Studying and Complementing the Use of Identifiers in Logs

Jianchen Zhao*, Yiming Tang*, Sneha Sunil† and Weiyi Shang*
*Department of Computer Science and Software Engineering, Concordia University
Montréal, QC, Canada
Email: {z_jianc, t_yiming, shang}@encs.concordia.ca
†Department of Computer Science and Engineering, SRM Institute of Science and Technology
Kattankulathur, Tamil Nadu, India
Email: ss4795@srmist.edu.in

***Abstract*—Logs contain a large amount of curated run-time information about the process of a software. Modern software systems have become more complex and larger in scale. They are typically executed in parallel or distributively, resulting in interleaved software logs and making log analysis challenging. Despite extensive research on automated logging analysis, none to our knowledge focuses on the use of logs, and they rarely augment logs to help with simpler analysis. Software log IDs are unique identifiers that developers can use to group and filter log entries. However, we found that, on average, only 21% of logging statements produce IDs, which can lead to loss of information in the log file. We propose LTID, a static analysis approach on log IDs, to remediate the aforementioned issue by extracting a dependency relation between log statements from source code. We build a dependency graph using static analysis and compute the dominance relations of each logging statement. We then propagate IDs to logs that do not contain them based on the dependency graph. We studied 21 well-known Java open-source software subjects and were able to inject IDs on average into 12% of logs without IDs. Through an open coding process, we also establish a categorization, which has a Cohen's Kappa agreement coefficient of 0.74, of the information gained to better understand the relations recovered by the ID propagation process.**

***Index Terms*—Log Mining, Software Artifacts**

I. Introduction

Developers rely heavily on software logs to complete their daily tasks [1]. In addition, software logs keep track of crucial run-time information that developers commonly use to facilitate software testing and development activities, including debugging [2], [3], [4], monitoring software processes [5] and transferring knowledge [6].

With the evolution of hardware and software over the last few decades, software applications have become more complex and large in scale. They are often required to be executed in parallel or distributively, which results in the interweaving of software logs. Such software logs may confuse developers and make the logs more difficult for developers to analyze. Therefore, grouping software logs are needed to facilitate log analysis.

Logs are typically filtered or grouped by identifiers (IDs). A line of software logs consists of static content predefined by developers and dynamic content generated during software run-time. Since an ID is usually specific to the execution context of software logs, it is regarded as dynamic content derived from a dynamic variable in a logging statement. Although there have been extensive prior studies[5], [7] that suggest how to improve logs, to the best of our knowledge, no one has considered IDs as their unique characteristic when compared to other variables enclosed in logging statements. Moreover, a lack of ID or misuse in logging practice could lead to unexpected problems.

Consider the code snippet in Listing 1. A data read operation is performed, and the procedure is logged. The logging statements show that the operation is about a `split`. When the operation starts and ends successfully, the split is logged accordingly. However, if an exception were to be thrown, the logging statement responsible for logging the error does not contain any information that links the error to a `split`. When multiple of these operations are started in parallel, and multiple of them fail, it would be hard or even impossible to correctly attribute the error to a split, as shown in Listing 2. It is not known which `RuntimeException` is thrown by the operation responsible for which `split`.

To address the aforementioned challenges, we conduct an empirical study on the IDs in logging statements, which have never been studied previously. We find that only about 21% of the logging statements contain IDs. Furthermore, the percentage of logs containing IDs diminishes as the logs' verbosity level increases. We also find that, although rare, changes to logs without IDs are more likely throughout historical commits than changes to logs without.

Based on these findings, we propose a simple yet natural approach as the first step in mitigating issues caused by missing IDs. The approach is based on a graph of the relationship between logging statements derived from control flow graphs (CFGs). Then, according to the dominance of each node in the graph, we propagate IDs from logs that contain the IDs to logs that do not. We evaluate the performance of our approach on a method level and successfully propagated IDs. We conclude by categorizing the information gained through propagating the IDs.

This paper's contributions are summarized as follows:

1) We conduct an empirical study on the IDs in logging statements.

Listing 1 An code snippet from Apache Hive, showcasing a logging statement missing an ID.

```
protected Void performDataRead() throws IOException, InterruptedException {
   try {
      ...
      LlapIoImpl.LOG.info("Processing data for {}", split.getPath());
      ...
      LlapIoImpl.LOG.trace("done processing {}", split);
   } catch (Throwable e) {
      LlapIoImpl.LOG.error("Exception while processing", e);
      ...
   }
}
```

Listing 2 A simplified hypothetical output of the snippet in Listing 1.

```
1 Processing data for 'path of a'
2 Processing data for 'path of b'
...
6 Exception while processing RuntimeException: path invalid
...
9 Exception while processing RuntimeException: killed
```

2) We propose a simple approach to inject IDs in logging statements to mitigate related issues.
3) We study the information gained by injecting IDs into logs.

Paper organization. The rest of this paper is organized as follows. Section II discusses related works. Section III establishes and presents the projects subject to our study. We detail our empirical study of the use of IDs in Section IV. Section V presents our approach to propagate IDs. In Section VI, we evaluate our approach and take a deeper look at the propagated IDs. Finally, we discuss threats to the validity of this study in Section VII and conclude in Section VIII.

II. Related Work

In this section, we present the prior research related to this paper.

Use of IDs in log analysis. As opposed to log IDs that appear in the log content, prior studies use log event IDs (also known as log pattern IDs in some research) to aid in log analysis. According to Debnath et al. [8], each log is given a unique ID that includes the log pattern ID and the sequence number of this log compared to other logs in the same pattern. The event log ID is one of the resources used to represent software logs by Liu et al. [9], while He et al. [10] assign a number to each log as its ID for further software log grouping. Nagappan et al. [11] propose an approach to abstract log lines and assign each log line a number in accordance with abstraction, in which the number is similar to the log event ID mentioned in the aforementioned papers. Lin et al. [12] propose an approach for grouping logs and argue that using logs with log event IDs could help them save time and resources. This argument is supported by He et al. [13], who contend that including event IDs in logs is a good logging practice that will aid in future log analysis.

To the best of our knowledge, there are no studies that foster log grouping using software log IDs that appear in log content, which are more direct than event log IDs.

Suggestion or improvement on logging statements. Work that suggests or improves logging statements typically seeks to answer three research questions: where to log, what to log, and how to log. To address the issue of where-to-log, Fu et al. [14] conduct an empirical study on logging practices in the industry to help with a better understanding, and Zhu et al. [15] propose a tool to assist developers in deciding logging position. The resolution of the what-to-log issue aims to improve log content. Yuan et al. [5] alleviate this issue by including additional software diagnostic information in logging statements. Ding et al. [7] propose an approach to automatically generate logging texts using a neural machine, which helps minimize developers' effort to consider what to log. Attempts at answering the question of how to log have the goal of aiding developers in developing and maintaining logging statements more effectively [16]. Chen et al. [16] characterize and detect anti-patterns in logging statements to provide developers with a logging guide. Li et al. [17] offer advice on choosing the log level for a new logging statement, while Tang et al. [18] suggest rejuvenating log levels for existing logging statements. Despite extensive work aiming at improving or suggesting logging statements, none have taken the software log IDs into account to improve software log grouping.

General empirical study on logging. Over the last decade, several empirical studies on software logging have emerged. The first empirical study [19] on software logging was performed in 2012 to understand the situation of logging practice and how developers modify software logs. Since then, a bunch of research has been conducted on logging practices in particular domains. For example, the logging practice was examined in Java-based open-source software projects [20], test code [3], mobile apps [21] and Linux kernel [22], respectively. Furthermore, some empirical studies have emerged using developers' standpoints to examine software logging. For example, Shang et al. [6] conduct an empirical study on email and web search inquiries about software logs to identify development knowledge types that developers frequently seek from the logs,

TABLE I: Summary of studied subjects. The columns *Files*, *SLOC*, *Commits* denote the number of files, number of source lines of code, and number of commits of each repository, respectively.

Subjects	Files	SLOC	Commits
hadoop	14,265	3,981K	26,227
hive	9,913	2,253K	16,404
hbase	5,402	992K	19,466
lucene	5,561	858K	36,203
tomcat	3,358	466K	24,531
activemq	4,892	463K	11,242
pig	2,054	408K	3,711
xmlgraphics-fop	3,055	331K	8,459
logging-log4j2	3,177	253K	12,358
ant	2,139	245K	14,897
struts	2,813	243K	6,406
jmeter	1,793	231K	17,766
karaf	2,250	182K	9,220
zookeeper	1,221	178K	2,435
mahout	1,920	166K	4,506
openmeetings	1,030	138K	3,531
maven	1,915	134K	11,639
pivot	1,219	127K	4,660
empire-db	466	51K	1,474
mina	345	25K	2,401
creadur-rat	244	14K	1,335

and Li et al. [1] study the benefits and costs of logging from developers' perspectives. Along with the aforementioned, an empirical study has been conducted to study logging library migrations for the Apache Software Foundation projects [23] and to study the stability of logging statements [24]. Although there are numerous empirical studies on software logging, none have studied software log IDs to facilitate log grouping, which raises our concern.

III. Study Setup

Our study[1] involves well-known open-source subjects taken from [3] ranging across many domains. They are summarized in Table I.

Each repository is obtained by cloning its source code on GitHub using `git`. The number of files and source lines of code (SLOC) are found using the tool `cloc`[25]. In total, we analyzed 21 projects, all different in scale. The list of projects includes libraries, notably *Hadoop*, *Log4j2* and *ActiveMQ*, software toolings such as *Ant*, *JMeter* and *Maven*, and applications such as *OpenMeetings*, *HBase* and *XMLGraphics-Fop*. Across all projects, the number of source lines of code ranges from 14K to 3,981K SLOC, for a total of more than 11M SLOC. All projects have a long history of commits, the shortest at 1K from the head commit and the longest at 26K.

We use srcML [26] to parse the java source files. SrcML is an XML format representing source code and a toolkit to convert source code to the srcML format. This allows us to perform queries on the structure of the code easily.

[1]The replication package is available at https://github.com/senseconcordia/logtracker

IV. Identifiers in Logs

Log entries often refer to a specific object, such as tasks and threads, or a piece of data. A simpler relation can be established between entries if they can be grouped. Log groupings help filter logs to reduce the complexity of understanding the process in a running instance that generates logs. Traditionally, this grouping is performed by following a simple rule: entries containing the same value of a specific identifier are related and should be grouped.

In this section, we discuss identifiers found in logging statements. We first establish our definition of *identifiers* and a heuristic to distinguish them from other variables in Section IV-A. Then, we study their presence in source codes and changes affecting them in historical commits in Section IV-C.

A. Establishing Identifiers

We define an identifier as a variable that can uniquely identify an object. An object could be a resource, a unit of data, a process, etc. Identifiers can be produced by default by the logging framework used by the program. An example of such identifiers is `thread ID`. The thread ID is particularly useful in analyzing logs of multi-threaded applications: grouping by thread ID can reduce the complexity of a log sequence and allows for the analysis of critical paths. Logs of finer and coarser execution units can also be grouped using identifiers referring to tasks, jobs, processes, etc.

After investigating the 21 Java projects, we found that objects referred to by IDs in logging statements can be classified into the following two categories:

Resources are units of data that the program processes. They can be obtained as inputs or computed at runtime. A unit of data can be, for example, rows of tables in a database, network packets, files, results of other data transformations, etc. Resources can also be resource handles, that is, indirect references to a resource indicating the location where the resource can be found and accessed. For example, sockets, virtual filesystem objects, network sessions, etc. Identifiers of these resources are used not only to keep track of them but also to relate them to each other to produce new data. Identifiers of resources are included in logs to record the data flow and to join information. Identifiers in this category can be location indicators such as URLs, IP addresses and indices, or a representative part of the data, such as hashes and names.

Computation Units are actors that produce and consume resources. Examples of computation units are threads, coroutines, and processes. Computation units can also be an abstract representation of actions such as jobs and tasks. Indirect references to external computation units, such as web jobs and requests. Identifiers of computation units are names or integers that can be encoded into other representations. Sometimes, the identifier is part of the computation, such as when it is used to determine a runtime behaviour. Thus they are essential in keeping track of the progress and the behaviour of the computation.

Although different, objects of these categories often refer to each other. For example, a network request is often related to a resource being accessed. We also note that they can be nested and heterogenous, thus adding layers of complexity to their relationships. Piecing together information from objects of both categories is essential in software analysis, and the persistence of that information is the ultimate goal of logging.

Because objects referenced by IDs are so different from each other but also interrelated, it is almost impossible to provide an accurate template to distinguish IDs just from the nature of what the variable reference and without data flow analysis.

Instead, to distinguish identifiers from normal variables, we use a heuristic about the name of the variables. Variable names can be easily accessed while reading the source code, but we note that, as with all heuristics, it comes at a cost to the accuracy. To formulate this heuristic, we sample the logging statements that contain variables via a traversal of the abstract syntax tree (AST) of source code and extract every logging statement. Similarly to the method used in prior research [3], we employ the following set of rules to pick out logging statements from the AST:

1) The AST element must be an invocation statement. This rule follows the fact that logging is necessarily performed via a function call.
2) The qualified name of the target of the invocation must contain the word "log". This rule follows that logging functions are usually instance methods in a dedicated class that handles related logging logic. The instances of that class are usually in a similarly named variable that refers semantically to logging. This is necessary to distinguish invocations of methods that have similar signatures to logging methods but do not log, such as `Math.log()`. In this last example, the qualified name of the target is `Math` and thus is not considered a logging statement.
3) The method invoked by the statement must have as a name either "info", "warning", "error", "fatal", "debug", "trace" or "log". This rule follows that they are the most common names for logging methods.

Then, after extracting all logging statements and sampling a subset of them, two authors manually and separately labelled all variable access and method call used as arguments in each logging statement of the sample to determine whether the values they could take could uniquely identify an object. We focused on the semantics of the names. Based on this labelling, we formulate the following heuristic applicable to the names found in the arguments: if a part of a name contains a keyword, then the logging statement is considered to have an identifier. The keywords are chosen such that it matches our labelling while also keeping the list short. The list of keywords we end up with is "ID", "IP", "name", "path", "URL", "URI". This is a very conservative list, but it ensures that the results are consistent across projects.

TABLE II: Number of log statements found in the studied Java projects, and the fraction of which containing IDs.

Subject	# Log	# Log w/ IDs	(%)
hadoop	18,962	5,076	(27%)
hive	10,854	2,454	(23%)
hbase	10,223	2,524	(25%)
lucene	6	1	(17%)
tomcat	2,830	1,095	(39%)
activemq	6,341	1,099	(17%)
pig	1,079	127	(12%)
xmlgraphics-fop	1,104	135	(12%)
logging-log4j2	2,902	504	(17%)
ant	32	6	(19%)
struts	1,332	388	(29%)
jmeter	2,221	537	(24%)
karaf	652	147	(23%)
zookeeper	2,226	484	(22%)
mahout	588	68	(12%)
openmeetings	658	148	(22%)
maven	331	67	(20%)
pivot	0	0	
empire-db	762	290	(38%)
mina	227	5	(2%)
creadur-rat	13	0	(0%)

B. Finding Identifiers in Source Code

An example of an identifier commonly used in groupings is "request ID", where a developer might be interested in the process started by a specific client request to a server. The developer can then interpolate the events that occurred in response to the request from the log group. However, identifiers are not always included in logging statements, intentionally or not. In response to the request, a server might initiate tasks whose logging statements do not include the request's ID. Then, the developer needs to perform additional operations to navigate the log file. Worst, subtasks might also be started from the main tasks and logs in log files become interleaved. In this scenario, it becomes difficult for a developer or even automated methods to relate the events of different tasks initiated by the same request.

Table II concretizes the problem. Using the previously defined heuristic in Section IV-A, we filtered and count the logging statements with IDs and without IDs. In all analyzed projects, we found that, on average, only 21% of the logging statements contain IDs. In Tomcat, 39% are logging statements with ID, the highest fraction across all projects. The projects with the least number of logging statements with IDs, without counting the outliers, are Mahout and XMLGraphics-Fop, at 12%. There are three outliers, Pivot, Mina and Creadur-Rat. No logs were found for Pivot, and no logs with IDs were found for Creadur-Rat. Mina has an exceptionally low count of logs with IDs. For these Creadur-Rat and Mina, most variable names used for their internal objects are shorter than descriptive and are not prefixed nor suffixed with any modifier keywords, in which case our heuristics fail. In the case of Tomcat, a large portion of the project failed to compile using our tooling.

We also observe low correlation between the number of logs and the fraction containing IDs. The Pearson product-moment

TABLE III: Number of log statements and log statements with IDs in each verbosity level.

Level	# logs	# logs w/ ID	(%)
other	299	30	(10%)
trace	2,578	757	(29%)
debug	1,633	4,687	(29%)
into	23,167	5,459	(24%)
warn	9,306	2,336	(25%)
error	11,572	1,891	(16%)
fatal	105	4	(4%)

correlation coefficient between them is only a little over 0.3. This could be problematic for larger projects, such as Hadoop, Hive and HBase. As the number of logging statements in a project grows, more information is recorded in log files. However, since the ratio of logs with ID does not grow, the information contained in the log files becomes hard to parse: logs could be separated by longer sequences of unrelated logs, and more interleaving could happen.

Table III shows the proportion of logging statements with IDs in each verbosity level. The largest percentage is in the `trace` level, and the lowest is in the `fatal` level. We notice that as the verbosity increases, the percentage drops. This is confirmed by a Pearson correlation coefficient of -0.9. This is problematic in debugging situations where high-verbosity logs are usually looked at first to identify problems. The `other` level includes logging statements where we couldn't find the appropriate level.

Finding 1: On average, only 21% of logging statements contain IDs. Furthermore, as the verbosity of the logging statement increases, this percentage diminishes. Combined, this could result in log files that are hard to follow.

C. Changes in Logs in Historical Commits

As with any other source code, logging statements evolve as a project grows. We look at the historical commits to understand how IDs in logging statements change.

Table IV shows the average number of logging statements that changes per commit. Old changes are deletions of a logging statement, new changes are additions, and revision changes are modifications. Modifications can split a single logging statement into multiple or merge a multiple into a single logging statement. Hadoop has the most changes to logging statements per commit, with 2.92 deletions, 1.69 additions and 0.36 modifications per commit. We note that the percentages of changes to logs with ID in commits are lower than the percentages of logs with ID, shown in Table II, which suggests that logs with IDs are more likely than changes to those without. Those changes could mean a high variability in log files: IDs could be in some versions and not in others.

There are even fewer logs with more than one ID. Table V shows that the average number of logs with more than one ID is only 2.39%, and the average number of logs with more than two IDs is only 0.36%.

Table VI shows the number of commits that contain at least one change to logs with an ID compared to those that also have changes to logs without IDs. We observe that only 4% of commits actually change logs with IDs. We also observe that in between those commits, many others do not affect logs at all, thus indicating that changes to logs with IDs rarely occur.

Finding 2: Changes to logging statements containing IDs are less common than changes to logging statements without ID. On average, only 17% of changed logging statements contain IDs.

V. LTID: Attaching Identifiers to Logging Statements

We propose LTID, an approach to extract the dependency relations of logging statements, with which we construct a log graph used to propagate IDs from logs that contain them to those that do not.

A. Overview

Our approach (Figure 1) leverages log graphs, i.e. pruned CFGs, and ID variables in log templates to identify the dependency relation between pairs of log entries. It first takes the source files as input to extract the corresponding log templates and variables (step 1) and build CFGs (step 3). It then generates log graphs derived from CFGs by eliminating irrelevant and non-logging nodes (step 4) and computes for each the dominance between pairs of nodes (step 5). Finally, to concretize the dependencies between pairs of log entries, based on the log graph and the computed dominance, it employs a set of ID variable names obtained by applying a heuristic (step 2) to propagate the ID variables of each logging statement (step 5).

B. Log Graph Construction

a) Extracting Logging Templates: Given source code files, our approach traverses the abstract syntax tree (AST) of each method to identify all logging statements using a keyword based search strategy commonly used in prior studies [3], [20]. We extract logging statements as described in Section IV-A. Specifically, logging statements are chosen based on the following rules: the statement is an invocation whose target's qualified name contains the word "log" or words associated with the logging level, such as "info", "error" and "warning". We also require that all chosen logging statements have an argument of `String` type, which could improve search strategy accuracy by filtering out some methods with names that include "log" but are not logging statements, such as `Math.log(Double double)`.

Once logging statements are given, we can extract log templates and variables from their arguments. Log templates are texts with generated variable names inserted in place of the dynamic content. The variable names are used to find variables that are IDs for identifying the dependency relations between log entries in a log file, while log templates are used to match

TABLE IV: Average number of logging statement changes per commit, and the proportion of which contain IDs. "Old" is deletion, "New" is addition and "Rev" is a modification of logging statements.

Subject	Old	Old w/ ID	(%)	New	New w/ ID	(%)	Revision	Rev w/ ID	(%)
hadoop	2.92	0.45	(15.29%)	1.69	0.26	(15.42%)	0.36	0.12	(34.46%)
hive	3.30	0.31	(9.26%)	1.37	0.12	(9.10%)	0.20	0.05	(25.18%)
hbase	1.74	0.18	(10.19%)	1.17	0.12	(10.29%)	0.26	0.05	(20.25%)
lucene	0.60	0.08	(13.37%)	0.56	0.07	(12.10%)	0.03	0.01	(22.43%)
tomcat	0.18	0.03	(14.86%)	0.15	0.02	(14.10%)	0.04	0.01	(27.95%)
activemq	1.52	0.20	(13.26%)	0.90	0.12	(13.33%)	0.05	0.02	(37.08%)
pig	0.65	0.03	(5.30%)	0.33	0.01	(4.00%)	0.03	0.00	(11.82%)
xmlgraphics-fop	0.87	0.11	(12.81%)	0.74	0.10	(13.02%)	0.06	0.01	(17.99%)
logging-log4j2	0.77	0.09	(12.30%)	0.48	0.06	(13.53%)	0.02	0.01	(39.87%)
ant	0.03	0.00	(12.29%)	0.01	0.00	(37.67%)	0.01	0.00	(13.19%)
struts	1.09	0.14	(12.50%)	0.85	0.10	(11.43%)	0.02	0.01	(31.41%)
jmeter	0.50	0.06	(12.27%)	0.40	0.05	(12.29%)	0.02	0.00	(16.71%)
karaf	0.24	0.04	(16.41%)	0.17	0.03	(18.22%)	0.01	0.00	(55.24%)
zookeeper	2.66	0.45	(16.78%)	1.99	0.33	(16.74%)	0.21	0.09	(41.63%)
mahout	1.01	0.08	(7.61%)	0.93	0.07	(7.52%)	0.06	0.01	(17.83%)
openmeetings	0.50	0.10	(19.72%)	0.64	0.09	(13.40%)	0.04	0.03	(68.82%)
maven	0.10	0.01	(7.58%)	0.09	0.01	(8.85%)	0.01	0.00	(3.45%)
pivot	0.00	0.00		0.00	0.00		0.00	0.00	
empire-db	1.20	0.10	(8.10%)	0.79	0.06	(8.07%)	0.00	0.00	
mina	0.78	0.03	(3.86%)	0.69	0.02	(3.54%)	0.07	0.01	(18.99%)
creadur-rat	0.02	0.00	(0.00%)	0.01	0.00	(0.00%)	0.00	0.00	

TABLE V: Average percentage of logs with at least one, two and three IDs.

logs w/ $\geq$ 1 ID	logs w/ $\geq$ 2 ID	logs w/ $\geq$ 3 ID
20%	2.39%	0.36%

TABLE VI: Number of commits with changes to logs.

# commits	# commits w/ ID	(%)
238,775	9,449	(4%)

the log entries with nodes in the log graph. Both log templates and variables could be directly extracted from the arguments of logging statements.

b) ID Identification: IDs in software logs are unique identifiers that can be used to group software logs. The software logs with identical IDs are related to one another and are commonly specific to a single system control flow, which suggests that the same system execution typically produces logs with Identical IDs. Therefore, learning about software IDs can aid developers in comprehending the relationships between log entries.

From the extracted logging variables, we identify variables that represent IDs by establishing a heuristic which we define in Section IV-A.

c) CFG Construction: Traditionally, CFGs are constructed using specific intermediate representation (IR) or compiled binaries, such as JVM's `.class` files, which lay emphasis on storing and representing the flow relationships in programs. According to a prior study, [27], CFG construction could be accomplished using static, dynamic, or hybrid strategies. However, as they grow in size and complexity, modern software systems typically result in large and complicated CFGs that require significant time and resources to construct using dynamic or hybrid strategies. Furthermore, CFGs may contain a lot of low-level information [28] in which we have no interest in this study. Therefore, in this study, we build CFGs using ASTs that store higher-level information without executing programs.

The goal of constructing CFGs is to determine the control flow relations between logging statements. To this effect, we present the CFG nodes using AST nodes at the statement level rather than the block level since using the block level would require additional work to locate the logging statements. We construct the CFG of each method using Spoon [29].

Although not used in this study, it is possible to connect the CFGs of every method to form an interprocedural CFG (ICFG). While preserving the various method invocation contexts, we substitute the CFG node for each invocation statement we encounter during the construction by a copy of the CFG of the invoked method. Because of dynamic dispatches, constructing this ICFG requires a data flow analysis to resolve the types and method invocations. Sometimes it is not easy to get a proper resolution. For example, when the method `Object.equals(Object other)` is invoked recursively; it is close to impossible to resolve the type of the `other` object since this method is used often and invoked in many places in the code base. Recursion also needs to be taken into account. Determining if the recursion depth using static analysis is non-deterministic. It requires symbolic execution of the program, which might never halt. If recursion occurs, we can replace the recursive invocation node with an edge to the previous instance of this invocation, effectively approximating the recursive behaviours with a loop. Furthermore, to facilitate the construction process and conserve resources, this construction can be done lazily only to take up as much memory as required.

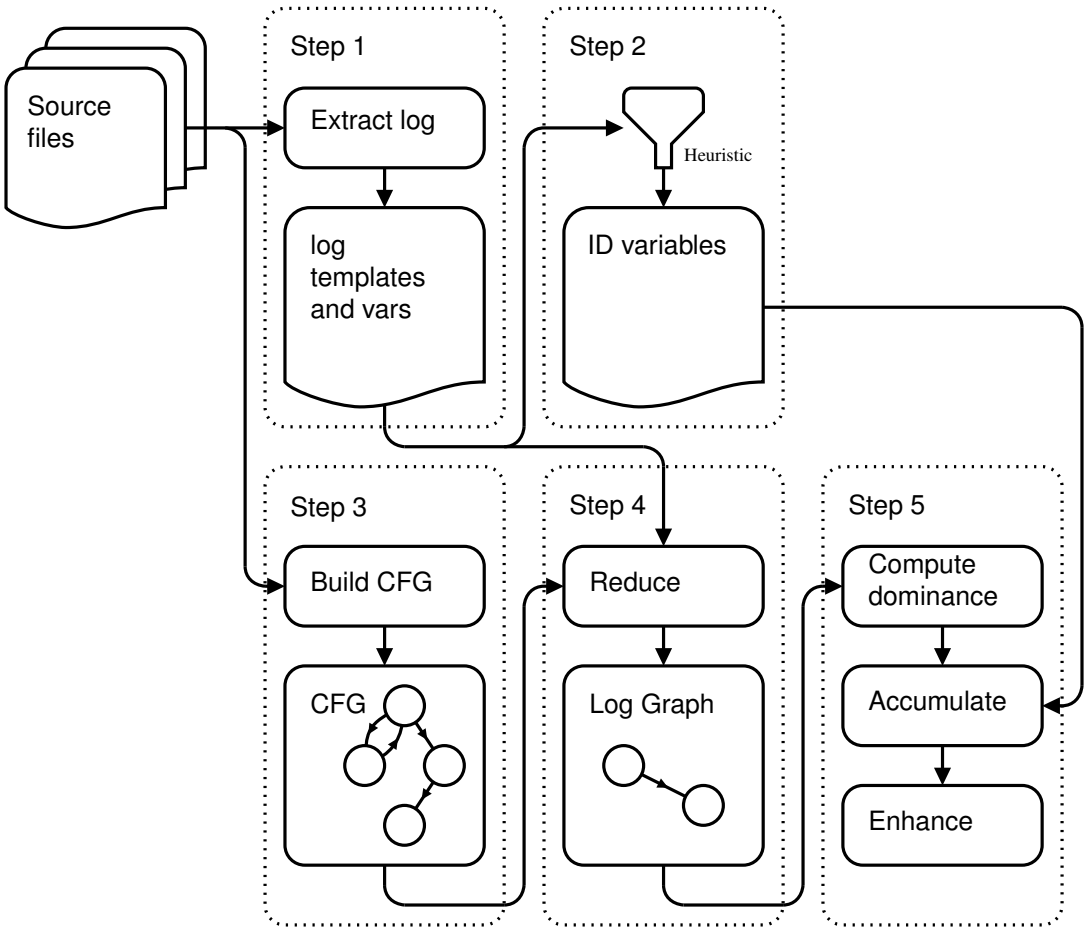

Fig. 1: Process of building a log graph.

d) Reducing the CFGs: Given that we are only interested in the control flow relationships between logging statements, the CFG now encompasses logging statement nodes and non-logging nodes, so we need to filter those unrelated nodes out. For each node we are not interested in, we remove it by discarding its incoming edges from its predecessors; we then make an edge from each predecessor to each of the removed node's successors. Extra care is taken when eliminating nodes that have self-edges due to loops and recursion by removing empty cycles that might arise in the process.

We call this resulting reduced CFG the log graph. Each vertex in the graph represents a logging statement. In addition, each directed edge indicates a dependency relationship between the log entries at the ends of the edge. In other words, a log at the destination end of an edge depends on the log at the origin end of the edge.

Formally, let $T := \{t_1, t_2, ..., t_k\}$ be the set of all log entry types appearing in a log file F, and t_i and t_j be two log entry types in T. There is a dependency relation between t_i and t_j, if t_i must always appear before t_j. We assume that when testing the order of t_i and t_j, they are put into the log file synchronously. We denote this relation as $t_i > t_j$. The log graph is the graph $G = (T, R)$ where $R := \{(t_i, t_j) \in T^2 : t_i > t_j\}$.

Algorithm 1 Algorithm to propagate logs using the dominator tree as described in prior research [30]

```
procedure FindIdentifiers(v)
    x ← Variables(v)
    x ← Heuristic(x)
    if Dominator(v) exists then
        u ← Dominator(v)
        y ← FindIdentifiers(u)
        x ← x ∪ y
        return x
    else
        return x
```

e) Computing Dominance: Dominance is then computed using the method described in [30]. Finally, we compute the dominator tree from the CFG. Algorithm 1 shows how the algorithm is used.

We then define the dependency relation between log entries using the computed dominance. Let $T = \{t_1, t_2, \ldots, t_k\}$ be the set of log entry types. Let $s_1, s_2, \ldots, s_k$ be the logging statement nodes in the log graph, with each generating type

$t_1, t_2, \ldots, t_k$ respectively. If s_i strictly dominates s_j for some integer i and j between 0 and k, then all entries of type t_j depends on t_i. We denote this relation as $t_i > t_j$.

We use Algorithm 1 to find identifiers given a log statement v. It first checks if v has itself some variables x that are IDs, using the procedure VARIABLES and HEURISTIC. Then, it uses the dominator u or not. If it does, then it recurses to find the identifiers y of u and returns the union of x and y. This procedure recurses until the root is reached, which has no dominators. In our case, it is the first statement of the method.

At the end of the procedure, the returned state x contains both v's and its dominators' IDs. If the log statement v does not have IDs, then x will be empty. Thus, the procedure returns only the results of finding IDs in its dominator.

VI. STUDYING THE ADDED ID BY LTID

There are many reasons why a logging statement might not contain an ID: it might not be practical due to the logging statement being reused in different contexts, it might not make sense in the semantic context of the logging statement for it to have an ID, or simply the developer did not think it was necessary. However, including an ID is important to ensure that no information is lost in a log file.

To alleviate the lack of IDs, we use the log graph described in Section V. Given logging statements with ID, we can propagate their IDs to statements without ID. This propagation is performed for all pairs of logging statements in the log graph that are neighbors. That is, for any pair of log event types s_i and s_j in the same log graph and their corresponding types t_i and t_j, if $t_i > t_j$ then we append all IDs of s_i to s_j if it is not already present in s_j. Similarly, although not in the scope of this study, IDs can also be propagated between log event instances in log files. In that case, the log graph can be approximated using temporal dependencies, e.g., using similar methods to the one proposed in [31].

We first evaluate our approach in Section VI-A. Then we propose categorizing the information gained through the propagation of IDs in Section VI-C.

A. Evaluating the Static Log Graph

Table VII presents the number of logs with successful ID injections compared to the number of logs without any ID. Aside from the Lucene, Ant, and Pivot, where no injections could be made, and Karaf and Logging-Log4j2, where they have an exceedingly high number of injections, the lowest percentage of successful injections is 2% in Tomcat, and the highest is 13% in Pig. On average, only 12% of logging statements were successfully injected with an ID.

We note that there is little to no correlation between the number of logging statements without IDs and the percentage with IDs. The correlation coefficient is 0.4, which is not significant enough to indicate any correlation.

Table VIII shows a different view of the number of logs with successful ID injections compared to the number of logs without any ID. Considering all projects, logs of the `other` and `fatal` levels had the highest percentage of logs with successful injections, with 76% and 47%, respectively. In contrast to the low percentage of logs with IDs of the `fatal` level, we find that almost half of them can have IDs injected. For other log levels, the percentage of injected logs is lower. Excluding `fatal` and `other`, we find that, on average, 11% of logs can have injected IDs. In the most common level, `info`, the percentage is lowest, at only 9%.

There is a significant variation from 0 to 96% across all projects. This is largely due to how projects fit into the scope of LTID. Since the analysis is performed at a method level, differing logging practices of the projects may cause varying results. For example, in Hadoop, many methods only contain one logging statement, in which case our method can not inject any IDs. In contrast, in Log4j2, many methods with logging statements have one at the beginning and another further down the control flow in an error-handling block.

TABLE VII: Number of logging statements in each project without any IDs compared with those of which an ID can be injected.

Subjects	# logs w/o IDs	# injections	(%)
hadoop	13,886	968	(7%)
hive	8,400	546	(7%)
hbase	7,699	468	(6%)
lucene	5	0	(0%)
tomcat	1,735	39	(2%)
activemq	5,242	247	(5%)
pig	952	127	(13%)
xmlgraphics-fop	969	20	(2%)
logging-log4j2	2,398	2,084	(87%)
ant	26	0	(0%)
struts	944	72	(8%)
jmeter	1,684	96	(6%)
karaf	505	485	(96%)
zookeeper	1,742	167	(10%)
mahout	520	15	(3%)
openmeetings	510	30	(6%)
maven	264	12	(5%)
pivot	0	0	
empire-db	472	27	(6%)
mina	222	0	(0%)
creadur-rat	13	0	(0%)

TABLE VIII: Number of logging statements of different log levels without ID compared with those of which an ID can be injected.

Level	# logs w/o IDs	# injections	(%)
other	269	205	(76%)
trace	1,821	266	(14%)
debug	11,646	1,484	(13%)
info	17,717	1,623	(9%)
warn	6,970	773	(11%)
error	9,681	922	(10%)
fatal	101	48	(47%)

B. Example Propagation of ID

Listing 3 shows an example of where injection of IDs can happen and might be helpful. There are two logging statements in the code snippet. The first one logs an action on some data identified using the variables `hsa` and `hri` defined earlier in

Listing 3 Code snippet is taken from HBase. An ID is propagated from the first logging statement to the second.

```
ServerName hsa = metaLocation.getServerName();
RegionInfo hri = metaLocation.getRegion();
...
LOG.info("deleting hdfs data: " + hri.toString() + hsa.toString());
...
Path p = new Path(rootDir + "/" + TableName.META_TABLE_NAME.getNameAsString(),
    hri.getEncodedName());
boolean success = fs.delete(p, true);
...
LOG.info("Deleted " + p + " sucessfully? " + success); // `hsa` can be injected here.
```

the program. The second one logs the result of the previous action. This time, it is only identified by the `hri` variable, included indirectly through the `p` variable. If we consider a grouping by the server name indicated by the `hsa` variable, the second log would not be included in the group. Furthermore, in cases where this code is executed by many parallel tasks acting on data from the same region, indicated by the `hri` variable, it becomes hard or even impossible to attribute the result of the action to the data on which it's acting.

In the same snippet, our approach successfully detects the potentially missing ID. Then, based on this detection, it can inject the ID into the logging statement. For demonstration, a comment is injected instead. The processed second log contains enough information to be related to the first one, and the relationship is preserved in log files. Even if executed by multiple parallel tasks, each result is correctly attributed to the action.

Finding 3: Using our approach, we were able to inject IDs into logs that had no IDs. We find that, on average, 12% of logs without IDs can have injected IDs.

C. Categorizing the Information Gain

By injecting IDs into the logging statements, we are essentially concretizing their dependency relation. In log filtering and grouping tasks, more logs are included. To better understand this informational gain, we propose a categorization based on a manual study. We sampled 378 instances of injection of IDs in logging statements across all twenty-one studied projects. Then all of the authors of this paper followed an open coding process to identify the categories.

The manual study was completed by four professional computer science researchers: two with Ph.D. degrees, one pursuing a Ph.D., and one visiting senior computer science student. The process is twofold. First, we each analyze the methods in the sample individually and consolidate our findings into a common set of categories. Second, we perform again, individually, two rounds of labelling the methods while considering the propagations of IDs into the established categories from the first step. In this second process, we have achieved a reasonably strong agreement with Cohen's kappa coefficient of 0.74.

The categories we identified are as follows. We present a short example of each category to help explain their differences.

Sequence (Listing 4) describes a linear relationship between two or more logs. Unlike branches, information gained in this category indicates the progress of a process. This kind of relationship does not implicitly reveal any runtime values, that is, those that are not explicitly added to the logging statement. However, they contain evidence that no error has happened in between.

Listing 4 Example of a sequence

```
Log.info('started task {}', id)
...
// inject id
Log.info('task completed')
```

Branch (Listing 5) describes a change in the runtime state about the branching of the control flow of a process. The conveyed relationship is nonlinear and non-deterministic. The log that will be produced to the logfile will depend on some runtime conditions. Like the sequence, it can reveal whether an exception has happened.

Listing 5 Example of a branch.

```
Log.info('started task {}', id)
...
if (condition) {
   // inject id
   Log.info('started subtask {}', sub_id)
}
```

Exception (Listing 6) describes an exceptional branching of an otherwise linear flow. It differs from branches in that it conveys unexpected behaviour. Although the exception is a form of control flow, we differentiate exceptions from branching because it informs that there is a shift in the state of the program into a different flow in an attempt to recover from an error. In the example in Listing 6, we remark that an exception relationship does not require the use of a `try catch` statement.

Listing 6 Example of an exception.

```
Log.info('started task {}', id)
...
switch (result) {
   case SUCCESS: ...
   case ERROR
      // inject id
      Log.error('got {}', result)
      throw new RuntimeException()
}
```

We also notice that in some cases, multiple logging statements are used in succession to record closely related information. In those cases, we can also classify them as follow:

Elaboration (Listing 7) describes an addition of optional details. The information contained in the first log is sufficient by itself, and the second is not, contains only auxiliary information on the runtime state and serves as a supplement to the first.

Listing 7 Example of an elaboration.

```
Log.info('started task {}', id)
// inject id
Log.info('might take a long time')
```

Explanation (Listing 8) describes the addition of ancillary details to the information contained in the first log. Usually, it explains the subject of the first log. The second log, which adds information, is necessary to convey the information correctly.

Listing 8 Example of an explanation.

```
Log.error('task {} failed', id)
// inject id
Log.debug('task received {}', signal)
```

Complement (Listing 9) describes a pairing of two different sufficient pieces of related information that are not about the same subject. The two logs can be seen as two different events that happen simultaneously, whereas, in elaborations and explanations, only one event is described.

Listing 9 Example of a complement.

```
Log.info('started task {}', id)
// inject id
Log.debug('using threads {}', thread_count)
```

Finding 4: We identified six categories of the dependency relations between logging statements without IDs and their referential logging statements with IDs: Sequence, Branch, Exception, Elaboration, Explanation, and Complement.

VII. Threat to Validity

A. External Validity

The 21 subjects studied are all open-source Java projects developed under the Apache Software Foundation. Our results could be biased towards software in the writing style found in Apache software and might not apply to projects maintained by other organizations. Differences in logging practices can be different.

The subjects are all written in Java. For this reason, the tools we have developed, the techniques we used and our results may not apply to projects using other languages such as C++, Python, Go etc.

However, even if we only considered Java projects, our results are still significant and may serve as a basis for future work. Java is a well-understood simple language and have overlapping features with many other popular languages. These projects are well-established and have been maintained by professional teams. Furthermore, they have been studied extensively in previous research [3], [20].

B. Internal Validity

To differentiate IDs from normal variables, we used a heuristic defined after a preliminary round of manual labelling. Our heuristic is a generalization of the results of the manual labelling and might impact the accuracy of finding IDs. IDs might be missed, or non-IDs might be erroneously considered as IDs. However, the preliminary labelling is performed by two developers to avoid ambiguities and uncertainty. We chose many keywords closely related to our subjects to reduce the chances of mislabelling the variables.

To find the dependency relations of logs using CFGs, sometimes approximations are required due to the dynamic behaviour of a program. Common language features that make static analysis difficult are dynamic dispatch, introspection, and runtime exceptions. To workaround these issues, we scope our analysis at the method level, where the construction of CFGs is simple and does not require data flow analysis. This method gets us close to the ground truth and allows us to set a baseline for future work.

C. Construct Validity

Our approach only aims to complement logging statements by preserving dependency relations in between. Thus, an evaluation of the approach can only be subjective.

However, although we find that the added information is useful, it might not always be the case, depending on the differing contexts in real-world scenarios. A large user study can be conducted to investigate the usefulness and quality of the gained information.

VIII. Conclusion

We conducted an empirical study on the IDs in logging statements of 21 Java open-source projects. To our knowledge, this is the first study that focuses on the uses of IDs in logs. We found that only a limited amount of logging statements contain IDs and that their proportion is even lower in high-verbosity logs. We then proposed a simple approach to injecting IDs in logging statements to mitigate this issue. The approach is based on control flow graphs obtained through static analysis of the source code. We found that although capable of propagating IDs to logs that do not contain any, our results only establish a baseline. We also study the information gained by injecting IDs which could prove helpful in future studies.

Our approach is based on static analysis, which is highly dependent on the language and tools available. It does not make use of the rich information produced at runtime. In future work, we aim to improve the performance and generalizability of our approach by mining log dependencies directly from log files.

REFERENCES

[1] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," **IEEE Trans. Software Eng.**, vol. 47, no. 12, pp. 2858–2873, 2021.

[2] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "Leveraging performance counters and execution logs to diagnose Memory-Related performance issues," in **2013 IEEE International Conference on Software Maintenance**, Sep. 2013, pp. 110–119.

[3] H. Zhang, Y. Tang, M. Lamothe, H. Li, and W. Shang, "Studying logging practice in test code," **Empirical Software Engineering**, vol. 27, no. 4, p. 83, Apr. 2022.

[4] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting Large-Scale system problems by mining console logs," in **Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles**, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 117–132.

[5] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in **Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems**, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, pp. 3–14.

[6] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge," in **2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2014, pp. 21–30.

[7] Z. Ding, H. Li, and W. Shang, "LoGenText: Automatically generating logging texts using neural machine translation," in **2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)**. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2022, pp. 349–360.

[8] B. Debnath, M. Solaimani, M. A. Gulzar, N. Arora, C. Lumezanu, J. Xu, B. Zong, H. Zhang, G. Jiang, and L. Khan, "LogLens: A Real-Time log analysis system," in **38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018**. IEEE Computer Society, 2018, pp. 1052–1062.

[9] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, "Logzip: Extracting hidden structures via iterative clustering for log compression," in **Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering**, ser. ASE '19. San Diego, California: IEEE Press, 2019, pp. 863–873.

[10] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in **2017 IEEE International Conference on Web Services (ICWS)**, Jun. 2017, pp. 33–40.

[11] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in **2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)**. IEEE, May 2010.

[12] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in **Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume**, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 102–111.

[13] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in **2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**, Jun. 2016, pp. 654–661.

[14] Q. Fu, J. Zhu, W. Hu, J.-G. 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**, ser. ICSE Companion 2014. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 24–33.

[15] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in **Proceedings of the 37th International Conference on Software Engineering - Volume 1**, ser. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 415–425.

[16] B. Chen and Z. M. Jiang, "Characterizing and detecting Anti-Patterns in the logging code," in **2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)**, May 2017, pp. 71–81.

[17] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" **Empir. Softw. Eng.**, vol. 22, no. 4, pp. 1684–1716, 2017.

[18] Y. Tang, A. Spektor, R. Khatchadourian, and M. Bagherzadeh, "Automated evolution of feature logging statement levels using git histories and degree of interest," **Science of Computer Programming**, vol. 214, p. 102724, 2022.

[19] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in **34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland**, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 102–112.

[20] B. Chen and Z. M. (Jack) Jiang, "Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation," **Empirical Software Engineering**, vol. 22, no. 1, pp. 330–374, Feb. 2017.

[21] Y. Zeng, J. Chen, W. Shang, and T.-H. p. Chen, "Studying the characteristics of logging practices in mobile apps: a case study on F-Droid," **Empir. Softw. Eng.**, vol. 24, no. 6, pp. 3394–3434, 2019.

[22] K. Patel, J. Faccin, A. Hamou-Lhadj, and I. Nunes, "The sense of logging in the linux kernel," **Empir. Softw. Eng.**, vol. 27, no. 6, pp. 1–44, 2022.

[23] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: a case study for the apache software foundation projects," in **Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016**, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 154–164.

[24] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements," **Empir. Softw. Eng.**, vol. 23, no. 1, pp. 290–333, 2018.

[25] "cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages."

[26] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in **Proceedings of the 2013 IEEE International Conference on Software Maintenance**, ser. ICSM '13. USA: IEEE Computer Society, Sep. 2013, pp. 516–519.

[27] K. Zhu, Y. Lu, H. Huang, L. Yu, and J. Zhao, "Constructing more complete control flow graphs utilizing directed Gray-Box fuzzing," **NATO Adv. Sci. Inst. Ser. E Appl. Sci.**, vol. 11, no. 3, p. 1351, Feb. 2021.

[28] G. Tan and T. Jaeger, "CFG construction soundness in Control-Flow integrity," in **Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security**, ser. PLAS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 3–13.

[29] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: A library for implementing analyses and transformations of java source code," **Softw. Pract. Exp.**, vol. 46, no. 9, pp. 1155–1179, Sep. 2016.

[30] L. Georgiadis, R. E. Tarjan, and R. F. Werneck, "Finding dominators in practice," in **European Symposium on Algorithms**, vol. 10. unknown, Jan. 2006, pp. 69–94.

[31] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "CloudSeer: Workflow monitoring of cloud infrastructures via interleaved logs," **SIGPLAN Not.**, vol. 51, no. 4, pp. 489–502, Mar. 2016.