Towards Log Slicing

Joshua Heneage Dawes[1 [0000−0002−2289−1620]], Donghwan Shin[1,2 [0000−0002−0840−6449]], and Domenico Bianculli[1 [0000−0002−4854−685X]]

[1] University of Luxembourg, Luxembourg, Luxembourg
{joshua.dawes,domenico.bianculli}@uni.lu
[2] University of Sheffield, Sheffield, United Kingdom
d.shin@sheffield.ac.uk

Abstract. This short paper takes initial steps towards developing a novel approach, called *log slicing*, that aims to answer a practical question in the field of log analysis: *Can we automatically identify log messages related to a specific message (e.g., an error message)?* The basic idea behind log slicing is that we can consider how different log messages are "computationally related" to each other by looking at the corresponding logging statements in the source code. These logging statements are identified by 1) computing a backwards program slice, using as criterion the logging statement that generated a *problematic* log message; and 2) extending that slice to include *relevant* logging statements.
The paper presents a problem definition of log slicing, describes an initial approach for log slicing, and discusses a key open issue that can lead towards new research directions.

Keywords: Log · Program Analysis · Static Slicing.

1 Introduction

When debugging failures in software systems of various scales, the logs generated by executions of those systems are invaluable [5]. For example, given an error message recorded in a log, an engineer can diagnose the system by reviewing log messages recorded before the error occurred. However, the sheer volume of the logs (e.g., 50 GB/h [9]) makes it infeasible to review all of the log messages. Considering that not all log messages are necessarily related to each other, in this paper we lay the foundations for answering the following question: *can we automatically identify log messages related to a specific message (e.g., an error message)?*

A similar question for programs is already addressed by *program slicing* [2, 14]. Using this approach, given a program composed of multiple program statements and variables, we can identify a set of program statements (i.e., a program slice) that affect the computation of specific program variables (at specific positions in the source code).

Inspired by program slicing, in this paper we take initial steps towards developing a novel approach, called *log slicing*. We also highlight a key issue to be addressed by further research. Once this issue has been addressed, we expect *log slicing* to be able to identify the log messages related to a given *problematic* log message by using static analysis of the code that generated the log. Further, since we will be using static analysis

```
(1) logger.info("check memory status: %s" % mem.status)
(2) db = DB.init(mode="default")
(3) logger.info("DB connected with mode: %s" % db.mode)
(4) item = getItem(db)
(5) logger.info("current item: %s" % item)
(6) if check(item) is "error":
(7)     logger.error("error in item: %s" % item)
```

Fig. 1. An example program P_{ex}

```
(1) check memory status: okay
(2) DB connected with mode: default
(3) current item: pencil
(4) error in item: pencil
```

Fig. 2. An example execution log L_{ex} of P_{ex}

of source code, we highlight that our approach is likely to be restricted to identifying problems that can be localised at the source code level.

The rest of the paper is structured as follows: Section 2 illustrates a motivating example. Section 3 sketches an initial approach for *log slicing*, while Section 4 shows its application to the example, and discusses limitations and open issues. Section 5 discusses related work. Section 6 concludes the paper.

2 Motivating Example

Let us consider a simplified example program P_{ex} (Figure 1) connecting to a database and getting an item from it. For simplicity, we denote P_{ex} as a sequence of program statements $\langle s_1, s_2, \ldots, s_7 \rangle$ where s_k is the k-th statement. We can see that P_{ex} contains logging statements (i.e., s_1, s_3, s_5, and s_7) that will generate log messages when executed[3]. Figure 2 shows a simplified execution log L_{ex} of P_{ex}. Similar to P_{ex}, we denote L_{ex} as a sequence of log messages $\langle m_1, m_2, m_3, m_4 \rangle$ where m_k is the k-th log message. Note that we do not consider additional information that is often found in logs, such as timestamp and log level (e.g., *info* and *debug*)[4], so these are omitted.

The last log message "`error in item: pencil`" in L_{ex} indicates an error. Calling this log message m_{err}, let us suppose that a developer is tasked with addressing the error by reviewing the log messages leading up to m_{err}. Though we have only four messages in L_{ex}, it is infeasible in practice to review a huge amount of log messages generated by complex software systems. Furthermore, it is not necessary to review all log messages generated before m_{err} since only a subset of them is related to m_{err}; for example, if

[3] If a program statement generates a log message when executed, it is considered a logging statement; otherwise, it is a non-logging statement.

[4] We ignore log levels since the user may choose a log message of any level to start log slicing.

we look at L_{ex} and P_{ex} together, we can see that the first log message "`check memory status: okay`" does not contain information that is relevant to the error message, m_{err}. In particular, we can see this by realising that the variable `mem` logged in the first log message does not affect the computation of the variable `item` logged in the error message.

Ultimately, if we can automatically filter out such unrelated messages, with the goal of providing a log to the developer that only contains useful log messages, then the developer will better investigate and address issues in less time. We thus arrive at the central problem of this short paper: *How does one determine which log messages are related to a certain message of interest?*

An initial, naive solution would be to use keywords to identify related messages. In our example log L_{ex}, one could use the keyword "`pencil`" appearing in the error message to identify the messages related to the error, resulting in only the third log message. However, if we look at the source code in P_{ex}, we can notice that the second log message "`DB connected with mode: default`" could be relevant to the error because this message was constructed using the `db` variable, which is used to compute the value of variable `item`. This example highlights that keyword-based search cannot identify all relevant log messages, meaning that a more sophisticated approach to identifying relevant log messages is needed.

3 Log Slicing

A key assumption in this work is that it is possible to associate each log message with a unique logging statement in source code. We highlight that, while we do not describe a solution here, this is a reasonable assumption because there is already work on identifying the mapping between logging statements and log messages [4, 11]. Therefore, we simply assume that the mapping is known.

Under this assumption, we observe that the relationship among *messages in the log* can be identified based on the relationship among their corresponding *logging statements in the source code*. Hence, we consider two distinct layers: the *program layer*, where program statements and variables exist, and the *log layer*, where log messages generated by the logging statements of the program exist.

To present our log slicing approach, as done in Section 2, let us denote a program P as a sequence of program statements and a log L as a sequence of log messages. Also, we say a program (slice) P' is a subsequence of P, denoted by $P' \sqsubset P$, if all statements of P' are in P in the same order. Further, we extend containment to sequences and write $s \in P$ when, with $P = \langle s_1, \ldots, s_u \rangle$, there is some k such that $s_k = s$. The situation is similar for a log message m contained in a log L, where we write $m \in L$. Now, for a program $P = \langle s_1, \ldots, s_u \rangle$ and its execution log $L = \langle m_1, \ldots, m_v \rangle$, let us consider a log message of interest $m_j \in L$ that indicates a problem. An example could be the log message "`error in item: pencil`" from the example log L_{ex} in Figure 2. Based on the assumption made at the beginning of this section, that we can identify the logging statement $s_i \in P$ (in the program layer) that generated $m_j \in L$ (in the log layer), our log slicing approach is composed of three abstract steps as follows:

Step 1: Compute a program slice $S_r \sqsubset P$ using the combination of the statement s_i and the program variables in s_i as a slicing criterion. Notice that, apart from the logging statement s_i that is a part of the slicing criterion, S_r is composed solely of non-logging statements because logging statements do not affect the computation of any program variable[5].

Step 2: Identify another program slice $S_l \sqsubset P$ composed of logging statements that are "*relevant*" to S_r. Here, a logging statement $s_l \in S_l$ is *relevant* to a non-logging statement $s_r \in S_r$ if the message that s_l writes to the log contains information that is relevant to the computation being performed by s_r. Formally, we write $\langle s_l, s_r \rangle \in \mathsf{relevance}_P$, that is, $\mathsf{relevance}_P$ is a binary relation over statements in the program P.

Step 3: Remove any log message $m \in L$ that was not generated by some $s_l \in S_l$.

The result of this procedure would be a *log slice* that contains log messages that are *relevant* to m_j.

We highlight that defining the relation $\mathsf{relevance}_P$ for a program P (intuitively, deciding whether the information written to a log by a logging statement is *relevant* to the computation being performed by some non-logging statement) is a central problem in this work, and will be discussed in more depth in the next section.

4 An Illustration of Log Slicing

We now illustrate the application of our log slicing procedure to our example program and log (Figures 1 and 2). Since, as we highlighted in Section 3, the definition of the $\mathsf{relevance}_P$ relation is a central problem of this work, we will begin by fixing a provisional definition. A demonstration of our log slicing approach being applied using this definition of $\mathsf{relevance}_P$ will then show why this definition is only provisional.

4.1 A Provisional Definition of Relevance

Our provisional definition makes use of some attributes of statements that can be computed via simple static analyses. In particular, for a statement s, we denote by $\mathsf{vars}(s)$ the set of variables that appear in s (where a variable `x` *appears in* a statement s if it is found in the abstract syntax tree of s). If s is a logging statement that writes a message m to the log, then, assuming that the only way in which a logging statement can use a variable is to add information to the message that it writes to the log, the set $\mathsf{vars}(s)$ corresponds to the set of variables used to construct the message m. If s is a non-logging statement, then $\mathsf{vars}(s)$ represents the set of variables used by s.

Now, let us consider a logging statement s_l, that writes a message m_l to the log, and a non-logging statement s_r. We define $\mathsf{relevance}_P$[6] over the statements in a program P by $\langle s_l, s_r \rangle \in \mathsf{relevance}_P$ if and only if $\mathsf{vars}(s_l) \cap \mathsf{vars}(s_r) \neq \emptyset$. In other words, a logging statement is *relevant* to a non-logging statement whenever the two statements share at least one variable.

[5] Assuming a logging statement does not call an impure function.

[6] We remark that this simple provisional definition of relevance misses relating statements that only share syntactically different aliased variables

```
(2) db = DB.init(mode="default")
(4) item = getItem(db)
(6) if check(item) is "error":
(7)     logger.error("error in item: %s" % item)
```

Fig. 3. Program slice S_r of the program P_{ex} when s_7 and its variable `item` are used as the slicing criterion

```
(3) logger.info("DB connected with mode: %s" % db.mode)
(5) logger.info("current item: %s" % item)
(7)     logger.error("error in item: %s" % item)
```

Fig. 4. Logging statements S_l relevant to S_r

4.2 Applying Log Slicing

Taking the program P_{ex} from Figure 1 and the log L_{ex} from Figure 2, we now apply the steps described in Section 3, while considering the log message $m_4 \in L_{ex}$ (i.e., "`error in item: pencil`") to be the message of interest m_i.

Step 1. Under our assumption that log messages can be mapped to their generating logging statements, we can immediately map m_4 to $s_7 \in P_{ex}$. Once we have identified the logging statement s_7 that generated m_4, we slice P_{ex} backwards, using s_7 and its variable `item` as the slicing criterion. This would yield the program slice $S_r = \langle s_2, s_4, s_6, s_7 \rangle$ as shown in Figure 3.

Step 2. The program slice $S_r = \langle s_2, s_4, s_6, s_7 \rangle$ yielded by Step 1 contains only non-logging statements (apart from the logging statement s_7 used as the slicing criterion). Hence, we must now determine which logging statements (found in P_{ex}) write messages that are *relevant* to the statements in S_r. More formally, we must find a sequence of logging statements $S_l \sqsubset P_{ex}$ such that $\langle s_l, s_r \rangle \in \mathsf{relevance}_P$ for any logging statement $s_l \in S_l$ and a non-logging statement $s_r \in S_r \setminus \{s_7\}$. For this, we use the provisional definition of relevance that we introduced in Section 4.1, that is, we identify the logging statements that share variables with the statements in our program slice S_r. For example, let us consider the non-logging statement $s_r = s_2 \in S_r$ (i.e., "`db = DB.init(mode="default")`"). Our definition tells us that the logging statement $s_l = s_3$ (i.e., "`logger.info("DB connected with mode: %s" % db.mode)`") should be included in S_l, since $\mathsf{vars}(s_3) \cap \mathsf{vars}(s_2) = \{\texttt{db}\}$. Similarly, the logging statement s_5 should be included in S_l since $\mathsf{vars}(s_3) \cap \mathsf{vars}(s_2) = \{\texttt{item}\}$, and the logging statement s_7 should be included in S_l since $\mathsf{vars}(s_7) \cap \mathsf{vars}(s_6) = \{\texttt{item}\}$. Note that the logging statement s_2 (i.e., "`logger.info("check memory status: %s" % mem.status)`") would be omitted by our definition because no statements in S_r use the variable `mem`. As a result, with respect to our definition of relevance, $S_l = \langle s_3, s_5, s_7 \rangle$ as shown in Figure 4.

```
(2) DB connected with mode: default
(3) current item: pencil
(4) error in item: pencil
```

Fig. 5. Log slicing result from $L_{\texttt{ex}}$ when m_4 is the message of interest

Step 3. Using $S_l = \langle s_3, s_5, s_7 \rangle$, we now remove log messages from L_{ex} that were generated by logging statements *not included* in S_l. The result is the sliced log in Figure 5.

4.3 Limitations and Open Issues

We now discuss the limitations of the definition of relevance presented so far, along with a possible alternative approach. We also highlight a key open issue.

Limitations. Using a combination of program slicing and our provisional definition of relevance seems, at least initially, to be an improvement on the keyword-based approach described in Section 2. However, the major limitation of this definition, that looks at program variables shared by logging and non-logging statements, is that a logging statement must use variables in the first place. Hence, this definition can no longer be used if we are dealing with log messages that are statically defined (i.e., do not use variables to construct part of the message written to the log). In this case, we must look to the semantic content of the log messages.

An Alternative. Our initial suggestion in this case is to introduce a heuristic based on the intuition that particular phrases in log messages will often accompany particular computation being performed in program source code. Such a heuristic would operate as follows:

1. For each non-logging statement s, inspect each variable v appearing in s.
2. For each such variable v, further inspect the *tokens* found in the string literals of logging statements that are reachable from s. The word *tokens* here is deliberately left vague; it could mean individual words found in string literals, or vectors of words.
3. For each variable/token pair that we find, we compute a score that takes into account 1) the frequency of that pair in the program source code; and 2) how close they are (in terms of the distance between the source code lines in which the variable/token appear), on average.
4. We say that, for a logging statement s_l and a non-logging statement s_r, $\langle s_l, s_r \rangle \in \mathsf{relevance}_P$ if and only if s_l contains tokens that score highly with respect to the variables found in s_r. Hence, we use the token-based heuristic to define the relation $\mathsf{relevance}_P$ with respect to a single program P.

We highlight that this *token-based* approach is to be used in combination with the backwards program slicing described in Section 3.

Further Limitations. While this heuristic takes a step towards inspecting the semantic content of log messages, rather than relying on shared variables, initial implementation efforts have demonstrated the following limitations:

- It is difficult to choose an appropriate definition of a *token*. For example, should we use individual words found in string literals used by logging statements, or should we use sequences of words?
- Depending on the code base, there can be varying numbers of *coincidental* associations between tokens and variables. For example, a developer may always use the phrase "`end transaction`" near a use of the variable `commit`, but also near a use of the variable `query`. The developer may understand "`end transaction`" as being a phrase related to the variable `commit` and not to the variable `query`, despite the accidental co-occurrence of the two variables.
- Suppose that a phrase like "`end transaction`" appears only once, and is close to the variable `commit`. The developer may intend for the two to be related. However, if we use a heuristic that combines the frequency of a pair with the distance between the variable and token in the pair, a single occurrence will not score highly. Hence, there are some instances of relevance that this heuristic cannot identify.

More Issues. In Section 3, we assumed that the mapping between log messages and the corresponding logging statements that generated the log messages is known. However, determining the log message that a given logging statement might generate can be challenging, especially when the logging statement has a non-trivial structure. For example, while some logging statements might consist of a simple concatenation of a string and a variable value, others might involve nested calls of functions from a logging framework. This calls for more studies on finding the correspondence between logging statements and log messages.

Another key problem is the inconsistency of program slicing tools across programming languages (especially weakly-typed ones such as Python). If the underlying program slicing machinery made too many overapproximations, this would affect the applicability of our proposed approach. Furthermore, the capability of the tools for handling complex cases, such as nested function calls across different components, can hinder the success of log slicing.

5 Related Work

Log Analysis. The relationship between log messages has also been studied in various log analysis approaches (e.g., performance monitoring, anomaly detection, and failure diagnosis), especially for building a "reference model" [12] that represents the normal behavior (in terms of logged event flows) of the system under analysis. However, these approaches focus on the problem of identifying whether log messages *co-occur* (that is, one is always seen in the neighbourhood of the other) without accessing the source code [6, 10, 13, 17, 18]. On the other hand, we consider the *computational* relationship between log messages to filter out the log messages that do not affect the computation of the variable values recorded in a given log message of interest.

Log partitioning. Log partitioning, similarly to log slicing, involves separating a log into multiple parts, based on some criteria. In the context of process mining [1], log partitioning is used to allow parallelisation of model construction. In the context of checking an event log for satisfaction of formal specifications [3], *slices* of event logs are sent to separate instances of a checking procedure, allowing more efficient checking for whether some event log satisfies a formal specification written in a temporal logic. Hence, again, log partitioning, or slicing, is used to parallelise a task. Finally, we highlight that our log slicing approach could be used to generate multiple log slices to be investigated in parallel by some procedure.

Program Analysis including Logging Statements. Traditionally, program analysis [14, 2] ignores logging statements since they usually do not affect the computation of program variables. Nevertheless, program analysis including logging statements has been studied as part of *log enhancement* to measure which program variables should be added to the existing logging statements [7, 15] and where new logging statements should be added [16] to facilitate distinguishing program execution paths. Log slicing differs in that it actively tries to reduce the contents of a log. Finally, Messaoudi et al. [8] have proposed a log-based test case slicing technique, which aims to decompose complex test cases into simpler ones using, in addition to program analysis, data available in logs.

6 Conclusion

In this short paper, we have taken the first steps in developing *log slicing*, an approach to helping software engineers in their log-based debugging activities. Log slicing starts from a log message that has been selected as indicative of a failure, and uses static analysis of source code (whose execution generated the log in question) to throw away log entries that are not relevant to the failure.

In giving an initial definition of the log slicing problem, we highlighted the central problem of this work: defining a good relevance relation. The provisional definition of relevance that we gave in Section 4.1 proved to be limited in that it required logging statements to use variables when constructing their log message. To remedy the situation, we introduced a frequency and proximity-based heuristic in Section 4.3. While this approach could improve on the initial definition of relevance, it possessed various limitations that we summarised.

Ultimately, as part of future work, we intend to investigate better definitions of relevance between logging statements and non-logging statements. If we were to carry on with the same idea for the heuristic (using frequency and proximity), future work would involve 1) finding a suitable way to define *tokens*; 2) reducing identification of coincidental associations between tokens and variables (i.e., reducing false positives); and 3) attempting to identify associations between tokens and variables with a lower frequency.

Acknowledgments. The research described has been carried out as part of the COSMOS Project, which has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 957254.

References

1. van der Aalst, W.M.P.: Distributed process discovery and conformance checking. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering. pp. 1–25. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
2. Agrawal, H., Horgan, J.R.: Dynamic program slicing. SIGPLAN Not. **25**(6), 246–256 (jun 1990). https://doi.org/10.1145/93548.93576, https://doi.org/10.1145/93548.93576
3. Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification. pp. 31–47. Springer International Publishing, Cham (2014)
4. Bushong, V., Sanders, R., Curtis, J., Du, M., Cerny, T., Frajtak, K., Bures, M., Tisnovsky, P., Shin, D.: On matching log analysis to source code: A systematic mapping study. In: Proceedings of the International Conference on Research in Adaptive and Convergent Systems. p. 181–187. RACS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3400286.3418262, https://doi.org/10.1145/3400286.3418262
5. He, S., He, P., Chen, Z., Yang, T., Su, Y., Lyu, M.R.: A survey on automated log analysis for reliability engineering. ACM Comput. Surv. **54**(6) (Jul 2021). https://doi.org/10.1145/3460345
6. Jia, T., Yang, L., Chen, P., Li, Y., Meng, F., Xu, J.: Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD). pp. 447–455. IEEE, IEEE, Honolulu, CA, USA (2017). https://doi.org/10.1109/CLOUD.2017.64
7. Liu, Z., Xia, X., Lo, D., Xing, Z., Hassan, A.E., Li, S.: Which variables should i log? IEEE Transactions on Software Engineering **47**(9), 2012–2031 (2021). https://doi.org/10.1109/TSE.2019.2941943
8. Messaoudi, S., Shin, D., Panichella, A., Bianculli, D., Briand, L.C.: Log-based slicing for system-level test cases. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 517–528. ISSTA 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460319.3464824, https://doi.org/10.1145/3460319.3464824
9. Mi, H., Wang, H., Zhou, Y., Lyu, M.R.T., Cai, H.: Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. IEEE Transactions on Parallel and Distributed Systems **24**(6), 1245–1255 (2013). https://doi.org/10.1109/TPDS.2013.21
10. Nandi, A., Mandal, A., Atreja, S., Dasgupta, G.B., Bhattacharya, S.: Anomaly detection using program control flow graph mining from execution logs. In: 2016 26nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD). pp. 215–224. KDD '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2939672.2939712
11. Schipper, D., Aniche, M., van Deursen, A.: Tracing back log data to its log statement: From research to practice. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). pp. 545–549 (2019). https://doi.org/10.1109/MSR.2019.00081
12. Shin, D., Bianculli, D., Briand, L.: PRINS: scalable model inference for component-based system logs. Empirical Software Engineering **27**(4), 87 (2022). https://doi.org/10.1007/s10664-021-10111-4, https://doi.org/10.1007/s10664-021-10111-4
13. Tak, B.C., Tao, S., Yang, L., Zhu, C., Ruan, Y.: Logan: Problem diagnosis in the cloud using log-based reference models. In: 2016 IEEE International Conference on Cloud Engineering (IC2E). pp. 62–67 (2016). https://doi.org/10.1109/IC2E.2016.12
14. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (Jul 1984). https://doi.org/10.1109/TSE.1984.5010248, https://doi.org/10.1109/TSE.1984.5010248
15. Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S.: Improving software diagnosability via log enhancement. ACM Trans. Comput. Syst. **30**(1) (Feb 2012). https://doi.org/10.1145/2110356.2110360

16. Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., Zhou, Y.: Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In: 2017 26th Symposium on Operating Systems Principles (SOSP). p. 565–581. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3132747.3132778
17. Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., Stumm, M.: Non-Intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 603–618. USENIX Association, Savannah, GA (Nov 2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhao
18. Zhou, P., Wang, Y., Li, Z., Tyson, G., Guan, H., Xie, G.: Logchain: Cloud workflow reconstruction & troubleshooting with unstructured logs. Computer Networks **175**, 107279 (2020). https://doi.org/https://doi.org/10.1016/j.comnet.2020.107279, https://www.sciencedirect.com/science/article/pii/S1389128619316731