Enhancing Log-Based Software Monitoring with Static Source Code Analysis

Vincent Therrien
vincent.therrien@polymtl.ca
Département de génie informatique et de génie logiciel, Polytechnique Montréal, Canada

***Abstract*—This work aims at improving software monitoring by relying on static source code analysis to interpret logs produced at runtime more effectively. Logs, which document software events, are a prime source of information for software maintenance and have been applied to numerous problems such as anomaly detection, performance analysis, and root cause analysis. However, these approaches seldom consider the *implementation* of the systems and instead use log sequences directly to make predictions. Source code contains valuable information that can be used to put logs in context and better understand how software systems behave internally at runtime. This more complete representation of systems leads to a deeper understanding of software issues that can be applied to applications such as log-based anomaly detection. A publicly available tool called Path Reconstructor is presented. It is shown to reconstruct execution paths in moderately complex systems but has not improved anomaly detection methods compared to simpler alternatives.**

***Index Terms*—Static Source Code Analysis, Execution Path, Log, Anomaly Detection.**

I. Introduction

Logs document software events at runtime, which makes them an essential source of information for maintaining complex systems. Developers include logging statements in the source code of applications to produce a timestamped record when the program executes a particular function. Their simplicity and efficiency makes them suited to a variety of monitoring areas.

- **Anomaly detection** relies on supervised (SVM, LR) or unsupervised methods (IM, PCA) to classify logs as normal or abnormal and signal potentially faulty application behaviors [1],
- **Performance regressions** can be detected by analyzing changes in log patterns [2], and
- **Root cause analysis** leverages logs to identify the location in the source code of erroneous statements [3].

In most cases, DevOps maintenance software uses log sequences as an *indirect* means to monitor system behaviors. The most reliable approach to identify software issues such as anomalies or regressions would be to examine the exact sequence of instructions executed by the program as they are described in its source code. Small programs are often debugged by stepping into their execution flow to obtain *direct* information about the system's internal behavior. Naturally, the large-scale nature of distributed applications makes this method completely impractical. They are monitored almost purely with logs and other metrics such as memory usage.

Applying machine learning techniques to monitor systems using only their logs works to some extent but is subjected to some shortcomings.

- **Unstable log data** (that is, logs that change over the development of a program) modify the obtained sequences and make ML models obsolete. The models thus need to be retrained periodically or devised specifically to account for instability [1].
- **Anomaly detection suffers from uneven performances and requires a lot of data**. Previous studies have evaluated several supervised and unsupervised anomaly detection methods and found that performances as measured with the F1-score vary across datasets. Also, the most efficient methods require labor-intensive manual labeling [4].
- **Monitoring tools detection lacks context**. Linking a decrease in software quality with logs, for example, can be difficult because, without static source code analysis, the monitoring program cannot determine precisely how regressions were introduced in the application.

In short, logs by themselves do not contain enough data to fully understand software systems. This is why researchers have investigated methods to enhance monitoring by leveraging source code. The basic idea is to reconstruct execution paths from logs to better comprehend how the source code works. In theory, obtaining the complete paths would make monitoring much easier, but in practice, it is hindered by a few limitations.

First, it is not possible, in most large systems, to accurately obtain the execution paths. This would require the addition of a logging statement to every single callable unit of a program, leading to unacceptable performance losses. In practice, logs often document significant events such as booting a VM or connecting to a server but not simple operations. It is possible, however, to *approximately* reconstruct execution.

Second, the behavior of parallel systems is not fully captured by logs because they present events sequentially. Parallel execution threads introduce noise in logs that needs to be considered during processing.

The aim of this work is to investigate how useful static source code analysis can be in software monitoring by considering its limitations and potential applications. The following research questions (RQ) are addressed:

- *RQ 1*: How precisely can execution paths be reconstructed from logs and static source code analysis in real software systems?
- *RQ 2*: Can the reconstruction of execution paths improve DevOps maintenance operations?

This article describes theoretical considerations (II), an implementation of the proposed methods (III), and an evaluation on real systems (IV).

II. Theoretical Considerations

This section describes some theory behind static source code analysis and how it is applied. The discussion does not assume the use of a specific programing language.

A. Core Concepts

A *program* is an application composed of callable *subroutines* commonly referred to as functions whose executions are triggered at *call points*. A *call graph* documents the relations between the subroutines of a program. Its nodes correspond to subroutines. A directed edge $A \rightarrow B$ indicates that the source subroutine A calls the target subroutine B. Such a graph is called *sound* if its edges list all possible runtime subroutine calls. It is called *precise* if it does not have any edges that are not possible at runtime [5]. An *execution path* is a sequence of subroutines listed in their execution order.

In most cases, a call graph that is both sound and precise is considered more practical because it can model all possible execution paths without including unnecessary nodes or edges, thus saving memory. Tip and Palsberg [6] reviewed call graph elaboration methods such as Class Hierarchy Analysis (CHA), Rapid Type Analysis (RTA), and Control-Flow Analysis (0-CFA). These methods are sound but imprecise. In fact, achieving precision is a challenge in the presence of dynamic features used by most modern programming languages [5].

B. Interest to Log-Based Monitoring

Reconstructing the execution path of a program with its call graph gives insight about its internal behavior that is not possible with logs alone. As an example, consider two nearly identical log sequences with a single differing log. This entry may indicate that the execution paths are very different. For instance, the dissimilar log might be produced by a library not used in the other sequence. Considering the logs alone hides this information and a machine learning algorithm could consider the two sequences similar even when they result from vastly different execution paths, leading to potentially erroneous predictions.

C. Execution Path Reconstruction Objectives

The execution path of a program can be reconstructed accurately or approximately from runtime logs to improve monitoring. A sound but not necessarily precise call graph is assumed. Also, the discussion is restricted to subroutines that belong exclusively to the program. For example, the methods defined within a program are included in the call graph, but underlying features such as JVM calls in Java programming or STL function calls in C++ programming are not included.

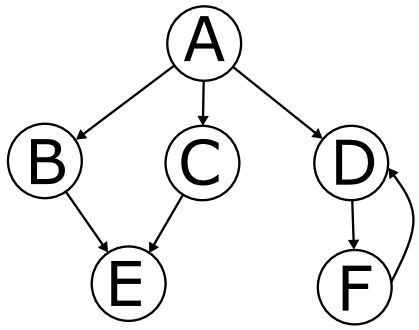

Figure 1: Demonstration call graph with edges $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $B \rightarrow E$, $C \rightarrow E$, $D \rightarrow F$, and $F \rightarrow D$

The execution of a program starts at an *entry point* such as the function `main` in C programming. Control is transferred to target subroutines at call points until they return a value or raise an exception. Figure 1 shows a simple call graph as a demonstration. Node A is the entry point.

An execution path is a sequence of subroutines noted as an ordered list that details how control is transferred at runtime. For instance, if a subroutine A calls a subroutine D which itself calls subroutine F, then the complete execution path is:

$$P = (A, D, F, D, A) \tag{1}$$

One notices that the last elements list the first ones in a reversed order. This is because control flow resumes to the source subroutine when a target subroutine terminates. Execution path can be more compactly noted as:

$$P = (A, D, F) \tag{2}$$

The *precision* of execution path reconstruction is defined as the fraction of correctly evaluated nodes (true positives) over the number of nodes. In other words, it is the ratio of logs that match a subroutine call in the correct order over the number of logs. The *recall* is defined as the fraction of correctly evaluated nodes over the number of subroutine calls. It corresponds to the ratio of logs that match a subroutine call over the number of subroutine calls. Empty elements noted as $\emptyset$ can be inserted in the reconstructed path to match the position of the real path; this operation is denoted as *matching* and is performed by a function noted as $m(P)$. The order of elements is never modified.

Consider the following sequences as an example:

$$\begin{aligned} P_{real} &= (A, B, E, D, F, C) \\ P_{reconstructed} &= (A, D, F, H) \\ P_{matched} = m(P_{reconstructed}) &= (A, \emptyset, \emptyset, D, F, H) \end{aligned} \tag{3}$$

where P_{real} is the true execution path, $P_{reconstructed}$ is the path reconstructed from logs, and $P_{matched}$ is the recon-

structed path with added padding to match subroutine call positions. Precision and recall are evaluated as:

$$precision = \frac{|P_{real} \cap P_{matched}|}{|P_{reconstructed}|} = \frac{3}{4}$$
$$recall = \frac{|P_{real} \cap P_{matched}|}{|P_{real}|} = \frac{3}{6} \quad (4)$$

The goal of execution path reconstruction is to maximize precision and recall to obtain a sequence that matches the original execution path as close as possible.

D. Execution Path Reconstruction Procedure

Given a call graph, a set of *possible execution paths* can be listed. The behavior of a program can be described using only its sequence of possible paths. The following algorithm identifies them:

1) Possible Paths Identification Algorithm:

a **Group cycling nodes**. Nodes recursively connected by edges are grouped. For instance, in figure 1, nodes D and F are grouped into a simplified node DF. The resulting graph does not contain cycles or loops.

b **List paths between indegree 0 and outdegree 0 nodes**. This step yields a list of sequences that start at the source node and end at sink nodes. In figure 1, the sequences are (B, E), (C, E), and DF.

c **List early terminated subroutines**. A target subroutine may terminate and resume to the source subroutine without calling a subroutine to which it is connected in the call graph. For instance, the sequence (B, E) actually comprises two possible sequences: (B, E) and (B).

d **Iterate recursively for cycled nodes**. Nodes grouped at step 1 are transformed into cycle-less graphs by removing the edge that connects to the entry point of the grouped nodes. Steps 1, 2, and 3 are then applied recursively.

Equation 5 shows the result obtained after applying the possible paths identification algorithm on the call graph on figure 1.

$$S_{paths} = \{(B), (B, E), (C), (C, E), (D), (D, F)\} \quad (5)$$

where S_{paths} is the set of possible paths. A call may be *repeated*, if it is comprised within a while or for loop. For instance, if the call point of subroutine E within subroutine B is repeated, then sequences (B, E_n), in which E_n stands for an arbitrary number of subroutine E calls, are also possible. Other sequences such as (E, C) or (B, F) are not possible.

2) Log to Subroutine Matching Equation: Runtime execution paths can be reconstructed from logs with the list of possible paths obtained from static source code analysis. Given a sequence of subroutines P_{logs} obtained by matching logs to call points, the set of possible real execution paths corresponds to matching possible paths as shown in equation 6.

$$S_{real} = \cup_{s \in S_{paths}} s | \{s \cap m(P_{logs}) \neq \emptyset\} \quad (6)$$

For example, if a log corresponding to subroutine E in figure 1 is obtained, two execution paths may have generated it: (B, E) and (C, E).

3) Logging Statements Location Considerations: Equation 6 assumes that logging statements are all located at the beginning of subroutines and directly indicate the execution path. In practice, logging statements can be placed anywhere in a program, even at the very end of a source subroutine after it calls target subroutines.

For instance, consider the example below:

Listing 1: Logging order consideration example

```
def C():
    E()
    print("Executed C")

def E():
    print("Executed E")
```

Executing function C in listing 1 results in the log sequence (E, C), which does not correspond to possible paths listed in equation 5. A *reordering function*, noted $r(P)$, uses the locations of call points and logging statements to reorder subroutines according to their execution order. For example, the sequence (E, C) is converted into (C, E) to obtain the execution order.

This technique can be applied when a logging statement and a call point belong to the same execution block, for example, the same if condition, but not when they belong to distinct blocks, as illustrated in the following example:

Listing 2: Logging blocks example

```
def C():
    if (random.uniform(0, 1) < 0.5)
        E()
    else:
        print("Executed C")

def E():
    print("Executed E")
```

Four types of log-subroutine pairs are defined:

A) *Anterior logging statements* are (1) placed before a call point and (2) on the same block-level or a higher block-level as a call point. The log is necessarily produced before the target subroutine is called, as assumed by equation 5.

B) *Posterior logging statements* are (1) placed after a call point and (2) on the same block-level or a higher block-level as a call point. The log is necessarily produced after the target subroutine is called, as in listing 1.

C) *Nonrestrictive logging statements* are (1) placed in a different block as a call point and (2) on the same or lower block-level. The log or target subroutine may not be produced or executed, respectively. Listing 2 shows such a statement.

D) *Terminal logging statements* are located within subroutines that do not comprise call points. They are treated

as a special case of anterior logging statements in which the target subroutine is an empty set.

The reordering function $r(P)$ relies on static source code analysis to classify logs and reorder them according to subroutine execution order. Posterior statements lead to a reversal of logs. Nonrestrictive statements are followed by wildcards that can be matched with any subroutine connected to the source subroutine. For example, if subroutine C in figure 1 comprises a nonrestrictive statement, it leads to the possible path (C, X), where X can be matched with elements E or $\emptyset$.

The path reconstruction algorithm described by equation 6 is modified with function $r(P)$.

$$S_{real} = \cup_{s \in S_{paths}} s | \{ s \cap m[r(P_{logs})] \neq \emptyset \} \tag{7}$$

III. Software Implementation

The procedure described in section II is implemented in three components. The developed tool is called Path Reconstructor and is publicly available at https://github.com/Vincent-Therrien/path_reconstructor.

A. Call Graph Generation

A call graph describes the relation between the subroutines of a program. Several tools generate such graphs from source and they can usually only analyze applications developed with a particular programming languages. There are two main types of call graph generators. **Static** generators only analyze the source code without executing it. Some of these tools are:

- pyan for Python
- Doxygen for C, C++, or Java
- java-callgraph for Java

The other way to generate call graphs is through **dynamic** analysis. The program is executed and each subroutine call is detected. The call graph is thus generated dynamically and new nodes are created online. Such tools comprise:

- pycallgraph for Python
- javacg-dynamic for Java

One obvious flaw with dynamic call graph tools is that the graphs that they produce are not necessarily sound. One function invocation may not be triggered during a program run and is hence not listed in the call graph. To obtain a better coverage, the graph utility can be executed under different conditions to increase the number of subroutine calls.

Static source code analysis produces sound graphs because, as they are not bound to listing only executed calls, they include all possible invocations. A problem with this approach is that obtained call graphs are not necessarily precise and instead often include more calls that there are in practice. Another issue is that dynamic features of programming languages do not work well with static analysis because runtime conditions can change the behavior of the program unpredictably. As most modern compiled languages include dynamic features, static analysis may produce graphs less efficiently for several applications.

Static call graph generation is used in this work to ensure sound graphs. The tools presented above often work by creating a text file in an intermediate representation format that is used by another utility to create a human-readable picture. For instance, pyan creates a symbolic graph and writes it in the .dot format. graphviz can thence be executed to process the file into an image. Two files need to be obtained at this step from the intermediate format:

- An **edge** file lists all subroutine calls. Its lines are each formatted as caller $\rightarrow$ callee.
- A **logged subroutines** file lists all subroutines that trigger a log.

B. Call Graph Simplification

The call graph obtained at the previous step might be too large to be used efficiently. Given a graph with hundreds of nodes and assuming that only a few of them are associated to subroutines that produce logs, it is possible to combine unlogged nodes in a way that does not lead to information loss and makes the graph more convenient to use. Two types of nodes can be compacted:

Serial nodes are placed in series. In other words, they form a continuous sequence of nodes in a single direction. Figure 2 shows an example. The grey node represent logged subroutines while nodes labelled with letters do not produce logs.

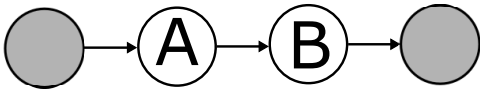

Figure 2: Serial nodes $A \rightarrow B$

When the rightmost subroutine is executed, subroutines A and B were necessarily executed in this order. The two nodes can be collapsed in a new node without losing information:

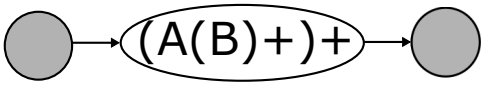

Figure 3: Collapsed serial nodes $A \rightarrow B$

The notation $(A(B)+)+$ is borrowed from regular expressions and indicates the set of possible subroutine execution order. For example, if a call to B is comprised within a loop, then the sequence (A, B, B) can be observed. By using finer static source code analysis, it is possible to use a more specific notation. For example, if the subroutines are very simple, as depicted in listing 1, the expression (AB) can be used instead to show that the two functions are always called sequentially.

Parallel nodes share the same source and target nodes as shown as an example in figure 4.

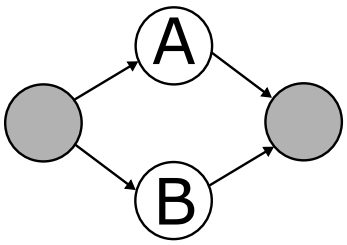

Figure 4: Parallel nodes A and B

When the rightmost node is executed and produces a log, it is not possible to determine which subroutine among A and B triggered the call. Either one of the subroutine was executed, which can be expressed using regular expressions in figure 5.

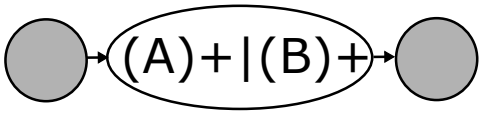

Figure 5: Parallel nodes A and B

The combination of serial and parallel nodes is applied recursively until there remains no node in the graph that can be simplified. Some nodes cannot be simplified because doing so would reduce the accuracy of the call graph. For instance, consider figure 6.

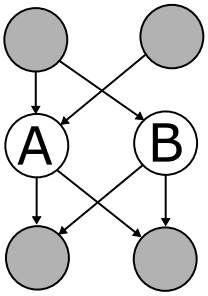

Figure 6: Call graph that cannot be simplified

In this case, subroutines A and B can be called in arbitrary ways and no regular expression can express the set of possible sequences. These nodes are hence left as is in the graph and used during path reconstruction.

The resulting call graph contains all the initial information but is optimized for execution path reconstruction. It can also be used by human developers to better understand where the logging statements are located in the source code and when they produce logs.

All graph-related operations are implemented in Python with the networkx library [7]. The call graph is created from the edge file and simplified with the log file, which are both created at the previous step.

C. Execution Path Reconstruction

Given a sequence of logs and a simplified graph obtained at the previous step, the execution path of a program can be approximately reconstructed. The procedure described in section II-D is implemented in Python. In short, the program starts at the source node (the program's entry point) and enumerates the nodes it encounters as it traverses the call graph to reach runtime-produced logs in order.

D. Reconstruction Evaluation

The precision and recall of execution path reconstruction can only be assessed during controlled tests because the complete trace of a program is not available during normal use. Given a list of logs, a simplified call graph, and the real execution path, performance indicators can be calculated as described by equation 4. As a demonstration, the simple program described in figure 1 is implemented. Subroutines are called following a uniform distribution. The program is executed twenty times and the reconstruction performance indicators are reported in table I. One notices that precision is always at 1.0. This is because precision indicates the fraction of retrieved elements that are true positives. Since the call graph is sound, the algorithm can only reconstruct (i.e. retrieve) possible paths. In the case of a simple program, the algorithm does not list erroneous paths, which leads to a high precision, but in more complex programs, the precision decreases since the exact path cannot be reconstructed. Recall evaluates the fraction of elements that are correctly retrieved. As the number of logs decreases, some subroutine calls become impossible to obtain. For instance, if logs D and F are eliminated, the nodes in the rightmost branch of the call graph cannot be detected.

Table I: Demonstrative Performance Indicators

Labelled Subroutines	Precision	Recall	F1-Score
A, B, C, D, E, F	1.0	1.0	1.0
A, B, C, D, E	1.0	0.903	0.949
A, B, C, D	1.0	0.677	0.808
A, B, C	1.0	0.548	0.708
A, B	1.0	0.323	0.488
A	1.0	0.032	0.0625
$\emptyset$	0.0	0.0	0.0

IV. Evaluation

A. Execution Path Reconstruction

Evaluation of the algorithm's ability to reconstruct execution paths is first performed. A relatively small application must be used to understand accurately the performances; therefore, distributed applications are not used at this point. The scikit-learn library [8] is a machine learning toolkit written in Python that is well maintained and documented. It also includes different verbosity levels, which is useful to measure the impact of log count on path reconstruction precision and recall. Version 1.0.1 is used.

The call graph is elaborated with pyan, which uses static source code analysis. Obtaining the real execution path for evaluation is more complicated. Two approaches can be used:

- Add a logging message at the beginning of each function. This approach works well but requires modifying the source code, which is unacceptable in many contexts and can lead to accidental regressions.

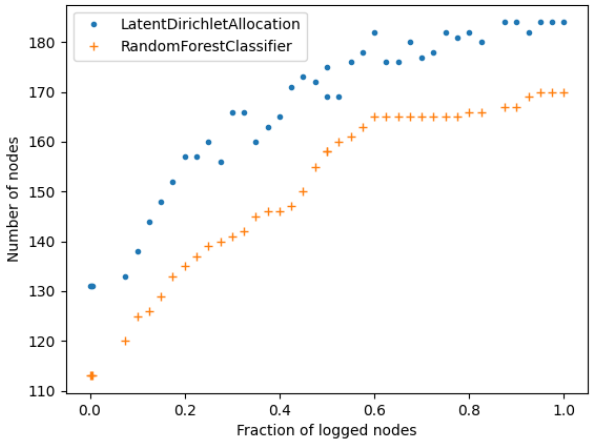

Figure 7: Number of nodes against number fraction of logged subroutines in simplified call graphs

- Use low-level features from the programming language to trigger an interruption on each function call. This leads to performance losses but does not require the modification of any code.

The second approach is applied with the sys.setprofile function from the standard Python distribution. Each call event is reported to a user-defined function that can be used to enumerate all subroutine invocations. As noted in the sys module documentation, this usage is reserved for specialized applications.

The call graph can restrict the scope of the execution path to make it more compact and useful. For example, in scientific Python libraries, numerous calls to functions in the numpy library are made. The same can be said for Java applications that make use of standard JVM functions. These calls make up the majority of subroutine calls but do not provide much insight in the behavior of a particular system because their implemented operations are not specific to one application. Thus, the call graph only includes the subroutines of a specific library – in this case, scikit-learn.

As described in section III, three files are created:

- Possible subroutine calls (edges in the call graph)
- Traces (calls performed at runtime)
- Logged subroutines (functions that produce a log)

The third file is created by sampling subroutines to simulate logs. That way, the effect of logs on path reconstruction can be calculated. When few logs are provided, the call graph can be simplified more thoroughly, as shown in figure 7. In these real applications, the call graph comprises more than a hundred edges and is difficult to read. By combining nodes that do not produce runtime messages, the graph size is reduced, which makes it easier to visualize. Also, contrarily to demonstrative programs, real applications comprise several subroutines that cannot be combined.

Execution path reconstruction is then performed from the simulated logs and the graph for the "RandomForestClassifier" module. The F1-score is reported in table II. As expected, the

Table II: Demonstrative Performance Indicators

Fraction of logged subroutines	F1-Score
1.0	1.0
0.9	0.989
0.8	0.936
0.7	0.802
0.6	0.102

performance of path reconstruction is directly linked to the number of subroutines whose execution can be tracked with a log. Removing a few log messages reduces performance slightly because the nodes between calls can be found, which results in completed paths. However, if too many logs are removed, the F1-score drops sharply because many possible paths between nodes become possible and the algorithm lacks enough information to determine which subroutines were executed. This observation if consistent with the performances obtained with the demonstrative application as presented in table I.

Scalability is a concern because finding possible routes between nodes requires $O(KN^3)$ operations for the K shortest paths [7]. Instead of finding all possible paths between nodes and then select the paths that are possible, the algorithm evaluate the 10 shortest paths and uses the rules described above to pick the likeliest execution path. This method makes the approach adapted to large graphs but decreases precision because incorrect paths can be retrieved.

RQ1 (how precisely can execution paths be obtained): The precision and recall of execution paths reconstructed from paths can be determined approximately and evaluated if the complete trace is available. Also, a call graph generated either dynamically or statically can be simplified to make the operations more efficient.

B. Enhanced Anomaly Detection

Execution path reconstruction adds information about the behavior of a system, as shown in figure 8.

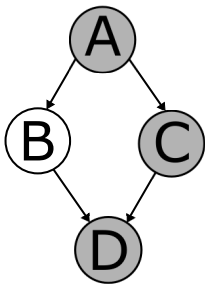

Figure 8: Example of additional knowledge provided by a graph

If A and D logs are obtained, it means that C was necessarily executed. This "additional knowledge" has been

shown to be obtainable in the last section and is applied to anomaly detection. Three issues are expected:

- Path reconstruction increases the amount of data, which makes prediction algorithms less fast.
- Paths add data that could be regarded as noise because it could fail to provide insightful information.
- If a reconstructed subroutine appears frequently, it could make a machine learning model overfit the data.
- Multiple threads cannot work with the approach because, by definition, they represent distinct execution paths. Similar thread IDs are grouped to avoid this problem.

Anomaly detection performances are tested with the HDFS dataset, which is obtained from the Hadoop application executed on Amazon servers. Since the source code needs to be available to reconstruct paths, other log datasets such as BGL cannot be used [4]. The version that was used to generate this dataset is not indicated, but it is known to have been released in 2018. The Hadoop version corresponding to this time, 3.0.1, is thus used. Contrarily to the last section, the execution path is not available and the precision and recall of the reconstruction cannot be assessed.

The program java-callgraph is used to generate the call graph statically. Since the source code of the application is very large, only relevant subsections are used to produce the graph. To identify them, the logging statements are identified manually in the source code and traced back to the runtime-produced logs. The execution path is reconstructed following the approach described above.

The supervised machine learning models described by He et al. (logistic regression, decision tree, and SVM) are applied to the HDSF dataset. Fixed one hour session windows are used. Results are reported in table III.

Table III: HDFS Anomaly Detection Performances

Model	F1 from He et al.	F1 from execution paths
LR	0.98	0.95
DT	1.00	0.97
SVM	0.98	0.96

As shown in table III, performances decrease when execution path reconstruction is used. This can be explained by the noise produced by the addition of new information

RQ2 (can execution paths improve DevOps operations): This work fails to demonstrate that execution path reconstruction can improve anomaly detection. However, the performance loss induced by execution paths is not large and the approach could potentially be modified to increase performances.

V. Future Work

Several additions can be made to this work.

Scaling concerns. Tests were performed on relatively small call graphs with less than 200 edges. More complex applications with even more subroutine calls and distributed features could make the approach unsuitable, so it needs to be modified to function in these contexts.

Apply the approach to other applications. For example, execution path reconstruction could be applied to root cause analysis to track the source of technical failures or empirical study to help developers manage software projects.

Validate path reconstruction on distributed systems. Execution path was validated with ground truth on single-process non-distributed applications. More complex and varied projects need to be used to validate and improve the system.

Use execution path to improve logging practices. Execution path reconstruction accuracy can be precisely determined, which can be used to improve logging practices. For example, serial logs (as depicted in figure 2) may not be useful because they capture information about the same branch. Path Reconstructor could be used to determine empirically how useful a logging statement is to understand the inner behavior of a software system.

Apply other machine learning methods. Only three machine learning methods were applied, but others could be attempted to see if they are adapted to execution paths. For instance, PCA, clustering, and invariants mining, which were described by He et al. [4] could be tested along with methods based on neural networks.

Execute tests on other programs and datasets. Only two applications were used for tests, which is insufficient to validate the approach thoroughly. Other applications, especially distributed systems, need to be considered to ensure that execution paths can be useful in practice.

Compare static and dynamic call graph generation. Both types of graph can be used to reconstruct paths. The static method was used because it produces sound graph, but a dynamic approach could lead to more precise and compact graphs, which could improve performance and execution time.

Integrate other metrics. The inclusion of performance indicators such as memory usage could provide additional knowledge about the systems that could be useful to human developers as well as automated monitoring methods.

Integrate log mining tools in more practical toolkits. This addition goes beyond the scope of this work but becomes more important as new log-based applications are developed. Before use, logs need to be:

- obtained from running a program,
- processed to remove noise, and
- in some contexts, traced back to their logging statements

Incompatible tools are independently developed by researchers without considering the need for these applications to work well together, which forces developers to spend time re-implementing already existing solutions. For example, in this work, tracing logs back to their logging statement in the source code was done manually because efficient applications for this use are not publicly available. Another concern is that language-agnostic tools are not sufficiently emphasized, which leads to solutions developed specifically for a single type of applications (usually Java-based distributed applications). Path Reconstructor is implemented in Python but uses as an input text files of a simple format, which makes it usable for any mainstream programming language.

VI. Related Work

Static source code analysis has been applied to several software monitoring methods.

Chen et al. have developed a tool called Pathidea that improves retrieval-based bug localization with log-inferred execution paths. Most bug localization tools only rely on information available in bug reports to generate a list of files that need to be inspected to identify a bug. Pathidea relies on static source code analysis and bug reports to determine the operations executed before the occurrence of a bug. Their tool provides additional insight that helps developers fix bugs more effectively. The team has assessed that their system outperforms other available methods [3].

Schipper et al. demonstrated that log processing can be improved by tracing logs produced at runtime with the statements in the source code that produced them. Their approach aims at helping developers understand the relation between runtime logs and source code by showing the exact logging statements that produce specific messages. Their tool works differently from other ones. Instead of trying to assign a log to its associated statement, it starts from source code statement and uses them as templates to match runtime message. This approach has been shown to be ready for real-life cases [9].

Ding et al. proposed a method to ease software failure diagnosis by reconstructing execution paths of a program before its halt. They identified three issues with this approach (unavailable user input, difficulty in reproducing the execution environment, and non-deterministic behaviors on multi-process systems) that make this task difficult to automate. Finding bugs often requires time-consuming manual inspections by developers, but their tool called SherLog automates this process and was successfully tested on real software systems [10].

VII. Conclusion

This paper presents an execution path reconstruction approach called Path Reconstructor that uses runtime logs to approximately determine the inner behavior of a software system. Two use cases are presented. First, the effectiveness of the approach to reconstruct execution paths is evaluated on a single-thread non distributed application. Second, reconstructed paths are used to detect anomalies in a distributed system. The accuracy of path reconstruction can be obtained precisely. However, anomalies were not detected more effectively than with simpler approaches. Future works should consider applying the method on other software systems and addressing scalability concerns.

References

[1] X. Zhang, C. Xie, R. Yao, Y. Xu, Q. Lin, and B. Qiao, "Robust log-based anomaly detection on unstable log data," *ESEC/FSE*, August 2019.

[2] R. Wang, S. Ying, C. Sun, H. Wan, H. Zhang, and X. Jia, "Model construction and data management of running log in supporting saas software performance analysis," 2017.

[3] A. R. Chen, T.-H. P. Chen, and S. Wang, "Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[4] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detectionshilin he, jieming," *IEEE 27th International Symposium on Software Reliability Engineering*, 2016.

[5] L. Sui, M. Dietrich, Jens amd Emery, S. Rasheed, and A. Tahir, "On the soundness of call graph construction in the presence of dynamic language features - a benchmark and tool evaluation," *Programming Languages and Systems (APLAS)*, 2018.

[6] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," *Proc. OOPSLA'00. ACM*, 2000.

[7] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[9] D. Schipper, M. Aniche, and A. van Deursen, "Tracing back log data to its log statement: From research to practice," *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019.

[10] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," *ASPLOS'10*, 2010.