Mitigating the Uncertainty and Imprecision of Log-Based Code Coverage Without Requiring Additional Logging Statements

Xiaoyan Xu, Filipe R. Cogo, *Member, IEEE,* Shane McIntosh, *Senior Member, IEEE*

Abstract—Understanding code coverage is an important precursor to software maintenance activities (e.g., better testing). Although modern code coverage tools provide key insights, they typically rely on code instrumentation, resulting in significant performance overhead. An alternative approach to code instrumentation is to process an application's source code and the associated log traces in tandem. This so-called "log-based code coverage" approach does not impose the same performance overhead as code instrumentation. Chen et al. proposed LogCoCo — a tool that implements log-based code coverage for Java. While LogCoCo breaks important new ground, it has fundamental limitations, namely: uncertainty due to the lack of logging statements in conditional branches, and imprecision caused by dependency injection. In this study, we propose Log2Cov, a tool that generates log-based code coverage for programs written in Python and addresses uncertainty and imprecision issues. We evaluate Log2Cov on three large and active open-source systems. More specifically, we compare the performance of Log2Cov to that of Coverage.py, an instrumentation-based coverage tool for Python. Our results indicate that 1) Log2Cov achieves high precision without introducing runtime overhead; and 2) uncertainty and imprecision can be reduced by up to 11% by statically analyzing the program's source code and execution logs, without requiring additional logging instrumentation from developers. While our enhancements make substantial improvements, we find that future work is needed to handle conditional statements and exception handling blocks to achieve parity with instrumentation-based approaches. We conclude the paper by drawing attention to these promising directions for future work.

Index Terms—Static Analysis, Code coverage, Software logging

1 Introduction

Developers strive to understand the behavior of large and complex software systems. To gain insights into a system's behavior, developers rely on software analysis tools [20], which may suffer from shortcomings. For example, instrumentation-based software analysis tools can generate considerable performance overhead during the execution of the analyzed program [11, 25, 33, 46]. In systems that are performance sensitive and need intensive monitoring, such as high-traffic web services and real-time applications, instrumentation-based methods may interfere with critical performance requirements that are essential for the system's functionality, and meeting implicit or explicit Quality of Service (QoS) expectations [4, 22, 41, 42]. Adapting instrumentation-based tools across systems is also challenging because such tools are often language specific [12]. Moreover, the deployment of instrumentation-based tools is non-trivial. In distributed systems, for instance, the deployment of instrumentation tools presents challenges due to the need for pervasive system modifications, which crosscut nearly every component of the system [29].

To address the aforementioned limitations in code coverage measurement, Chen et al. [10] proposed LogCoCo—a tool to measure the code coverage of Java-based systems by exploiting broadly available system execution logs. Unlike instrumentation-based coverage tools, LogCoCo does not add overhead that affects the overall system's performance because it relies on execution logs that systems are often already generating to, e.g., monitor system health, debug runtime issues, and comply with legal requirements in regulated industries. Despite code coverage being traditionally applied in the context of testing, the application context of LogCoCo is broader, as it can produce code coverage measurements based on any set of execution logs and under any execution scenario. This indicates that any software that uses logging can potentially benefit from this approach, particularly in scenarios where traditional instrumentation-based methods are intrusive or infeasible.

While LogCoCo makes an important contribution, it is not without limitations. First, LogCoCo's nature of inferring coverage by execution logs limits its performance to measuring the coverage of log-sparse areas of the source code, for which *uncertainty* can occur. Uncertainty refers to the code region that cannot be determined as covered or not covered based on the execution logs. Second, the development paradigm of dependency injection can cause *imprecision* in LogCoCo's coverage measurement, since dependency injection dynamically modifies system execution flows during runtime. For example, a common practice of dependency injection in unit testing is mocking and patching, where a function can be replaced by a mocked object and is not invoked during system execution, even though static analysis indicates that it is. Without an understanding

- *Xiaoyan Xu and Shane McIntosh are with the David R. Cheriton School of Computer Science, University of Waterloo, Canada. E-mail: {xiaoyan.xu, shane.mcintosh}@uwaterloo.ca*
- *Filipe R. Cogo is with the Huawei Centre for Software Excellence, Canada. E-mail: filipe.roseiro.cogo1@huawei.com.*

Manuscript received date; revised date.

of dependency injection, LOGCOCO can yield inaccurate coverage measurements.

In this paper, we propose LOG2COV—a log-based coverage approach that aims to address the aforementioned limitations caused by uncertainty and imprecision. In addition to these conceptual enhancements, we contribute a prototype implementation of LOG2COV for PYTHON, which expands the reach of log-based coverage concept to a new programming language. We perform an exploratory evaluation to verify the performance of log-based coverage measurement for PYTHON. We systematically select three of the largest, most active and log-dense open-source systems on GITHUB as subject systems for our evaluation. We benchmark our performance measurements in comparison with COVERAGE.PY—a popular instrumentation-based coverage tool for PYTHON. More specifically, we compare the results of COVERAGE.PY against the results of LOG2COV to verify the performance of the latter. Our results show that our direct port of LOGCOCO to PYTHON achieves high precision, recall, and F1 score. Our results also confirm the presence of the limitations of uncertainty and imprecision. Armed with an empirical understanding of these limitations, we enhance LOG2COV to mitigate uncertainty and imprecision. We strive to reduce uncertainty without asking developers to supply additional logging statements in log-sparse conditional blocks. Such conditional blocks result in uncertainty because there are no log statements to ascertain which execution flow was invoked. We propose a technique that incorporates program slicing [45] and data flow analysis [1] to trace variables that are referenced within conditional expressions, and infer their values to resolve the expression and determine the coverage status of statements inside conditional blocks. To address the imprecision problem, we identify the usage of dependency injection in unit testing and intervene in the analysis of system execution flows to correct the corresponding coverage measurement.

We evaluate our approach, which addresses uncertainty and imprecision, by answering the following RQs:

(RQ1) To what extent can we reduce the uncertainty of log-based code coverage?
Motivation: Log statements are not inserted everywhere. Log-sparse regions of a system can lead to uncertainty in log-based coverage measurement. While prior work [10] suggests inserting additional logging statements as a solution, we aim to relax this constraint on users by mitigating uncertainty without requiring changes to the system under scrutiny.
Results: Our approach resolves 1%-11% of lines whose coverage status could not be ascertained by execution logs alone, and it achieves 100% of accuracy across our subject systems.

(RQ2) To what extent can we reduce the imprecision of log-based code coverage?
Motivation: An inaccurate coverage tool is not of practical value. The practice of dependency injection can cause imprecision in log-based coverage measurements, since it dynamically modifies the system execution flow during runtime. To improve the practical value of log-based coverage estimation, we set out to mitigate the impact of dependency injection on the precision of measurements.
Results: Our approach achieves up to 4 percentage points improvement on the precision of LOG2COV when dependency injection is being used, and has no negative impact on the performance of LOG2COV when dependency injection is not being used.

2 BACKGROUND AND RELATED WORK

In this section, we situate our work with respect to the literature on software logging and code coverage.

2.1 Software Logging

Software logging is a popular approach to recording events that occur while a program executes [48]. A log trace is the text output of such a recorded event, and it is generated by executing the log statement that is inserted into the source code by developers [17]. During the system execution, the collected log traces are saved in log files. A logging statement typically contains four types of components: a logging object, a verbosity level, static texts, and dynamic contents [9]. During the system execution, the verbosity level (e.g. *INFO*, *DEBUG*, *ERROR*) determines whether a log trace should be outputted, the dynamic contents represent the state of the system while it is running, whereas static texts provide a human-readable description of the logging context (e.g., type of event).

Because of the runtime information that the log files contain, software logging is heavily and widely used in large enterprise applications for monitoring and debugging [21]. For example, anomaly identification [40], system monitoring [39], failure analysis [23, 35, 36, 49], and test analysis [43] all rely on analyzing logs emitted in the execution of large-scale software systems. Moreover, the runtime information from logs can be leveraged to complement static analysis [31], thereby providing a more comprehensive understanding of system behavior.

Previous work has studied developers' logging practices. Chen et al. [8] studied five server-side projects and five client-side projects from the APACHE Software Foundation to assess whether the logging practices of client-side projects are similar to those of the server-based projects. They analyzed log density, which is defined as the ratio between total lines of source code and total lines of logging code, across those projects. They found that the pervasiveness of logging varies from project to project. Alves and Paula [2] explored the logging practices of 1,166 open-source PYTHON projects that use containers. They found that over 99% of the studied projects use the built-in PYTHON logging library, and that the logging verbosity levels *DEBUG* and *INFO* are used almost twice as much as *WARNING* and *ERROR*.

Yet, empirical investigations demonstrate that no well-established logging standard exists for proprietary [14] and open source systems [8, 48]. Researchers have made efforts to fill this gap by suggesting *where to log* (i.e., specifying where logging statements should be placed) and *what to log* (i.e., specifying the information that log statements should record). Yuan et al. [47] studied 250 randomly sampled reported failures across five large and widely used software systems and found that missing logging statements increase the time to resolve failures up to 2.2 times compared to the average resolving time. The authors proposed ERRLOG, a static checking tool that automatically instruments log statements to record the error locations and error context while searching the codebase for these kinds of exception blocks. Zhu et. al [52] proposed `LogAdvisor`, which utilizes existing logging statements to automatically learn logging practices for *where to log* and uses that information to provide developers with recommendations. He et al. [17] carried out the first empirical study in the context of logging statements, concentrating on the natural language descriptions of those statements. They summarized three categories of logging descriptions in logging statements, including the description of program operation, the description of error conditions, and the description of high-level code semantics (i.e., variable description, function description, and branch description).

Software logging offers a wide range of benefits, covering the dimensions of diagnosing system failures, tracking execution status, understanding system behavior, and recording important transactions or operations in system executions. Yet there is no universal standard of logging. Inadequate logging can pose negative impacts on log analysis.

2.2 Code Coverage and Instrumentation-Induced Overhead

Code coverage is a technique to determine which code locations (e.g., branches or statements) are executed when a system runs under determined conditions. Code coverage is mainly used to assess and enhance the quality of tests [51], such as unit tests and integration tests [3, 5, 24, 30, 34].

While both open-source and proprietary tools that measure code coverage are popular and broadly available (e.g., JACOCO[1], Semantic Designs tools[2], COBERTURA[3]), they typically use the same instrumentation technique, which consists of inserting probes (either at the source code [6] or the binary/bytecode level [28]) to capture the runtime system behavior [46]. For example, Yang et al. [46] compared 17 coverage-based testing tools that support different languages (e.g., JAVA, C, and C++) and found that they are all based on instrumentation. However, instrumentation causes performance overhead, making the overall system execution slower [16, 44]. Instrumentation-induced overhead can be categorized as either offline, caused by the process of inserting the probes, or online, caused by the execution of the probes to record the execution traces [46].

Prior research studied the performance overhead caused by instrumentation-based code coverage tools. Such overhead is often measured by comparing the performance of executing tests with/without turning the coverage tool on. Chen et al. [10] measured the overhead of JACOCO, a commonly used JAVA coverage tool [10], and observed that negative performance impact varies between workloads. Nonetheless, they found that JACOCO brought a noticeable performance overhead (greater than 8% on average) on the System Under Test (SUT) across all benchmark tests. Holmes et al. [18] measured the performance overhead introduced by the state-of-the-art PYTHON coverage tool COVERAGE.PY.[4] The authors measured the performance overhead in terms of the number of test actions (e.g., method calls) performed in a 60-second time range. As a benchmark, the authors selected a set of PYTHON libraries as the SUT and the generated tests by TSTL [18], a domain-specific language for creating test harnesses. They observed that turning off code coverage tools can lead to executing at least 10% more test actions (on average), and up to 50 times as many test actions. The median improvement in SUTs was 2.03 times, with a mean improvement of 6.12 times as many test actions.

The state-of-the-practice code coverage tools rely on instrumentation, which results in non-negligible performance overhead for large-scale software systems.

2.3 Log-based Code Coverage

To address the performance overhead problem in code coverage measurement, Chen et al. [10] proposed LOGCOCO, a tool to measure code coverage using execution log traces of JAVA-based systems. LOGCOCO measures three kinds of code coverage: Must-Coverage, Must-Not-Coverage, and May-Coverage. Must- and Must-Not-Coverage refer to statements that are either necessarily covered or not covered by an execution flow. May-Coverage refers to the statements inside conditional branches at which there is no logging statement indicating its reachability. Unlike instrumentation-based coverage tools, LOGCOCO does not add extra overhead that affects the overall performance of program execution because it avoids the usage of probes (see Section 2.2) by relying on readily available execution log traces to determine coverage. Evaluation results indicate that LOGCOCO achieves high precision in measuring code coverage [10]. As stated by Chen et al., the coverage information computed by LOGCOCO highly depends on the number of logging statements in the source code [10]. This implies that the performance of LOGCOCO may be limited by the subject systems because the amount of logging statements varies from system to system.

Log-based code coverage measurement is currently limited to JAVA. Exception handling and conditional branches lead to different program execution flows, and missing logging statements on such critical points poses negative impacts on the identification of causally related program execution flows and failures.

3 LOG2COV: LOG-BASED CODE COVERAGE FOR PYTHON

In this section, we present our solution to relax the constraint that LOGCOCO only works for JAVA programs. First, we describe the implementation of LOG2COV, which begins as a direct implementation of LOGCOCO for PYTHON programs (Section 3.1). We then present the design (Section 3.2)

[1] http://www.eclemma.org/jacoco/
[2] http://www.semdesigns.com/Products/TestCoverage/
[3] http://cobertura.github.io/cobertura/
[4] https://coverage.readthedocs.io/en/6.4.1/

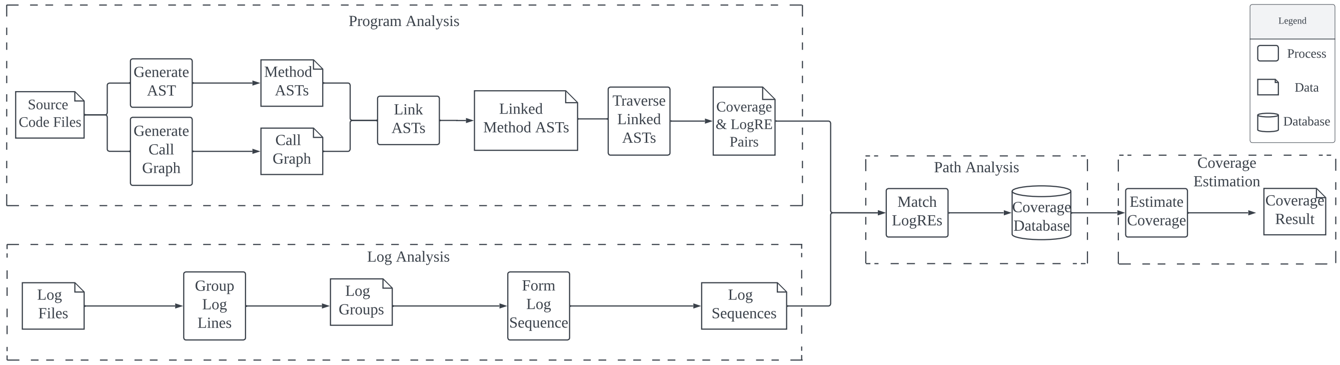

Fig. 1. Overview of the design of Log2Cov.

and results (Section 3.3) of an evaluation regarding precision, recall, F1 score and performance overhead of this initial version of Log2Cov. The limitations that we identified during this initial evaluation inspire the key improvements that we propose for Log2Cov (Section 3.4).

3.1 System Design

The design of Log2Cov is similar to that of LogCoCo [10] and contains four phases: (1) Program Analysis, (2) Log Analysis, (3) Path Analysis, and (4) Coverage Estimation. The Log2Cov design is illustrated in Figure 1.

3.1.1 Phase 1 – Program Analysis

The Program Analysis phase takes the program's source code as input to derive a mapping between *LogREs* and their corresponding coverage information. A *LogRE* is a (sequence of) regular expression(s) composed during the analysis of the program. It denotes the occurrence of execution flows of the program and is used to match against log traces to reveal the code coverage [10]. For example, the *LogRE* `(module@3module@2)+` matches the location component of the log trace `module@3module@2module@3module@2`, where `module` refers to a source file name, and the numerals refer to the lines where the log statements that generate this given log trace appear.

We follow the same procedure as proposed by Chen et al. [10] to analyze the source code of the program. We first obtain the Abstract Syntax Tree (AST) of each method of the program using Python's AST library. The obtained ASTs are stored as files on disk, strictly following the project level hierarchy. We then obtain a static call graph of the program using PyCG, which has been shown to outperform other tools for the same task [38]. The resulting call graph is represented as a map in which the key is the caller and the value is a list of callees. Both caller and callee are denoted as the relative path to the method in dot notation. The call graph shows the relative path of each method to the system root directory, which enables us to chain together ASTs associated with each method. Such a procedure involves traversing the AST body of each method, identifying function calls, and replacing the callee's name with the location of the callee's AST. Finally, we perform an AST traversal for all methods' ASTs to find all possible execution flows of the program and generate the respective *LogREs*. During the AST traversal, once we encounter a logging statement, we record its module name and line number to compose the *LogRE*. Similar to LogCoCo, we label each statement as Must-Covered, Must-Not-Covered, or May-Covered during the construction of *LogREs*.

3.1.2 Phase 2 – Log Analysis

In this phase, we analyze the log files and form a string of concatenated log sequences. A log sequence is composed of multiple patterns of `module@lineno`, where each pattern is extracted from a log trace.

We analyze log files by leveraging the log format, which includes the timestamp, thread id, and location of each log trace. We describe how we use each item below.

Timestamp is used to sort log traces. Since multiple log files can be generated during testing (e.g., a log file of passed tests and a log file of failed tests), we merge log traces from multiple log files into one log file and then sort log traces by their timestamp.

Thread ID is used to form log groups. A log group is a group of log traces generated by a single thread of execution. Because tasks can execute concurrently, their generated log traces may be interleaved in the log file [10]. To form log sequences that indicate correct program execution flows, we group log traces by their thread ID to construct log groups.

Location is the module name and line number of the log statement from which the log trace is generated. We extract the location of the log trace to form the pattern of `module@lineno`. Thus, for each log group, we build a log sequence (a concentration of such patterns) with respect to the timestamp.

After obtaining log sequences, we chain them to build a string that can be used to match against *LogREs*.

Note that execution log files may contain the log traces generated by external libraries. Since the scope of coverage measurements is typically limited to the subject system's codebase, we exclude the log traces from libraries outside the subject system's codebase.

TABLE 1
Overview of the candidate systems.

System	#Commits	#Files	LOC	#Logs	Log Density	Log Rank
saltstack/salt	113,265	6,207	711,159	8,511	1.20×10^{-2}	1
home-assistant/core	43,494	21,517	1,012,962	7,026	6.94×10^{-3}	2
openstack/nova	59,487	4,340	377,843	2,510	6.64×10^{-3}	3
edx/edx-platform	60,737	9,246	377,613	1,764	4.67×10^{-3}	4
cloudera/hue	34,120	8,050	1,611,784	3,894	2.42×10^{-3}	5
zulip/zulip	40,693	6,112	181,406	200	1.10×10^{-3}	6
dimagi/commcare-hq	158,124	7,516	393,528	311	7.90×10^{-4}	7
demisto/content	31,266	30,948	679,235	55	8.10×10^{-5}	8
ansible/ansible	46,249	5,583	143,326	3	2.09×10^{-5}	9
frappe/erpnext	20,186	29,716	151,811	2	1.32×10^{-5}	10

3.1.3 Phase 3 - Path Analysis

In this phase, we perform regular expression matching for all *LogREs* within the log sequences, creating a coverage database that contains mappings between a code statement and its code coverage (Must-, May-, or Must-Not-Coverage). A statement can have different code coverage labels in different *LogREs*. As suggested by Chen et al. [10], a statement is considered in Must-Coverage if it is labelled as Must-Covered by at least one *LogRE*. A statement is considered in May-Coverage if it is not labelled as Must-Covered by any *LogRE* but as May-Covered by at least one *LogRE*. A statement is considered in Must-Not-Coverage if it is not labelled as Must-Covered or May-Covered, but as Must-Not-Covered by any of the *LogREs*.

3.1.4 Phase 4 – Coverage Estimation

In this phase, we estimate the proportion of the covered code statements in the subject system. We use the coverage database resulting from Phase 3 (Section 3.1.3). Since there are code regions that are labelled as May-Covered, we estimate the lower and upper bound of the coverage of the entire system. The lower bound of coverage excludes May-Covered statements and it is calculated as:

$$\frac{\#\ of\ Must\ labels}{Total\ \#\ of\ labels}$$

On the other hand, the upper bound of coverage includes May-Covered statements and it is calculated as:

$$\frac{\#\ of\ Must\ labels\ + \#\ of\ May\ labels}{Total\ \#\ of\ labels}$$

3.2 Exploratory Evaluation of Log2Cov (Design)

We conduct an exploratory study to evaluate LOG2COV and assess its limitations. We compute its precision, recall, F1 score, as well as the overhead that it incurs. We select COVERAGE.PY as our baseline for comparison because it is the most popular coverage tool for PYTHON [19]. More specifically, we use the coverage report of COVERAGE.PY as our ground truth and compare the coverage status of every statement in our coverage database against the coverage report of COVERAGE.PY.

3.2.1 Studied Systems

We perform our evaluation of LOG2COV over a set of systems written in PYTHON that can benefit from LOG2COV. More specifically, we begin with candidate PYTHON systems that are large and active. We select large systems in terms of the total number of files because we believe that small systems have little to gain from a log-based coverage tool. We also set activity (total number of commits) as a criterion because the total number of commits is closely related to activity density (commits per month and maximum consecutive months with commits) and high-profile repositories tend to have a greater density of activity [37].

We curated our collection of candidate PYTHON systems from systems that are hosted on GITHUB. We started by querying the public GITHUB dataset on GOOGLE BIGQUERY,[5] and filtering systems by their source code language. In total, we obtain 341,097 PYTHON systems. Next, we sort these systems by their number of commits and files. The rationale is to obtain an activity rank based on the number of commits and a size rank based on the number of files. The overall rank for each system is determined by summing its activity and size ranks, and then sorting the systems in descending order by that sum. We then selected ten candidate systems for measuring log density. We measured log density because the log-based code coverage approach is only suitable for systems that have a considerable density of logging statements. A system is selected as a candidate if the repository is not a fork and has over 80 stars (a common heuristic used to fetch mature projects [7]). We started with the top-ranked systems until we had selected 10 systems. The log density was calculated according to Equation 1:

$$Log\ Density = \frac{\#\ Log\ Generating\ Lines\ of\ Code}{\#\ Source\ Lines\ of\ Code} \quad (1)$$

Finally, among the ten candidate systems, we selected the top three systems based on the log density to be our subject systems. As shown in Table 1, SALTSTACK SALT, HOME ASSISTANT CORE, and OPENSTACK NOVA stand out among other candidates. For brevity, we henceforth refer to these systems as SALT, HOME ASSISTANT, and NOVA, respectively.

3.2.2 Execution Scenarios

We select testing as the execution scenario for its ubiquity across our subject systems. Since LOGCOCO was evaluated using testing scenarios [10], our choice ensures the consistency in comparative evaluation of the performance of LOG2COV. We selected available test suites from their GITHUB repositories: unit and integration for SALT, unit for HOME ASSISTANT, and unit and functional for NOVA. It is worth mentioning that we did not choose the number of tests or type of tests as the criteria in the system selection stage because any quantity (or type) of tests is enough to compare the results of COVERAGE.PY with LOG2COV.

For test simulation, we used a consistent format string of

```
"%(asctime)s %(created)f %(levelname)s
↪  %(thread)d [%(name)s@%(lineno)d]
↪  %(message)s"
```

for the log format across all the subject systems, in which the `%(created)f` and `%(thread)d` stand for the timestamp and thread id respectively. To ensure that the log traces can be combined and analyzed properly, it is necessary to have a consistent logging format containing the log message attributes of `created` and `thread`. This is because we

[5] https://cloud.google.com/bigquery/public-data

need to ensure that: (1) the thread id is contained in the resulting log traces to group log traces (see Section 3.1.2); and (2) the timestamp is precise enough to sort the log traces combined from multiple log files (note that `asctime` is in the precision of millisecond, while `created` is the Unix time in the precision of microsecond).

Meanwhile, each test suite was executed with DEBUG-level logging and Coverage.py to collect the execution logs and coverage reports. Since we intend to evaluate Log2Cov with different log densities, we filtered the DEBUG logs to simulate INFO-level verbosity. We avoided re-running tests at INFO verbosity to ensure consistency in the execution logs for our comparative analysis of Log2Cov's efficiency under varying log verbosity levels. Compared to INFO verbosity, DEBUG verbosity leads to an increase in both the density of log statements within the source code, with increases ranging from 52% to 91%, and the volume (lines of log traces) of generated execution logs, with increases ranging from 130% to 719%.

3.2.3 Benchmarking Results

As Log2Cov operates at line level, we use the line level results of Coverage.py as our ground truth to compute performance scores for both Must-Coverage and Must-Not-Coverage of Log2Cov. Since May-Coverage occurs, we perform analyses that (a) exclude May-Coverage and (b) consider May-Coverage as positive in the calculation. Considering May-Coverage as positive means equating it to Must-Coverage for Must-Coverage assessments and to Must-Not-Coverage for Must-Not-Coverage assessments.

3.2.4 Overhead Measurement

We aim to assess the overhead incurred by Log2Cov and compare it with that of Coverage.py. We focus on both online and offline overhead of execution time. Our experiments were conducted on a server equipped with an Intel(R) Xeon(R) CPU E5-1620 @ 3.60GHz and 64GB of RAM. We ran the selected execution scenarios (see 3.2.2). Each execution scenario was executed five times for both Coverage.py and Log2Cov overhead measurements.

Online Overhead is measured by quantifying the percentage increase in program execution time with Coverage.py and logging enabled. We define Log2Cov's online overhead as logging overhead because execution logs are the only runtime requirement. To establish an upper bound for Log2Cov's online overhead, we measured the overhead associated with DEBUG-level logging, as it is the most verbose logging level in the standard Python logging parlance and thus represents the most resource-intensive scenario. For the baseline of logging overhead, we configured the log level to be 1000 as it is larger than the numeric value of any standard logging level, practically silencing all logging messages and representing the minimal logging overhead scenario.

Offline Overhead, not directly affecting the system operations, is measured by the time Coverage.py and Log2Cov take to generate coverage results. Specifically, Coverage.py's time to produce an XML report based on collected data and Log2Cov's time for completing the phases of Program, Log, and Path Analysis (As shown in Section 3.1). We exclude Log2Cov's Coverage Estimation phase because it contributes minimally to the overall processing time due to its low complexity, and does not align with Log2Cov's core functionality of producing a coverage database.

TABLE 2
Exploratory Evaluation Result: Precision, Recall, and F1 Score

Test	Log Level	Precision (%)				Recall (%)				F1 Score (%)			
		Must		Must-Not		Must		Must-Not		Must		Must-Not	
		w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o
S U	DEBUG	94	55	85	36	91	85	90	79	92	66	87	49
S U	INFO	94	51	80	38	85	75	92	85	89	60	86	53
S I	DEBUG	84	57	86	56	92	89	72	63	88	70	79	60
S I	INFO	86	67	76	45	91	89	66	53	88	76	71	49
H U	DEBUG	100	53	69	3	93	87	98	72	96	66	81	6
H U	INFO	100	55	42	3	95	91	92	44	97	69	57	6
N U	DEBUG	99	76	45	19	94	92	80	62	96	83	58	29
N U	INFO	98	75	39	10	90	87	82	55	94	80	53	17
N F	DEBUG	95	69	75	46	93	90	83	74	94	78	78	57
N F	INFO	97	78	74	46	87	84	94	90	92	81	83	61

3.3 Exploratory Evaluation of Log2Cov (Results)

Below, we present the results of our exploratory study.

3.3.1 Precision, Recall, and F1 Score

Our analysis reveals that some lines were not reported as covered or uncovered by Coverage.py. Specifically, we observed that these excluded lines were the lines broken up by the practice of *line continuation*,[6] as well as the first line of doc-strings (i.e., `"""`). Since it is impractical to manually inspect all such lines, we report two kinds of metrics for Must- and Mut-Not-Coverage. One metric considers the excluded lines to be true positive (correctly labelled by Log2Cov), while the other considers them as false positive (incorrectly labelled). For example, under the precision part in Table 2, the left column of Must-Coverage shows the precision for which we consider lines excluded by Coverage.py are correctly labelled, while the right column shows the precision for which we consider lines excluded by Coverage.py are incorrectly labelled.

Table 2 shows the result of precision, recall and F1 score when not considering May-Coverage. Log2Cov achieved high precision for Must-Coverage measurement, ranging from 84% to 100%. These precision measurements are on par with those of Chen et al. [10], who observed Must-Coverage precision values of 83%-100% when comparing LogCoCo to JaCoCo (a popular instrumentation-based coverage tool for Java) in 6 Java systems. The precision of Must-Not-Coverage for the systems Home Assistant and Nova are lower than that of Salt, note that the log density of Salt is greater than that of Home Assistant and Nova. When comparing within the same testing scenario, the precision of Must-Not-Coverage is lower than the precision of the Must-Coverage in most cases. Our observation of Must-Not-Coverage precision does not align with that of Chen et al. [10], who observed Must-Not-Coverage precision to be 100% in all studied systems. However, we conclude that log-based coverage measurement performs better for Must-Coverage than for Must-Not-Coverage in terms of precision. This is because the lack of log statements inside conditional branches negatively affects the precision of Must-Not-Coverage, particularly when conditional branches are partially logged. For example, if there are log statements

[6] https://peps.python.org/pep-0008/

only in the if branch but not in the else branch, log-based coverage measurement infers that the else branch does not get executed, even though it may have been executed in certain system execution flows.

Table 2 also indicates how the precision, recall, and F1 score are influenced by the change in log density. We observed that the precision for Must-Coverage and recall for both Must- and Must-Not-Coverage remains largely consistent when reducing the log density (from DEBUG to INFO verbosity). This suggests that LOG2COV's performance, in terms of recall for both Must- and Must-Not-Coverage and precision for Must-Coverage, is largely unaffected by additional log density beyond the INFO level of logging. The precision for Must-Not-Coverage decreased by up to 27 percentage points with the reduction of log density. This trend aligns with our finding that Must-Not-Coverage precision relies more heavily on the density of logging statements. Fewer logging statements, as is the case with INFO verbosity, result in reduced precision in identifying unexecuted code. We also observed that F1 scores vary across different settings (i.e., test type, log level). We believe that this is the case because precision and recall vary across different settings. Each row in Table 2 represents the evaluation metrics for a specific setting (i.e., execution scenario and log level). Different settings may activate distinct execution paths, each with varying log placement and log densities. Under these conditions, it is expected that we observe variance in the evaluation metrics (Precision, Recall, and F1 score). Meanwhile, an increase in uncertainty was also observed when the log verbosity was decreased. Indeed, the decrease in verbosity results in a transfer of Must- and Must-Not-Coverage to May-Coverage. Specifically, under the five testing scenarios across the three subject systems, we observed that the magnitude of the transfer reached up to 3% when the verbosity decreased from the DEBUG to the INFO level. This increase in uncertainty is expected, given that fewer logging statements being recorded equates to less information being available for analysis. Our observation aligns with the implication drawn in the LOGCOCO's paper, i.e., that additional instrumentation of logging can reduce the amount of May-Coverage [10]. Based on these findings, we conclude that systems with denser logging (e.g., using DEBUG verbosity) allow execution to generate more logs. The increase in data points enhances the accuracy and certainty of log-based coverage measurement, particularly regarding the precision of Must-Not-Coverage and a reduction of uncertainty.

When considering May-Coverage as positive, we observed that the precision of Must-Coverage dropped while the precision of Must-Not-Coverage increased. Whether the May-Coverage is covered or not depends on the execution scenarios. In our study, the changes in precision imply that May-covered lines are more likely to be not covered. We also observed that the recall for both Must- and Must-Not-Coverage increases when we consider May-Coverage as positive. This observation aligns with our intuition that recall tends to increase when May-Coverage is considered. Due to the page limit, we include the table with May-Coverage as positive in our replication package.[7]

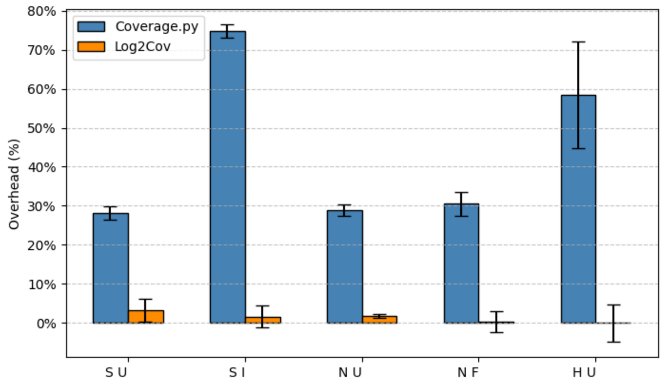

Fig. 2. Online overhead of Coverage.py and Log2Cov.

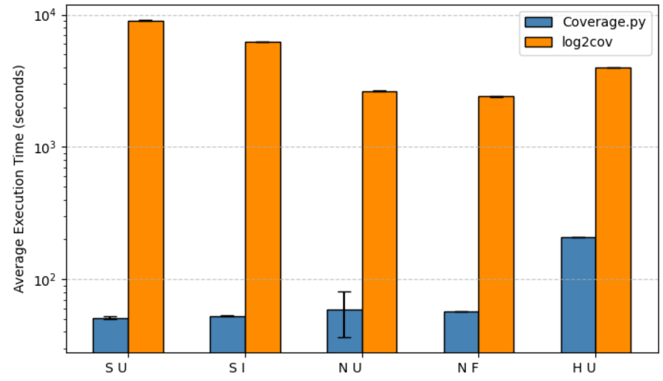

Fig. 3. Offline overhead of Coverage.py and Log2Cov.

3.3.2 Overhead

Figure 2 shows the online overhead of COVERAGE.PY and LOG2COV for the five test scenarios. Within each test scenario, the figure shows the average (mean) performance overhead (in percentage) as well as the confidence intervals across five repeated runs. The results indicate that the average overhead for COVERAGE.PY across different test suites ranges from 28% to 75%. This level of overhead shows the large performance cost that instrumentation-based techniques incur. In contrast, we observed that logging overhead is minimal (<3% on average) across all scenarios. This demonstrates that, in practice, even the most verbose logging setting (DEBUG level) imposes considerably less overhead than COVERAGE.PY in all scenarios. Meanwhile, we performed the Mann-Whitney U test for comparing the overhead distributions of COVERAGE.PY and LOG2COV across the five execution scenarios, with each scenario treated as an independent test. By applying Holm-Bonferroni correction, we adjusted the significance level to 0.01 (0.05 divided by 5). The result shows that COVERAGE.PY introduced a significantly larger overhead than LOG2COV in every execution scenario. Considering that online overhead is a critical factor for a running system, we argue that log-based coverage measurement, as implemented in LOG2COV, can directly address the problem of the large performance cost that is incurred by instrumentation-based coverage measurement techniques.

[7] https://zenodo.org/doi/10.5281/zenodo.7729838

Figure 3 shows that the offline overhead of LOG2COV is significantly higher than that of COVERAGE.PY, which can extend to several hours in some scenarios. The offline overhead for LOG2COV and COVERAGE.PY vary across execution scenarios. The variations in LOG2COV's processing time within the same subject system are due to the variations of the overhead in Path Analysis phase, as Program Analysis and Log Analysis phases are identical in the context of processing the same subject system. Despite the higher offline overhead, LOG2COV's processing can be scheduled during non-peak hours to mitigate the negative impact on user experience. Users can further decompose LOG2COV's execution by executing each of its phases separately. For example, Program Analysis phase can be scheduled upon the update of the system under analysis. When there is a need to measure coverage for a specific execution scenario, users can proceed with Log Analysis and Path Analysis phases.

3.4 Limitations of Log-based Coverage Measurement

Although LOG2COV achieved high precision, recall, and F1 score in measuring statements labelled as Must-Coverage and Must-Not-Coverage, we noticed two areas in which there is plenty of room for improvement.

3.4.1 Uncertainty

Our exploratory evaluation reveals that, across all execution scenarios, the proportion of lines labeled as May-Covered reached up to 69%. This level of uncertainty is likely beyond what users would tolerate. Chen et al. [10] stated that it is the developer's responsibility to add logging statements to reduce such uncertainty. In theory, uncertainty can be addressed by logging in a complete way (i.e., logging entry into every conditional branch and exception handling block). However, such an approach is impractical in real-world applications. Logging extensively across the system can lead to unacceptable performance overhead and inflation of execution logs, which poses challenges for log retention, problem diagnosis and automated log analysis.

Although there are studies suggesting logging strategies [13, 26, 27, 47, 50, 52] and analyzing developers' logging practices [8, 14, 48], logging is often incomplete and inconsistent across systems. Thus, we propose an approach to reduce uncertainty without requiring additional logging. Our approach not only circumvents the practical challenges of extensive logging but also makes log-based coverage measurement more adaptable across systems and environments. The intuition behind our approach is that branching expressions typically test variable values (e.g., `if (a > 10)`, where `a` is a variable), and that the runtime values of these variables may be inferred by the prior log messages. Hence, to infer variable settings (and heuristically evaluate the branching expression), we propose an application of program slicing and data flow analysis.

3.4.2 Imprecision

For mislabelled lines, we randomly selected a set of them for manual root cause inspection. For example, if a line `foo@10` was mislabelled as Must-Covered, we used the execution log file of LOG2COV (not from the SUT) to retrieve the *LogRE* of which the Must-Coverage contains this line. We also identified the corresponding method that was the entry point for LOG2COV to produce such a *LogRE*. We then manually inspected the program execution flows starting from that method to reason about the cause of mislabelling. After inspecting 200 examples, we observed the following repetitive root causes of mislabelling: 1) the practice of dependency injection and 2) the lack of logging statements in conditional blocks.

Although we did not identify causes of imprecision that are related to syntax and specifications of PYTHON, we recognize that they may impact the precision of log-based coverage measurement. This is because log-based coverage measurement relies on call graphs to simulate program execution flow, and the process of call graph generation can be inherently sensitive to syntax variations. Imprecision in call graph generation can result in inaccurate function call invocations, thereby impacting the precision of coverage measurement. In the case of PYTHON, which is known for its dynamic features, creating an accurate call graph is particularly challenging. To address this issue, we used PYCG (see Section 3.1.1). PYCG is adept at handling PYTHON's dynamic characteristics, including modules, generators, function closures, and multiple inheritance. It has reported a precision of 99.2% and a recall of 69.9%. This high precision of PYCG may help to explain why we did not detect cases where inaccurate function call invocation leads to imprecision.

Without requiring additional logging statements, improvements can be made to resolve the imprecision caused by dependency injection. In unit testing, the practice of mocking and patching techniques (a type of dependency injection) can cause imprecision of the measurement for both Must- and Must-Not-Covered labels. Mocking is a common approach in object-oriented software development to simulate software dependencies, speed up the testing process, and confine the scope of testing to the component under test [32]. According to PYTHON documentation,[8] patching is used for replacing methods and attributes of existing objects with mocks. An internal method in the codebase can be replaced by a mocked object using the patching technique, and this diverts away from the statically defined flows. In our replication package,[7] we provide an example of how dependency injection and insufficient logging lead to mislabelling, as well as a detailed explanation.

4 RQ1: TO WHAT EXTENT CAN WE REDUCE THE UNCERTAINTY OF LOG-BASED CODE COVERAGE?

Users have come to expect coverage measurements to be precise about the status of program elements, i.e., statements are either covered or not covered; however, the log-based coverage approach may report that the status of a block of program elements is uncertain (i.e., May-Coverage). This uncertainty tends to arise when log statements are absent from conditional blocks and exception handling blocks. While log statements in such blocks provide hints about the execution of statements within them, they are not the only method that an observer can use to determine whether the

[8] https://docs.python.org/3/library/unittest.mock.html

Algorithm 1: Trace definition of a targeting variable

```
   Input: var, def-use chains, call graph
   Output: resultMap
 1 Function Slicing(var):
 2     Initialize an empty map resultMap
 3     defList ← GetDEFs(var, def-use chains)
 4     if defList is not empty then
 5         foreach DEF in defList do
 6             if DEF contains other variables then
 7                 foreach otherVar in DEF do
 8                     varMap ← Slicing(otherVar)
 9                     Merge varMap into resultMap
10             else
11                 if var not in resultMap then
12                     resultMap[var] ← empty list
13                 Append DEF to resultMap[var]
14     else
15         if var is a function parameter then
16             callerFuncs ← GetCallers(var, call
                graph)
17             foreach callerFunc in callerFuncs do
18                 arg ← GetArg(var, callerFunc)
19                 Initialize an empty map varMap
20                 if arg contains variables then
21                     foreach otherVar in arg do
22                         varMap ← Slicing(otherVar)
23                 DEF ← BuildDef(arg, varMap)
24                 if var not in resultMap then
25                     resultMap[var] ← empty list
26                 Append DEF to resultMap[var]
27     return resultMap
```

statements inside such blocks are actually executed during runtime. For example, practitioners may combine hints from log messages with a careful inspection of the codebase to reason about the settings of variables that are referenced in conditional expressions, which may ultimately identify the path of execution. Inspired by this intuition about how practitioners analyze logs and code, we propose an approach to mitigate the uncertainty of log-based coverage by resolving May-Coverage caused by `if-else` conditional blocks. We call our approach "Resolve May-Coverage".

To evaluate our approach, we apply "Resolve May-Coverage" to the coverage database of DEBUG-level scenarios obtained in our exploratory evaluation (see Section 3.3). We compare each line in the resolved coverage to the report generated by COVERAGE.PY. Meanwhile, we measure the amount of May-Coverage that resides in conditional branches of which the conditional statement is Must-Covered, and we refer to this as "Resolvable Coverage". Additionally, we measure the execution time of "Resolve May-Coverage" phase, repeating five times for each execution scenario. Subsequently, we use LOG2COV's execution time (reported in Section 3.3.2) to assess the offline overhead attributed to this phase. The details of our approach are explained below.

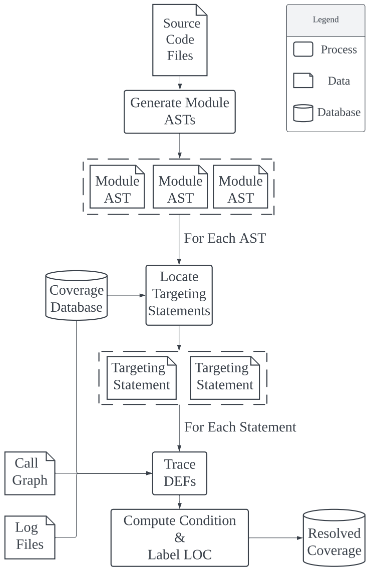

Fig. 4. Resolve May-Coverage Phase.

Approach. To resolve May-Coverage statements, we append the "Resolve May-Coverage" phase to the "Path Analysis" phase of Section 3.1.3. This phase takes as input the system's source code, call graph, execution logs, and the coverage database generated by the "Path Analysis" phase to produce a mapping of resolved May-Covered statements to their coverage status. Figure 4 provides an overview of the "Resolve May-Coverage" phase, which we describe below.

(1) Locate targeting statements: We locate the conditional statements of interest by analyzing the system's codebase and traversing the module's AST to identify `if-else` statements. We query the coverage database to determine if the conditional statement is Must-Covered and the associated code region, i.e., the statements within conditional blocks, is May-Covered. We refer to such an `if-else` statement as the targeting statement, and the variable within the targeting statement as the targeting variable.

(2) Trace the definition of variables in the targeting statement: We perform backward slicing [15] and data-flow analysis based on the module's AST and the module level `def-use` chains. The `def-use` chains are computed using BENIGET.[9] Each `def-use` chain links an identifier's `USE` to its `DEF` [45].

For each targeting variable, we trace its `DEF`. As illustrated in Algorithm 1, we begin by finding the closest `DEF` or `USE` of the targeting variable, where the `USE` needs to be a log statement that logs the targeting variable (line 3). If a `USE` is found, we construct a `DEF` using the value extracted from the log trace. A targeting variable can have multiple `DEF`s identified because there can be multiple definitions coexisting under different conditions. These definitions do not reassign the variable but rather establish different initial values for it depending on the execution flow. In cases where other variables are involved in the collected `DEF`, we add those variables as targeting variables and recursively apply backward slicing to trace their `DEF`/`USE` (lines 6-9).

[9] https://github.com/serge-sans-paille/beniget

TABLE 3
Performance of Resolve May-Coverage.

Test	Resolvable	Resolved	Accuracy	Overhead
S U	401	44 (11%)	100%	2%
S I	83	2 (2%)	100%	1%
N U	559	13 (2%)	100%	12%
N F	92	2 (2%)	100%	4%
H U	581	5 (1%)	100%	7%

If we cannot identify the DEF/USE based on the module-level def-use chain, we check if the targeting variable is a function parameter (line 15). If so, we use the call graph (as obtained in phase 3.1.1) to locate the caller functions of the function in which the targeting variable resides (line 16). Note that we collect the caller function only if the statement of that function call is labelled as Must-Covered in the original coverage database. For each collected caller function, we trace the function call to obtain the value of the targeting variable, which is provided as an argument, and then construct its DEF (lines 17-26).

After tracing the DEF of each variable in the targeting statement, we obtain a mapping that links each variable to its corresponding DEFs, preserving the dependency order.

(3) Compute the condition for the targeting statement and label LOCs: We apply the Cartesian product to generate all possible combinations of values for the variables in the targeting statement. For each combination, we construct and evaluate the code snippet representing the conditional expression of the targeting statement. The overall condition result is determined by evaluating the conditional expression with different variable inputs, leading to three possible outcomes: *True*, *False*, or both. When the outcome is either *True* or *False*, it indicates that all input combinations result in the same conditional value. When the outcome is both *True* and *False*, it means that the expression evaluates to *True* for some input combinations and *False* for others.

After obtaining the branch selection result of an if-else condition, we label the statements in the branches accordingly. For example, if the branch selection result is computed as *True*, we label the statements in the if-branch as Must-Covered; and if there are any elif branches or an else branch, then the statements in those branches are labelled as Must-Not-Covered. In addition, we label the statements of the callee function in the conditional branches if they are initially labelled as May-Covered. Similar to the PATH ANALYSIS phase, a statement is considered as Must-Coverage if it is labelled as Must-Covered by at least one analysis of the targeting statement, and a statement is considered as Must-Not-Coverage if no analysis of the targeting statement labels it as Must-Covered. An example demonstrating our approach to resolving uncertainty is provided in our replication package.[7]

Results. Table 3 provides an overview of the performance. The column "Resolvable" indicates the quantity of May-Coverage that is identified as being able to resolve, and the column "Resolved" shows the quantity of May-Coverage resolved to Must-Coverage or Must-Not-Coverage by our approach. In SALT, NOVA, and HOME ASSISTANT, our approach achieved 100% accuracy in resolving May-Coverage compared with the report of COVERAGE.PY. Meanwhile, we found that COVERAGE.PY excluded some lines. Since we manually inspected all such cases, there is no need to consider the excluded lines as incorrectly labelled. Furthermore, we observed that the "Resolve May-Coverage" phase imposes a minimal offline overhead, with mean values ranging from 1% to 12% across the execution scenarios.

Discussion. After conducting an evaluation on our approach of "Resolve May-Coverage" with five testing scenarios across the three subject systems, our approach has demonstrated its effectiveness in resolving May-Coverage. The accuracy of 100% suggests that our approach is capable of accurately resolving May-Coverage in different contexts, making our approach reliable for practitioners seeking to improve log-based code coverage without requiring additional logging statements in log-sparse conditional blocks.

While the quantity of resolvable May-Coverage resolved by our approach was up to 11%, this still represents a considerable reduction in the uncertainty of log-based code coverage. Additionally, we observed that the number of May-Covered lines resolved tended to be consistent across the three subject systems, with a difference of only 10 percentage points. One important factor to note is that our approach skips evaluating conditions if the targeting variables are dependent on function calls, except for log statements, during the backward slicing process. This ensures that the performance of our approach is not hindered by dependencies that are outside of the control of the SUT. As seen in Table 3, the quantities of resolved May-Coverage suggest that the involvement of function calls in the data flow of the targeting variable is large and similar across the subject systems.

In addition, our approach does not resolve May-Coverage incorrectly. This means that practitioners can confidently use our approach without fear of negatively affecting the performance of LOG2COV. However, if practitioners wish to expand the resolvable uncertainty and the resolved quantity of such uncertainty, future work can be done on handling conditional statements and exception-handling blocks by the data flow analysis that involves external dependencies. This could potentially improve the performance of our approach in resolving the uncertainty in log-based code coverage measurement.

5 RQ2: TO WHAT EXTENT CAN WE REDUCE THE IMPRECISION OF LOG-BASED CODE COVERAGE?

Imprecision is hardly accepted in an analysis where outcomes must be assertive. Without requiring developers to add logging statements to the source code, we explore the extent to which imprecision can be improved by addressing dependency injection, in particular by identifying methods that are replaced by mock objects during unit testing.

Approach. We modify the Program Analysis phase of LOG2COV by adding a new step namely "Remove Dependency". This step takes the linked ASTs (see Section 3.1.1) as input and removes the ASTs of the methods that are identified as being replaced by mock objects with the patching technique (see Section 3.4.2). We explain the details of the "Remove Dependency" step below.

TABLE 4
Performance of Log2Cov w&wo resolving dependency injection.

System	Resolve Dependency	Precision			
		Must		Must-Not	
Salt	No	94%	55%	85%	36%
	Yes	**95%**	**56%**	**87%**	**38%**
Home Assistant	No	100%	53%	69%	3%
	Yes	100%	**54%**	69%	3%
Nova	No	99%	76%	45%	19%
	Yes	99%	76%	**49%**	**20%**

(1) Dependency Identification: In this step, we identify targeting methods (methods that are replaced by mock objects by analyzing all the test modules of each subject system). For each test module, we traverse its AST to identify the use of patching. In Python, such a usage pattern invokes either the `patch()` or the `patch.object()` function of the Python's `mock` library.[8] The difference between these two functions is how they resolve the naming hierarchy of the targeting method. The `patch.object()` function requires the module/class containing the targeting method to be imported before patching, and `patch()` takes a string of a path and resolves it to a method. To collect the full relative path to a targeting method, we build a mapping between the module/class name and its relative path by analyzing the import-related nodes in the AST of the test module. In our replication package,[7] we provide an example of patching that demonstrates the use cases of the two different functions, as well as the result of the dependency identification.

(2) Dependency Removal: In this step, we remove the AST files of the methods that are patched with mock objects. This step ensures the Program Analysis phase does not traverse to them, which causes imprecision.

We evaluate Log2Cov on the same subject systems with the updated Program Analysis phase. Specifically, we re-execute the Program Analysis and Path Analysis phases of Log2Cov to obtain the new coverage database for DEBUG-level scenarios. We then compare the precision of Must- and Must-Not-Coverage with our prior results (Section 3.2), respectively. In addition, we measure the execution time of the Program Analysis and Path Analysis phases, both with and without "Remove Dependency", to understand its impact on offline overhead.

Results. Table 4 demonstrates the results of our approach. There is a 1 percentage point increase in the precision of Must-Coverage, and a 2 percentage point increase in Must-Not-Coverage precision in Salt. In Nova, there is an improvement of up to 4 percentage points. Regarding Home Assistant, there is a 1 percentage point increase in the precision of Must-Coverage when we consider the lines excluded by Coverage.py are incorrectly labelled. Moreover, the mean execution time for Program Analysis and Path Analysis, when applying the "Remove Dependency" step, is consistently shorter compared to scenarios excluding this step. This indicates that our approach introduces no additional offline overhead.

Discussion. We observed that Salt benefited the most from our approach. Since the amount of dependency injection can vary in different systems, it is reasonable that the improvements of our approach vary. Although improvements are not guaranteed, our approach does not harm the performance of Log2Cov in any subject systems, suggesting it is at the very least safe to be consistently enabled.

6 Threats to Validity

Construct Validity. In this paper, we use Coverage.py to obtain the "ground-truth" of coverage status. Coverage.py may not reflect the exact coverage, since it may contain bugs itself. However, Coverage.py is quite mature and stable, and is the de facto standard coverage tool recommended by the official Python documentation.[10]

Internal Validity. We evaluate Log2Cov using the Salt, Home Assistant, and Nova systems, which have the greatest log density, and are among the largest and most active Python systems on GitHub. Our study results may only reflect the performance of our approach to systems that have a sufficient amount of logging statements. However, the very nature of the log-based coverage measurement is to leverage log traces. Hence, log-based coverage measurement is not applicable to every system. Indeed, while this paper sets out to improve the imprecision and uncertainty of log-based coverage measurements when entering log-sparse areas of code, we still rely upon log traces to minimize the scope in which our static analyses will be performed.

External Validity. We adapt the log-based coverage measurement to Python. Our results may not generalize to systems written in other languages (e.g. JavaScript). However, the primary technical requirements of porting log-based coverage to another language are AST parsers and static call graphs, which are generally available for many programming languages. While the amount of effort required is not small, it is not an insurmountable challenge.

We propose a solution to the imprecision problem by addressing the challenge of patching in unit testing. Since patching is a subset of dependency injection, our approach may be limited to measuring coverage in a unit testing scenario. However, the core of our approach is to make Log2Cov infer the system execution flow as expected in the execution scenarios. In the unit testing scenario, we identify the methods replaced by mock objects, and remove the identified methods from both the call graph and the linked ASTs so that Log2Cov does not have the knowledge that those methods are involved in the system execution flow. For other dependency injection scenarios, we only need to identify the injection pattern and identify the methods being injected, then modify the call graph accordingly. For example, we can add the dependency as a callee of the method that is being injected. However, the idea of manipulating the call graph and correcting the execution flow in the analysis remains the same. Therefore, we believe our approach is applicable to any dependency injection technique.

7 Conclusions and Implication

Code coverage is a common measurement that practitioners rely upon. While modern code coverage tools provide valuable insights, they impose a performance overhead due to code instrumentation. In this study, we have demonstrated

[10] https://docs.python.org/3/library/trace.html

that log-based code coverage tools offer a promising alternative to traditional code instrumentation approaches for PYTHON systems. By developing and evaluating LOG2COV, we have shown that analyzing the program and its execution logs can be an effective way to measure coverage for PYTHON systems, and improvements of log-based coverage measurement can be made without requiring additional logging instrumentation from developers. To the best of our knowledge, our approach is the first work that directly addresses the shortcomings of imprecision and uncertainty of log-based coverage measurement. While our enhancements make substantial improvements to mitigate such shortcomings, future work is still needed to achieve parity with instrumentation-based coverage approaches. For example, we are actively exploring using a similar approach as we resolve uncertainty to evaluate the `if-else` condition that results in false negative (mislabelled Must-Not-Coverage) to further reduce imprecision and leveraging the development environment of the SUT, which contains required dependencies, to further reduce uncertainty. Moreover, our natural next step is to combine the log-based approach and the instrumentation-based approach—using LOG2COV in general and instrumentation-based approach (COVERAGE.PY) in log-free code block to complement the log-based result.

ACKNOWLEDGMENTS

The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Huawei and/or any of its subsidiaries or affiliates. Moreover, our results do not in any way reflect the quality of Huawei's products.

REFERENCES

[1] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, 1976.

[2] M. Alves and H. Paula, "Identifying logging practices in open source python containerized application projects," in *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, 2021.

[3] P. Ammann and J. Offutt, "Introduction to software testing edition 2," 2017.

[4] P. Arafa, G. M. Tchamgoue, H. Kashif, and S. Fischmeister, "Qdime: Qos-aware dynamic binary instrumentation," in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2017.

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, 2014.

[6] I. Baxter, "Branch coverage for arbitrary languages made easy," 2002. [Online]. Available: http://www.semdesigns.com/Company/Publications/TestCoverage.pdf

[7] S. Biswas, M. J. Islam, Y. Huang, and H. Rajan, "Boa meets python: A boa dataset of data science software in python language," in *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2019.

[8] B. Chen *et al.*, "Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, 2017.

[9] B. Chen and Z. M. Jiang, "Extracting and studying the logging-code-issue-introducing changes in java-based large-scale open source software systems," *Empirical Software Engineering*, vol. 24, 2019.

[10] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. J. Jiang, "An automated approach to estimating code coverage measures via execution logs," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[11] K.-R. Chilakamarri and S. Elbaum, "Reducing coverage collection overhead with disposable instrumentation," in *15th International Symposium on Software Reliability Engineering*, 2004.

[12] P. K. Chittimalli and V. Shah, "Gems: a generic model based source code instrumentation framework," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012.

[13] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A {Cost-Aware} logging mechanism for performance diagnosis," in *2015 USENIX annual technical conference (USENIX ATC 15)*, 2015, pp. 139–150.

[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*, 2014.

[15] M. Harman and R. Hierons, "An overview of program slicing," *software focus*, vol. 2, 2001.

[16] C. Häubl, C. Wimmer, and H. Mössenböck, "Deriving code coverage information from profiling data recorded for a trace-based just-in-time compiler," in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 2013.

[17] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[18] J. Holmes, A. Groce, J. Pinto, P. Mittal, P. Azimi, K. Kellar, and J. O'Brien, "Tstl: the template scripting testing language," *International Journal on Software Tools for Technology Transfer*, vol. 20, 2018.

[19] A. Hora, "What code is deliberately excluded from test coverage and why?" in *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2021.

[20] D. Jackson and M. Rinard, "Software analysis: A roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000.

[21] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *2008 IEEE International Conference on Software Maintenance*, 2008.

[22] A. Juvonen, T. Sipola, and T. Hämäläinen, "Online anomaly detection using dimensionality reduction techniques for http log analysis," *Computer Networks*, vol. 91, 2015.

[23] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure data analysis of a lan of windows nt based com-

puters," in *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, 1999.

[24] P. S. Kochhar, F. Thung, D. Lo, and J. Lawall, "An empirical study on the adequacy of testing in open source projects," in *Asia-Pacific Software Engineering Conference*, 2014.

[25] N. Kumar, B. R. Childers, and M. L. Soffa, "Low overhead program monitoring and profiling," *ACM SIGSOFT Software Engineering Notes*, vol. 31, 2005.

[26] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Software Engineering*, vol. 22, 2017.

[27] Z. Li, T.-H. Chen, and W. Shang, "Where shall we log? studying and suggesting logging locations in code blocks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[28] R. Lingampally, A. Gupta, and P. Jalote, "A multipurpose code coverage tool for java," in *Hawaii International Conference on System Sciences (HICSS)*, 2007.

[29] J. Mace and R. Fonseca, "Universal context propagation for distributed system instrumentation," in *Proceedings of the thirteenth EuroSys conference*, 2018.

[30] L. Mariani, D. Hao, R. Subramanyan, and H. Zhu, "The central role of test automation in software quality assurance," *Software Quality Journal*, vol. 25, 2017.

[31] S. Messaoudi, D. Shin, A. Panichella, D. Bianculli, and L. C. Briand, "Log-based slicing for system-level test cases," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 517–528.

[32] S. Mostafa and X. Wang, "An empirical study on the usage of mocking frameworks in software testing," in *International Conference on Quality Software*, 2014.

[33] J. Mußler, D. Lorenz, and F. Wolf, "Reducing the overhead of direct application instrumentation using prior static analysis," in *European Conference on Parallel Processing*, 2011.

[34] H. A. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. V. Nguyen, "Interaction-based tracking of program entities for test case evolution," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017.

[35] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *IEEE/IFIP international conference on dependable systems and networks (DSN)*, 2007.

[36] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer, "Improving log-based field failure data analysis of multi-node computing systems," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, 2011.

[37] P. H. Russell, R. L. Johnson, S. Ananthan, B. Harnke, and N. E. Carlson, "A large-scale analysis of bioinformatics code on github," *PloS one*, vol. 13, 2018.

[38] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "Pycg: Practical call graph generation in python," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021.

[39] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process*, vol. 26, 2014.

[40] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *IEEE International Conference on Software Engineering (ICSE)*, 2013.

[41] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. K. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[42] E. Sun and D. Kaeli, "A binary instrumentation tool for the blackfin processor," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009.

[43] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads," *Automated Software Engineering*, vol. 24, 2017.

[44] M. M. Tikir and J. K. Hollingsworth, "Efficient instrumentation for code coverage testing," *ACM SIGSOFT Software Engineering Notes*, vol. 27, 2002.

[45] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, 1984.

[46] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, 2009.

[47] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[48] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *IEEE International Conference on Software Engineering (ICSE)*, 2012.

[49] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, 2012.

[50] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "The game of twenty questions: Do you know where to log?" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017.

[51] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, 1997.

[52] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 415–425.