The Game of Twenty Questions: Do You Know Where to Log?

Xu Zhao
University of Toronto

Kirk Rodrigues
University of Toronto

Yu Luo
University of Toronto

Michael Stumm
University of Toronto

Ding Yuan
University of Toronto

Yuanyuan Zhou
University of California
San Diego

ABSTRACT

A production system's printed logs are often the only source of runtime information available for postmortem debugging, performance analysis and profiling, security auditing, and user behavior analytics. Therefore, the quality of this data is critically important. Recent work has attempted to enhance log quality by recording additional variable values, but logging statement placement, i.e., where to place a logging statement, which is the most challenging and fundamental problem for improving log quality, has not been adequately addressed so far. This position paper proposes we automate the placement of logging statements by measuring how much uncertainty, i.e., the expected number of possible execution code paths taken by the software, can be removed by adding a logging statement to a basic block. Guided by ideas from information theory, we describe a simple approach that automates logging statement placement. Preliminary results suggest that our algorithm can effectively cover, and further improve, the existing logging statement placements selected by developers. It can compute an optimal logging statement placement that disambiguates the entire function call path with only 0.218% of slowdown.

ACM Reference format:
Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. The Game of Twenty Questions: Do You Know Where to Log?. In *Proceedings of HotOS '17, Whistler, BC, Canada, May 08-10, 2017,* 7 pages.
https://doi.org/10.1145/3102980.3103001

HotOS '17, May 08-10, 2017, Whistler, BC, Canada

ACM ISBN 978-1-4503-5068-6/17/05.
https://doi.org/10.1145/3102980.3103001

1 INTRODUCTION

As today's systems grow in scale and complexity, information collected during their execution has become invaluable for postmortem debugging, performance analysis and profiling, security auditing, and user behavior analytics. The problem of how to collect this information, however, is challenging. There is a reluctance to use intrusive techniques, such as deterministic replay [1–4], given their overhead and the required non-trivial modifications to the software stack. Instead, developers rely on manually inserting logging statements.

The quality and informativeness of output log data is critically important. Most efforts to improve the quality of existing log messages are based on automatically collecting additional diagnostic information in each logging statement [7, 11], or on adjusting the verbosity of these statements [10]. They do not address the issue of *where* to place logging statements – a more fundamental and yet much more challenging problem. In earlier work [9], we proposed automatically placing *error* logging statements at locations where software error conditions (e.g., non-zero system call return values) may occur. While this is a crucial first step towards the placement of logging statements for postmortem debugging, many failures, especially complicated ones with long fault-propagation paths, require additional logs that capture non-error but still important execution states. In this paper, we refer to such logs as INFO-logs, since they typically have INFO verbosity and are enabled by default in production environments.[1] In mature software systems, such as HDFS, these INFO-logging statements are almost as prevalent as all the WARN/ERROR/FATAL logging statements combined (816 versus 912 logging statements, respectively), demonstrating that they are equally critical for effective postmortem diagnosis.

[1] More verbose levels, e.g., DEBUG, usually output events that are less important, and are typically disabled by default in production environments. However, we are aware of some commercial systems that enable DEBUG-level logs by default.

However, automatic placement of INFO-logging statements is much more challenging than automatic placement of error logging statements or enhancing existing logging statements. First, without domain expertise, it is hard to know which program locations are more "log-worthy" than others. Even for programmers with domain expertise, deciding whether or not to insert an INFO-logging statement at a particular location is like "shooting blind", since it is often hard to predict the usefulness of the statement prior to the software release and the occurrence of unexpected failures. In contrast, error logging statements are often much easier to place (e.g., wherever a generic error condition is encountered), and languages with built-in exception support, such as Java or Scala, make it even easier as one can simply place error logging statements in the exception-catch blocks.

Second, unlike inserting error logging statements, where overhead is often not a concern because the system is already in an erroneous state, each additional INFO-logging statement introduces performance overheads and uses up storage resources. This is perhaps why, despite the importance and prevalence of INFO-logs, we are not aware of any guidelines in any systems that specify where one should place INFO-logging statements.

In this position paper, we propose to completely automate the placement of logging statements for a variety of purposes, including:

- *Postmortem debugging*, so that developers or administrators can understand a failure by reconstructing the failed execution path (from automatically placed logging statements).
- *Performance analysis and profiling*, so that developers can analyze the performance of a system, e.g., the latency of each request (from automatically placed logging statements).
- *Security auditing*, so that vendors can analyze exactly what information each user was able to get access to and what information each user was able to change. In case of a security breach, vendors can identify what information leaked out and what may have been compromised.
- *User behavior analytics*, so that vendors can analyze users' behavioral patterns to optimize a product or service accordingly.

Up to now, our focus has been on achieving the first objective, which is also the focus of this paper.

Our key observation is that log data is used to determine which requests, functions, and execution paths led to a failure. Information theory can help us place logging statements so that (1) they provide the information needed, and (2) do so in an efficient way. To motivate the approach, consider how the problem of logging is similar to the game of twenty questions, where a player's goal is to identify an object in twenty yes/no questions or less. Information theory suggests the best strategy is to ensure each question eliminates at least half of the remaining uncertainty (i.e., possibilities). Similar to the game, placing a logging statement is like asking a yes/no question whose answer depends on whether the statement gets executed or not.

However, there are two differences between the twenty questions game and placing a logging statement. First, the cost is not measured by the number of logging statements being inserted, but by the number of log messages being *printed*.[2] Secondly, the placement of logging statements is not targeted to a specific failure, but *any* failure that could occur.

We propose an algorithm design that realizes the following: given a performance overhead threshold (e.g., less than 2% slowdown), it computes the "best" placement of INFO-logging statements such that uncertainty is minimized. In information theory, uncertainty is a measure of how sure you are about the next bit of information to be received. Similarly, we model the "uncertainty" of software by considering the total number of distinct code paths a program execution may have taken. Intuitively, a program with a small number of possible paths has less uncertainty than one with more paths. Note that execution paths can be measured at different granularities. Function call path and branch-level profile are examples of two granularities.

We have implemented a simple prototype that can automatically place INFO-logging statements. Furthermore, we show that the algorithm automatically places such statements in the same program locations where programmers manually inserted logging statements, demonstrating that our idea indeed matches developers' intuition in logging. Since the placement is automatic, the algorithm does not "forget" to place a logging statement, which happens all too often in the real-world where many logging statements are only added as after-thoughts, i.e., they are added after a failure has occurred [10]. We further show that our algorithm can improve existing log-placements even in mature systems like HDFS. Finally, we show that our algorithm can compute a log-placement strategy that completely disambiguates the entire call path of HDFS requests with only 0.218% of performance slowdown.

2 SOFTWARE UNCERTAINTY

The central task of postmortem debugging is to determine which path was taken by the failed execution. We refer to the space of all possible execution paths as the uncertainty space, X, or simply uncertainty, and x represents one specific execution path. Shannon's entropy [8] is a useful measure

[2]We assume that the size of the binary executable is not of primary concern since the number of static logging statements will primarily affect binary size.

```
1 boolean invalidateBlock(Block b,
2  DatanodeInfo dn) throws IOException {
3  DatanodeDescriptor node = getDatanode(dn);
4  if (node == null)
5   throw new IOException("Can't invalidate"+b
6   +"because datanode "+dn+" doesn't exist");
7
8  //Check how many copies we have of b
9  NumberReplicas nr = countNodes(b.stored);
10 if (nr.replicasOnStaleNodes() > 0) {
11    postponeBlock(b.corrupted);
12    return false;
13 } else if (nr.liveReplicas() >= 1) {
14    // We have at least one copy on live node,
          can delete it.
15    addToInvalidates(b.corrupted, dn);
16    removeStoredBlock(b.stored, node);
17    return true;
18 } else {
19    return false;
20 }
21 }
```

Figure 1: Code snippet from HDFS.

of uncertainty that relies solely on the probability of each possible outcome. It is defined as,

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x) \tag{1}$$

where $p(x)$ is the probability of observing the xth execution path. For now we assume that we are able to observe all of X, i.e., all possible paths, along with the probability of each path being taken in a production environment. Obviously this is impossible in practice, but it could be approximated by obtaining traces from running realistic workload generators, during Alpha and Beta testing, or even by continuously sampling the paths in a production environment. This entropy, $H(X)$, measures uncertainty. Intuitively, it means that in order to represent all possible outcomes in X, we will need at least $H(X)$ bits of information. Note that the granularity at which the path is being recorded (i.e., at the branch, function, or component level) is inconsequential to the algorithm.

Consider the HDFS method shown in Figure 1. Figure 2 shows the control flow graph (CFG) for this method. There are four possible execution paths. If we assume each path is equiprobable, the entropy of this method is 2.0, indicating that on average we need two bits to represent each path. In reality, the probability of these four paths is different. Path 3 occurs most frequently as it represents the most common case. The other three paths handle rare conditions: the datanode

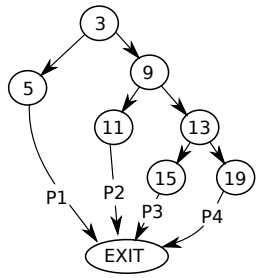

Figure 2: The control flow graph of `invalidateBlock()`. Each node represents a basic block. The number in the node is the line number of the first code line of the respective basic block. P1, P2, P3, P4 number the four possible execution paths.

cannot be located (path 1), the block is corrupted (path 2), or there does not exist any live replica of the block (path 4).[3] Assume we observe that the probabilities of the four paths are 1%, 4%, 90%, and 5%. Then Equation 1 gives us entropy $H = 0.605$. This indicates there is less "uncertainty" when executing this method in production than if the four paths were equiprobable.

3 A LOG-PLACEMENT ALGORITHM

We now describe our log-placement algorithm. A log-placement is a set of locations in the program where logging statements are placed. Given a program and a performance degradation threshold, the core idea of the algorithm is to decide for each subsequent logging statement where in the program to place the statement such that (1) the threshold is respected, and (2) the log-placement minimizes uncertainty relative to all other placements that respect the threshold. This requires us to be able to measure the overhead and uncertainty of each log-placement so that we can identify the best placement.

For each basic block, we consider two possibilities: to place a logging statement there or not. Thus, we can enumerate all possible overall log-placements. Figure 3 shows four of these for `invalidateBlock()`.

Each log-placement clearly reduces the entropy, i.e., uncertainty. Our algorithm allows us to measure the precise uncertainty that remains in the program after placing each logging statement. To do this, we first need to consider every sequence of logging statements the program could output, then measure the entropy given each log sequence, and finally aggregate the entropies by considering the probability of the corresponding log sequence. More formally, the entropy of a program under a log-placement strategy, L, can be defined

[3]Note that only path 1 is an actual error condition; path 2 and 4 are non-error, but rare, conditions.

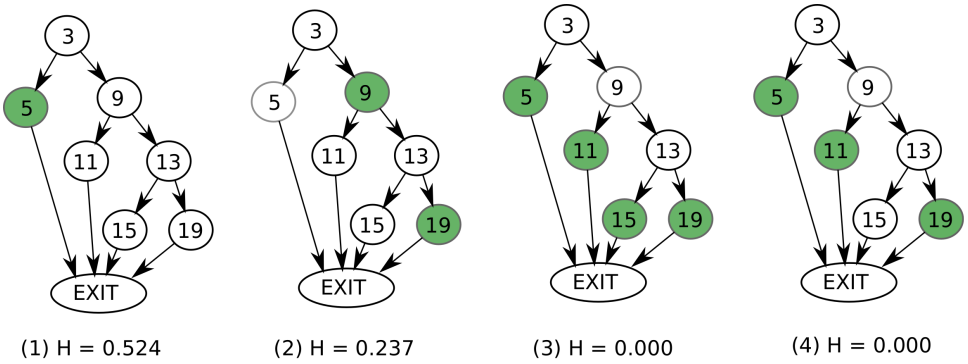

Figure 3: Four different log-placements for `invalidateBlock()`. A colored node indicates a logging statement has been placed in the basic block. We also show the remaining uncertainty under each log-placement strategy.

by Equation 2:

$$H_L(X) = \sum_{l \in L\text{'s log sequences}} p(l)H(X_l) \quad (2)$$

where l is a possible log sequence output by the program's execution, $p(l)$ is the probability of the program outputting log sequence l, and $H(X_l)$ is the entropy of the program when the execution outputs sequence l.

For example, consider placement 2 in Figure 3. With this log-placement, l could be one of the following: (1) empty, i.e., no log is output after the method is executed, which is the output of path 1; (2) <9>, which is the output of either path 2 or 3; or (3) <9,19>, which is the output of path 4. Recall that the four execution paths have probabilities 1%, 4%, 90%, and 5%, so the probabilities of an execution of this method outputting the three log sequences are 1%, 94%, and 5%, respectively. If the log output is either empty or <9,19>, the entropy $H(X_l)$ is 0, because one can unambiguously know that the path is either path 1 or path 4. However, if the log output is <9>, then one still cannot determine whether the execution took path 2 or 3; more precisely, the remaining uncertainty $H(X_l)$ when l = <9> is 0.253, as calculated by Equation 1. We can now calculate the entropy of this log-placement: $H_L(X) = p(\text{empty}) \times H(X_{\text{empty}}) + p(\text{<9>}) \times H(X_{\text{<9>}}) + p(\text{<9,19>}) \times H(X_{\text{<9,19>}}) = 0.237$.

Both placement 3 and 4 in Figure 3 eliminate *all* uncertainty since different execution paths will output different log sequences.

One also has to consider the performance overhead of each placement. We can simply run the program with the inserted logging statements using workload generators, and measure the average overhead. However, because we consider a large number of placement strategies, running each against real workload may be time consuming. We can first estimate the overhead of each path as,

$$\text{overhead}(x) = \frac{|L(x)| \times \text{LOG\_LAT}}{\text{runtime}(x)} \quad (3)$$

where $|L(x)|$ is the number of log messages in path x for placement strategy L, LOG_LAT is the approximate latency of executing each logging statement, and runtime(x) is the runtime of path x, measured when we collect its trace. For example, in `invalidateBlock()`, the runtime of path 3 is 8.5 ms on our server cluster, and the execution of a logging statement is at least 20 µs without printing any parameter values. With these measurements, we can estimate that under placement 2 and 3 in Figure 3, the overhead introduced to this execution path is 0.25%. We can further estimate the average overhead considering all possible execution paths to be

$$\sum_{x \in X} p(x) \times \text{overhead}(x) \quad (4)$$

This allows us to evaluate those placements with estimations under the performance threshold.

Putting these steps together, our log-placement algorithm consists of the following steps: (1) run the program under representative workloads to collect the paths executed and their individual latencies; (2) enumerate all log-placement strategies, whether function level or basic block level, and calculate the uncertainty and overhead of each strategy; (3) select the placement strategy with the least amount of uncertainty among the ones that respect the overhead threshold; and (4) further run the program with this placement under a realistic workload to validate that it respects the threshold.

Consider `invalidateBlock()`. After evaluating all the basic block level placement strategies, we found that placement 4 in Figure 3 is the optimal strategy – it eliminates all uncertainty, while only introducing 0.02% overhead. The low overhead is because on the most frequent path, path 3, it does not output any log info at all. In fact, placement 4 is exactly how HDFS developers placed the logging statements. In the basic block for line 11, developers placed the following logging statement:

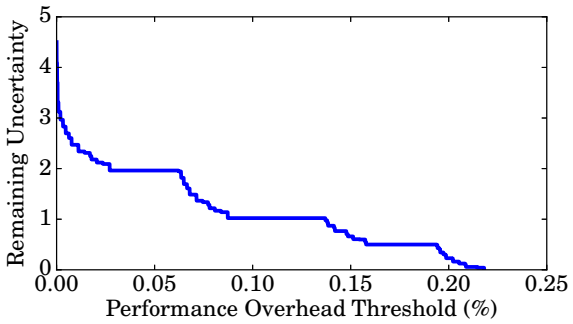

Figure 4: The relationship between uncertainty and the overhead threshold for function-level log-placements.

```
Log.info("postponing invalidation of "+b+" on "
 + dn + " because" + nr.replicasOnStaleNodes()
 + " replica(s) are located on nodes with"
 + " out-of-date block reports");
```

Similarly, before line 19, developers placed the following logging statement:

```
Log.info("BLOCK* invalidateBlocks: "+b+" on "
 +dn+" is the only copy and was not deleted");
```

This indicates that our algorithm matches developers' manual logging practice, showing the promise of automating log-placement.

4 PRELIMINARY RESULTS

We have implemented our algorithm in Python. It takes traces of execution paths, either at function or basic block granularity, along with their execution latencies. The algorithm calculates the probability of each execution path, which is simply the number of appearances of this path divided by the total number of paths. It then enumerates every possible log-placement and calculates its total uncertainty and average performance overhead as described in Section 3. The algorithm terminates when all placements are enumerated, then reports the log-placement that has the least amount of uncertainty and respects the performance overhead threshold.

In order to understand the trade-off between uncertainty reduction and the overhead of logging, we need to analyze the log-placements generated by our algorithm in practice. To do this, we ran SQL aggregate and join workloads from the HiBench [6] benchmark suite on a three-node Hadoop cluster, and traced the execution paths in HDFS. Every trace was generated by logging the beginning and end of each method from which we determined their sequence of execution in a path as well as their overall latencies (after subtracting the instrumentation overhead). Since over 2,000 unique methods were executed across all traces, it is too expensive, computationally, to evaluate every possible log-placement strategy. Instead, we applied a few simple optimizations to reduce the number of placements considered: first, we combined sequences of methods that always appear together. Second, we restricted the search to the set of methods guaranteed to affect the entropy of each considered placement. We calculated this set by removing one method at a time from the placement containing all methods, so long as the removal did not increase the entropy. Unfortunately, this no longer ensures the final placements are optimal, but nonetheless demonstrates the trade-off. Under the optimizations, our tool enumerated a total of 65,536 different log-placements.

Figure 4 serves to illustrate the relationship between uncertainty and the overhead threshold. Intuitively, the program contains the largest amount of uncertainty (4.500) when we cannot tolerate any overhead for logging. At the other end of the spectrum, we see that with only 0.218% of overhead, we can disambiguate the program's paths entirely with no uncertainty. The small overhead is because our algorithm carefully avoids placing logging statements on the hot path, and because our workload is I/O bound, so the computational cost of logging does not affect the critical path. The shape of the graph in the middle indicates the trade-off between uncertainty and overhead.

We also compared our basic block level log-placement with developers' manual log-placement on HDFS. Currently we do not have an automated way to collect basic block level traces; therefore for each function, we simply checked to see whether our algorithm could improve, without increasing overhead, the existing log-placement under any path probabilities, i.e., resulting in less uncertainty. we found 13 functions where our algorithm reported that it can improve log-placement. Here we show two of them. The first function is shown below:

```
1 LocatedBlock locateFollowingBlock(..) {
2  .. catch (IOException e) {
3   LOG.info("Exception when adding block",e);
4   if (Time.now() - localstart > 5000) {
5    LOG.info("Waiting for replication for"
6    + (Time.now()-localstart)/1000 + "s");
7   }
8   try {
9    LOG.warn("Sleeping " + src);
10   Thread.sleep(sleeptime);
11   sleeptime *= 2;
12  } catch (InterruptedException ie) {
13   LOG.warn("Caught exception ", ie);
14  }
15 }
16 }
```

There are four logging statements and our algorithm suggested that removing the one on line 3 would result in the same amount of uncertainty, but less overhead. The second example is shown below:

```
1 void shutdown() {
2  LOG.info("Waiting for threadgroup to exit,"
3  +" threads is "+threadGroup.activeCount());
4   if (threadGroup.activeCount() == 0)
5     break;
6   try {
7     Thread.sleep(sleepMs);
8   } ...
9 }
```

Our algorithm reported that if we remove the logging statement on line 2, and add logging statements at line 5 and line 7 respectively, we can remove uncertainty, i.e., differentiate the paths at line 5 and line 7 without introducing additional overhead. But a careful reader will notice that the existing logging statement at line 3 outputs `threadGroup.activeCount()`, whose value can be used to determine the branch direction at line 4. This shows a limitation of our current implementation; i.e., we do not consider which variable values should be included in logging statements. The problem of including causally-related variable values in a logging statement has been solved by LogEnhancer [11], and we are currently implementing it on top of our algorithm.

5 FUTURE WORK

Our current prototype uses a brute force search algorithm. Given N log points (whether functions, basic blocks, etc.), the algorithm must evaluate 2^N unique placements. Each placement must be evaluated against every unique path in order to determine the possible log outputs. This makes the final complexity $O(2^N \times |X|)$, where $|X|$ is the number of unique paths. As we saw in Section 4, this does not scale for the benchmark experiment, necessitating a better algorithm or a sufficient approximation.

In addition, we currently instrument the code manually to collect path profiles. In the future, we plan to design and implement a continuous tracing system that can be used to sample execution paths in a production environment.

The coverage of these traces may not be exhaustive, as some functions or basic blocks will not be exercised by the workload we use. We can add a logging statement for every function that is not in the trace, together with branch variables and parameter values to disambiguate multiple paths.

Our algorithm does not currently consider which variable values should be included in each logging statement, nor does it consider the effect of logging such branch variables in disambiguating paths. Furthermore, in contrast to manually placed logging statements, the automatically placed statements are not human-readable, as we do not currently automate the construction of meaningful static text. We plan to address both issues in the future.

6 RELATED WORK

Some prior works studied the problem of log-placement. Yuan *et al.* [9] analyzed existing logging practice and recognized that a majority of failures manifest within a small subset of generic error patterns. They further developed Errlog, a tool that places error logging statements at strategic locations based on error patterns. Fu *et al.* [5] presented an empirical study and summarized current logging practice in the industry. Li *et al.* [7] enhanced the work of Fu *et al.* and offered log suggestions based on existing log-placements using a machine learning algorithm. In contrast, our log-placement strategies are generated by analyzing program code and performance traces. It does not require any existing logging statements, and it can automate log-placement from scratch. In addition, our algorithm can help improve existing log-placements, whereas the learning based approach relies on existing logging statements to learn its model. Finally, these works do not consider performance overhead.

Others studied how to enhance existing logging statements. LogEnhancer [11] enhances existing logging statements by appending extra causally-related information which disambiguates some execution paths. Yuan *et al.* studied existing logging practices and proposed a tool to adjust the verbosity of each logging statement [10]. These works are complimentary to this paper because they address different aspects of log automation. Working together, the exercise of log printing could be fully automated.

7 CONCLUDING REMARKS

We propose to automate the placement of logging statements based on measuring the software's uncertainty that can be eliminated by different log-placement strategies which respect a certain performance slowdown threshold. Preliminary results have demonstrated that our approach can automate the placement of logging statements and even surpass the existing log-placements in mature systems. Future work should include a more scalable version of the algorithm, a continuous tracing system to sample execution paths, and automated generation of log message content including what variables should be logged in order to disambiguate more paths.

REFERENCES

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89. ACM, 2003.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI '04, pages 259–272. USENIX Association, 2004.

[3] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 217–231. USENIX Association, 2014.

[4] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI '07, pages 271–284. USENIX Association, 2007.

[5] 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*, ICSE Companion 2014, pages 24–33. ACM, June 2014.

[6] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *26th International Conference on Data Engineering Workshops*, ICDEW '10, pages 41–51. IEEE Computer Society, 2010.

[7] H. Li, W. Shang, Y. Zou, and A. E. Hassan. Towards Just-in-time Suggestions for Log Changes. *Empirical Software Engineering*, pages 1–35, October 2016.

[8] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(4):623–656, October 1948.

[9] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, Y. Zhou, and S. Savage. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation*, OSDI '12, pages 293–306. USENIX Association, 2012.

[10] D. Yuan, S. Park, and Y. Zhou. Characterizing Logging Practices in Open-Source Software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112. IEEE Press, 2012.

[11] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the 16th International Conference on Architecture Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 3–14. ACM, 2011.