 7: The event logs available in this study are mainly conceived for manual analysis purposes. Moreover, they are only occasionally shared among different development teams.

Lack of a unique timestamp format. A timestamp denotes the time the entry was recorded in the log. We observed that the CSCIs belong to different product lines adopt different timestamp representations. For example, the timestamp of the BL lines is represented by the `04 12:15:32.708` format; on the other hand, the format adopted by HMI is `03/06/2014 20:26:34`, while the MW adopts the format `2014-04-03 17:38:19.486`. The different timestamp representations can be easily inferred by looking at the sample entries reported by Table IV. Moreover, when a date is part of the text message of a log entry, its representation might differ from the one adopted by the timestamp, as shown by *line 15* of Table IV. We estimated that around 21% patterns observed for the CSCI #2, i.e., 26 out of 124, report a date as a part of the textual content of the entry, as shown by the leftmost bar of Fig. 8. Adopting different time formats makes it hard to develop an automatic support to browse logs from different product lines; however, it is not a critical issue in case of manual log analysis.

TABLE IV. Examples of Log Entries

ID	Family	Timestamp	Sample Message
1	MW	2014-04-03 18:17:55.010	POS TIMECOUNT SECTOR ACTION FLI CALLSIGN
2	MW		0 1 0 3 352 GMI**
3	MW		1 1 0 3 815 UAE**
4	MW		2 1 0 3 503 OHY**
5	MW		3 1 0 3 332 HVK**
6	BL	04 12:15:32.669	[BL_CSCI-011] ** ** Z ACU C 05550 Apr 04-11:32:36 IG 0001 IN:00 OUT:00 1 [0.0] R:1
7	BL	04 12:16:37.849	[BL_CSCI-017] ACD: FillItemVehicle: Sezione COP - OUT: ” - Type: 0 - Range: -5232 - Bearing: -5230
8	BL	04 12:16:37.849	[BL_CSCI-017] Acd: fillFdpSection: Id: ** - KeyId:5760646
9	BL	15 11:19:00.383	[BL_CSCI-034] Event: CheckExpired: currTime Apr 15 2014 11:19:00 - Range [Apr 15 2014 11:19:00 / Apr 15 2014 11:19:00]
10	BL	15 11:19:00.383	[BL_CSCI-034] Event: CheckExpired: current Time lesser than Range
11	BL	15 11:19:00.393	[BL_CSCI-034] CCheckFplChanges: LoadFpl: VEHICLE ID: BER**
12	BL	15 11:19:00.485	[BL_CSCI-034] ACD - VehicleId: BER** - ItemVoloPoint: - PointType:
13	BL	17 16:29:58.969	[BL_CSCI-000] NEW CONFLICT: viol_ID 17534 - traj1_ID 2977 - seg1_ID 13946 - Last Point x=[-56.000000] y=[-88.000000]
14	BL	17 14:31:10.530	[BL_CSCI-000] SVM_addSegmentViolation: violID 16684, tStart: 15:09:06 , traj1ID 2795:0 seg1ID 55260, Last Point (-108,-16)
15	BL	04 12:15:32.708	[BL_CSCI-016] Order: HandleTrajectoryPrediction: VHLID TH*** - TerTime: Apr 04 12:32:54
16	BL	04 12:15:32.718	[BL_CSCI-016] COrder: SetStateRelatedTrj [THY2GT] - ROUTEREPORTFIX [CLKPN] After SetStateRelateOrder - State[7]
17	BL	17 16:22:55.021	[BL_CSCI-000] FindRisk: FL1=(THY3VG) SectIDSUCC=(ACU) - CFL1=330.000000 - XFL1=330.000000
18	HMI	2014-04-03 17:38:19.486	- CVawConflict– CVawConflict::resetConflictVawTable call
19	HMI	03/06/2014 20:26:35	- TRS_SUP Warning: SUP_IMsend: Send message to HDI (applic_id = 0 - order_id = 162 - size = 62)
20	HMI	07/15/2013 11:43:04	- TRS_VEHICLE Message: operator reset hook of vehicle BER**

Weak naming convention. The same concept might be occasionally referred to with different identifiers. We observed several instances of this issue, even within the same CSCI. For example, the *callsign*, i.e., the id of a vehicle, is represented by `VEHICLE ID` (line 11), `VehicleId` (line 12), or `VHLID` (line 15); similarly, the *current time* concept is indicated either by `currTime` or `current Time` (line 9 and 10, respectively). Both the example are reported in Table IV. The adoption of different identifiers to represent the same concept is a drawback that strongly impacts both manual and automatic log analysis.

Key/value pairs representation. The log entries make extensive use of keys and values. Fig. 8 shows the percentage of patterns that report at least one key/value by CSCI. For example, we observed that 64 out of 69 patterns generated by the CSCI #1 report at least one value. As it can be inferred from Fig. 8, the number of patterns reporting at least one value ranges from a minimum of 74% (CSCI #3) to a maximum of 93% (CSCI #1). A closer look into available data revealed the adoption of a variety of key/value representation formats. For example, Table IV shows sample entries adopting the `key:value` (line 8), `key [value]` (line 16), and `key=value` (line 17) representations. Key/value pairs might be represented with different formats even within the same log entry, such as shown by the line 17.

We conducted a systematic analysis of the patterns reporting one or more key/value pairs. The analysis made it possible to identify 9 different ways of representing key/value pairs. Fig. 9 reports the percentage of patterns by key/value representation. Results indicate that CSCI #1 and #3 makes extensive use of the `key=value` representation, which account for more than 60% of cases; on the other hand, the CSCI #2 shows relevant percentages across a variety of representations. It is worth noting that for a significant percentage of patterns, which ranges from 9% (CSCI #3) to 22% (CSCI #2) there is no key indication regarding the value, i.e., `missing` point of Fig. 9. The CSCI #2 is the only CSCI where we observed log entries containing different key/value formats within the same text body (rightmost bar of Fig. 9). Adopting a uniform key/value format is extremely important to facilitate the splitting of the log entries into token [26].

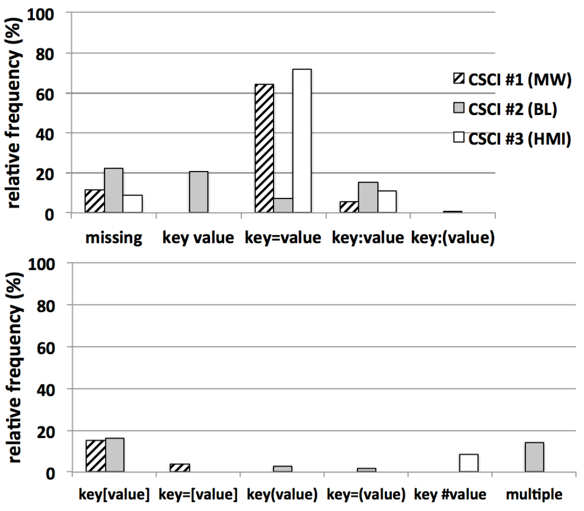


Fig. 9: Relative frequency of log patters, by key/value representation format, for each product line.

Same information sparse across many lines. We observed that a common logging practice is to dump entire data structures, such as the elements of a list, or the rows of a table, across several lines in the log. This practice is adopted for the sake of better visualization of the information in the log. For example, the lines from 1 to 5 of Table IV report a tabular representation of the fields indicated by line 1. We estimated that around 10% of patterns observed for the CSCI #1 and #2 (i.e., `sparse` data point of Fig. 8) result from this practice. While this approach is extremely valuable to ease the visualization of complex data structures, it makes it hard to infer the dump of a given data structure automatically. The heading line does not indicate the size of the dump (i.e., the number of entries related to the same dump) and other independent lines might interleave the dump in the case of concurrent access to the log.

Other context information. The use of a severity value for each message in the log is not a common practice in the software platform considered in this study. As shown by Table IV, the CSCIs belonging to BL and MW families do not adopt a severity indication. On the other hand, a significant number of collected patterns (e.g., around 81% in the case of CSCI #2) provides an indication of the *originating entity* of the log entry, as shown by the `origin` series of Fig. 8. However, as a feedback from the development team, we observed that the originating entity of a log entry might assume a variety of formats, such as the thread id, the name of the CSCI, or the name function. The systematic adoption of severities and event origins might be strongly beneficial to automatic log filtering activities in reference scenario.

Finding 8: The log entries make extensive use of fields, such as dates, message origins, key/value pairs; however, there is no common representation format both within the same CSCI, and across different CSCIs. While weak format conventions are less critical in the case of manual forensics of collected logs (which is the main use case of event logs at Selex ES), they pose several challenges to the development of automatic log analysis tools.

V. Industrial Challenges with Event Logging

Selex ES is currently devoting a strong effort to standardize event logging practices among the teams belonging to different product lines. Our measurement study was extremely valuable to this objective: for example, it allowed identifying a set of obsolete logging procedures that will be progressively thrown out within the next software releases, or to pinpoint log format issues that have already been fixed. Nevertheless, there are several inherent industrial challenges that prevent a real company-wide harmonization of current logging practices. It is worth noting that the CSCIs are developed, integrated and maintained by independent teams. By means of interviews with different developers and integrators, we realized that event logging is not regulated by strict practices: the three product lines have different log structures and different log production mechanisms. Nevertheless, this is an expected drawback for any large-scale software integration process where each development team is the exclusive owner of the *logging knowledge*.

Event logging has been constantly improving and evolving during the 20 year project timeframe. We observed that the logging practices are scarcely documented, such as the logging mechanism and related APIs. The logging knowledge is demanded to a limited number of managers and senior developers who encourage a basic set of practices and recommendation at team-level. Selex ES is encouraging the definition of a **logging policy**, i.e., a comprehensive company-wide document which systematizes logging concepts and consolidates the amount of best logging practices gained by means of a constant field experience. The logging policy encompasses different areas, such as log production and management, format of the log entries, and placement of the logging points. The definition of a logging policy is not a trivial task because logs from product lines serve different purposes. The logging policy is expected to enforce uniform data representations (e.g., timestamps, dates, key/value pairs), the adoption of a severity value for each log entry, and to establish a thesaurus to group different terms representing the same concept. Moreover, the policy should address the means that aim to accomplish the enforcement, such as contracts, APIs, and source code reviews. Log analysis procedures adopted at Selex ES would strongly benefit from the implementation of the logging policy. Many current analysis tasks, such as retrieving the log entries produced by a given CSCI at the time a failure occurred, checking the value of critical inputs/variables, retrieving information from multiple event logs, are strongly manual. The adoption of a logging policy would make it possible to develop a baseline of tools to speed-up log forensics through browsing and filtering utilities across different CSCIs.

At the time being, it is hard to predict the time it will take to migrate towards a uniform logging mechanism implementing the policy. It is not even clear whether reengineering the logging mechanism is actually feasible in practice. As pointed out by our study, logging is a pervasive feature, with around 3.27% of the entire source code of the reference platform being represented by logging points. Adding, removing or updating a logging point involves a change to the source code, which represents a cost. Let us consider a hypothetical software project accounting for total 30,138 LOC, i.e., the number of logging points of the software platform considered in this study. This project would cost around 187K$ with an effort of 124 person-month over 18 months (by assuming 1,500$ as cost per person-month), according to a constructive cost model (COCOMO II). Despite being a simplistic analogy, we might assume the cost of this hypothetical software to be a rough estimate of the cost it might take to reengineer the logging mechanism of the target system. As a result, a comprehensive reengineering strategy must clearly cope with the return of investment (ROI) that can be actually achieved by means of better event logging strategies.

VI. Threats to Validity

As for any data-driven measurement study, results might be biased by the data available in the study. Event logging strongly relies on the human expertise. For this reason, the measurement of the logging practices can potentially lead to rather different results by moving from an application domain to another. More important, it should be considered that our aim is not to provide definitive measurements of current logging practices, which would clearly require the analysis of a large set of industrial projects.

We are aware that the findings provided by our study have been inferred based on one case study, such as [29]; however, our measurements contribute (i) to improve the findings and (ii) to establish new knowledge in an area, i.e., analysis of the logging practices within industrial domains, which is still an open and rather under-explored research field. Our analysis encompasses a highly critical industrial software in the transportation domain written in C, and C++ at Selex ES. We are confident that our results on logging practices should be generalizable to other critical industrial systems. It is worth noting that the results regarding the analysis of the logging mechanism discussed in Section III are partially confirmed by other studies in the area [27], [28]. Moreover, the classification of the logging patterns in Section IV has involved a strong manual effort and direct communication with the software developers at Selex ES, which increases our level of trust on analysis results.

VII. Conclusion

System developers, testers and integrators, agree that event logs are a key source of information for a variety of critical tasks. The lessons learnt in the context of our activity indicate that the logging process is strongly developer dependent. Our study provides a variety of findings along different research directions, such as current logging practices and objectives, and event logging as affected by industry practices.

In the future we will collaborate with Selex ES to both establish the logging policy and to develop a prototype API to enforce the policy across the different CSCIs. Our future work will also encompass the analysis of further industrial software platforms in order to decrease the threats to validity of our study and to fill the gap between industry practices and effective event logging. More important, we will provide a set of recommendations for the practitioners aiming to reengineer an existing logging infrastructure in a large-scale development process, based on the findings of our assessments.

Acknowledgment

This work has been partially supported by the MINIMINDS PON Project (n. B21C12000710005) funded by the Italian Ministry of University and Research and by the European Commission in the context of the FP7 ICEBERG Project, Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 324356.

References

[1] R. K. Iyer, Z. Kalbarczyk, and M. Kalyanakrishnan. Measurement-Based Analysis of Networked System Availability. *Performance Evaluation: Origins and Directions*, pages 161–199, 2000.

[2] A. J. Oliner and J. Stearley. What Supercomputers Say: A Study of Five System Logs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 575–584. IEEE Computer Society, 2007.

[3] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and Modeling of Computer Reliability as Affected by System Activity. *ACM Transactions on Computer Systems*, 4:214–237, August 1986.

[4] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why Do Internet Services Fail, and What Can Be Done About It? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[5] B. Schroeder and G. A. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 249–258. IEEE Computer Society, 2006.

[6] B. Murphy and B. Levidow. Windows 2000 Dependability. In *MSR-TR-2000-56 Technical Report*, Redmond, WA, June 2000.

[7] A. Pecchia, A. Sharma, Z. Kalbarczyk, D. Cotroneo, and R. K. Iyer. Identifying compromised users in shared computing infrastructures: A data-driven bayesian network approach. In *Proceedings of the Int'l Symposium on Reliable Distributed Systems (SRDS)*, pages 127–136. IEEE, 2011.

[8] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure data analysis of a LAN of windows NT based computers. In *Proceedings of the Eighteenth Symposium on Reliable Distributed Systems (18th SRDS'99)*, pages 178–187, Lausanne, Switzerland, October 1999. IEEE Computer Society.

[9] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *Int'l Conference on Dependable Systems and Networks (DSN 2008)*, Anchorage, Alaska, June 2008.

[10] C. Simache and M. Kaâniche. Availability assessment of sunOS/solaris unix systems based on syslogd and wtmpx log files: A case study. In *PRDC*, pages 49–56. IEEE Computer Society, 2005.

[11] M. F. Buckley and D. P. Siewiorek. VAX/VMS event monitoring and analysis. In *FTCS*, pages 414–423, 1995.

[12] M. Cinque, D. Cotroneo, R. Della Corte, and A. Pecchia. Assessing direct monitoring techniques to analyze failures of critical industrial systems. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, Nov 2014.

[13] U. S. Department of Defense. MIL-STD-498. Overview and Tailoring Guidebook. *[Online]. Available at: www.abelia.com/498pdf/498GBOT.PDF*, 1996.

[14] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Reflection: Dynamic model-based analyis of component-based distributed embedded systems. In *Modellierung von Automotive Systems*, 2006.

[15] H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal Analysis of Log Files. *Journal of Aerospace Computing, Information and Communication*, 7(11):365–390, 2010.

[16] Mars Science Laboratory. http://mars.jpl.nasa.gov/msl.

[17] IBM. Common Event Infrastructure . http://www-01.ibm.com/software/tivoli/features/cei.

[18] The Apache Software Foundation. Logging services project. http://logging.apache.org/.

[19] Microsoft Corporation. Windows Event Log. http://msdn.microsoft.com/en-us/library/aa385780(v=VS.85).aspx.

[20] C. Lonvick. The bsd syslog protocol. *Request for Comments 3164, The Internet Society, Network Working Group, RFC3164.*

[21] A. Rabkin, W. Xu, A. Wildani, A. Fox, D. Patterson, and R. Katz. A Graphical Representation for Identifier Structure in Logs. In *Proceedings Workshop on Managing systems via log analysis and machine learning techniques (SLAML)*, 2010.

[22] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the International Conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 3–14, 2011.

[23] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*.

[24] J. Viega and J. Vuas. Can aspect-oriented programming lead to more reliable software? *Software, IEEE*, 2000.

[25] M. Cinque, D. Cotroneo, and A. Pecchia. Event logs for the analysis of software failures: A rule-based approach. *Software Engineering, IEEE Transactions on*, 39(6):806–821, June 2013.

[26] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive Logfiles for Autonomic Systems . *Proceedings of the IEEE Parallel and Distributed Processing Symposium, 2004*, April 2004.

[27] A. Pecchia and S. Russo. Detection of software failures through event logs: An experimental study. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 31–40, Nov 2012.

[28] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.

[29] Q. Fu, J. Zhu, W. Hu, J. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? An empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 24–33, New York, NY, USA, 2014. ACM.

[30] J. Stearley and A. J. Oliner. Bad words: Finding faults in spirit's syslogs. In *Proc. Int'l Symposium on Cluster Computing and the Grid (CCGRID 2008)*, pages 765–770.

[31] A. Pecchia, D. Cotroneo, R. Ganesan, and S. Sarkar. Filtering security alerts for the analysis of a production saas cloud. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14. IEEE Computer Society, 2014.