Comparing Execution Trace Using Merkle-Tree to Detect Backward Incompatibilities

Atsuhito Yamaoka*, Teyon Son*, Kazumasa Shimari*, Takashi Ishio†, Kenichi Matsumoto*,
*Graduate School of Science and Technology, Nara Institute of Science and Technology, Nara, Japan
Email: {yamaoka.atsuhito.xv2, son.teyon.sr7@is.naist.jp, k.shimari, matumoto}@is.naist.jp
†School of Systems Information Science, Future University Hakodate, Hokkaido, Japan
Email: ishio@fun.ac.jp

***Abstract*—The use of libraries is crucial in software development. Library users should update their libraries to address bugs and vulnerabilities that are fixed in newer versions. However, updating libraries can lead to software malfunction due to backward incompatibilities. Therefore, it is necessary to carefully examine the changes in the library, identify incompatible behavior, and modify the software accordingly when applying updates. Identifying the cause of incompatibility is challenging as updates often include changes to APIs other than the one used by the user. We propose a method to detect candidate library methods that cause backward incompatibilities in client-side library updates using Merkle tree. Our approach involves conducting unit tests on the client software, which includes library API calls, before and after the library updates. The execution traces of these tests are collected at the Java bytecode instruction level. By constructing Merkle trees for each execution trace before and after the update, we efficiently compare the control structures and return values to identify the differences indicating backward incompatibilities. To validate the effectiveness of our method, we conducted a case study on three instances of incompatibility in open-source software.**

***Index Terms*—Library Update, Compatibility, Dynamic Analysis**

I. Introduction

Libraries are used to enhance the productivity and quality of software [4]. They are constantly evolving, adding new functionality and fixing any vulnerabilities. As a result, software developers need to regularly update the libraries they use [10].

Libraries are expected to maintain backward compatibility, meaning that their behavior should not change after applying a version update. Semantic Versioning enables developers to check whether libraries keep the compatibility. Semantic Versioning is indicated by three numbers separated by "." in MAJOR.MINOR.PATCH. Basically, MINOR and PATCH updates in libraries keep backward compatibility. However, it has been reported that backward compatibility is not always maintained for updates other than the Major version [8], [12]. Steve et al. revealed that breaking changes have occurred in approximately one-third of Java library releases [11]. Mostafa et al. also reported that only 82 out of 296 documented incompatibilities were explicitly documented [8]. As a result, when using the library, it is possible to encounter incompatibilities on the client side, and finding a solution may be unclear.

Client-side developers should verify the compatibility of libraries if the information provided by the library developer regarding incompatibilities is insufficient. There are static and dynamic methods for verifying compatibility. Foo et al. proposed a method to check library compatibility based on the static analysis by computing the differences between source-level elements of the library. Their method could suggest upgrades for 10% of the libraries [1]. However, there have been some runtime errors in an earlier part of the CI/CD pipeline in suggested updates, so it is necessary to consider adapting dynamic analysis. Jezek et al. evaluated static analysis tools to detect incompatibilities and found that all investigated tools have failed to detect certain API incompatibilities [3].

Some research has tackled to verify compatibility in dynamic methods involving test execution. A method for identifying type incompatibilities in JavaScript libraries [6] There are some methods using other OSS runtime information. Møller et al. and Mujahid et al. proposed methods for detecting incompatibilities using dynamic information from OSS that depend on the same library [7], [9]. A method is also proposed to improve test coverage by using both static and dynamic call graphs to obtain control dependencies in the source code of the library and to perform compatibility verification based on the differences in the source code when updates are made [2]. Furthermore, some methods generate tests to detect incompatibilities for dependent libraries based on other project execution information registered in a database [15].

These methods can verify the existence of backward compatibility, but they do not directly present information to the developer on which methods in the library have differences. When developers encounter incompatibilities, they need to understand how the internal behavior has changed and how this change has affected execution.

In this study, we propose a method to detect candidates for the library methods that caused the incompatibility. To detect incompatibilities, we record execution traces in the execution of unit tests on the client software and compare the behavior of the libraries before and after updating them. We create and compare hash values using the Merkle tree structure for each method from these instruction sequences related to the control path and return value. To confirm the usefulness of the proposed method, we applied our method to three cases of OSS incompatibility. The results showed that the proposed method was able to detect incompatible library methods with reasonable performance.

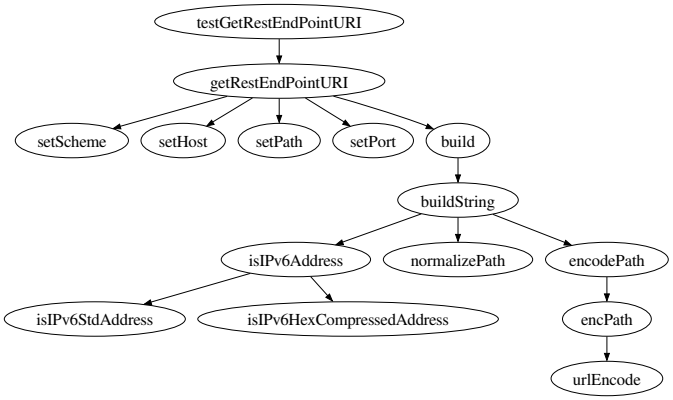

Fig. 1. API Call Graph in the unit test for the incompatible function in Wasabi

In the remainder of the paper, Section II shows a motivating example of our research. Section III describes the proposed method. Section IV shows some examples where our method is applied. Section V discusses threats to validity. Finally, in Section VI, we summarize this study and discuss future works.

II. Motivating Example

This section covers a motivating example in Compsuite, a dataset that records cases of incompatibility when updating libraries in the OSS. This is an example where the client software *wasabi*[1] calls the library *httpcomponents-client*[2] in a test case. The incompatibility occurs when the *httpcomponents-client* version is updated from 4.5.1 to 4.5.10. In this version update, it is challenging to check the source code changes due to modifications in 217 class files within the library.

Developers can track changes by following the library API calls on the client software. Figure 1 shows the call tree in this case. The root node is the test method on the client software. It calls and tests the client method. The client method calls the five methods in the library. Both before and after the update, there are more than ten library methods that are executed, excluding any duplicate methods. Three methods have differences in these source code, but it is difficult to analyze whether the changes in each method affect the execution and its return value. Furthermore, it is also difficult to use the debugging method such as directly inserting print statements into the library source code because the library version is often managed by a build automation tool such as Maven. Therefore, to detect the causes of incompatibilities, it is necessary to monitor the method changes in behavior and its return value.

III. Proposed Method

We propose a method for detecting candidate methods that may cause incompatibility in the library. By capturing alterations in the execution sequence of a method call, it becomes possible to identify which method's behavior has been modified due to a library update.

[1]https://github.com/intuit/wasabi

[2]https://github.com/apache/httpcomponents-client

At first, execution traces are collected and summarized using hash values for each method invocation. Next, we construct Merkle trees [5] and calculate hash values respectively. Finally, we compare the two Merkle trees and detect the candidates of the method with backward incompatibilities from different hash values in the tree. The following sections describe the proposed method in four steps.

1) Collect Execution Traces
2) Merkle-Tree Construction
3) Calculate Hash Values
4) Detect Backward Incompatibilities

In this study, our target language is Java. However, the hash value calculation and Merkle tree-based difference output methods can be applied to execution traces from other programming languages. Our implementation for Merkle tree construction is available on GitHub[3].

A. Collect Execution Traces

For both before and after OSS library updates, we execute the program utilizing OSS unit tests and collect an execution trace of each run. We leverage the execution trace collection tool SELogger [13], which is an implementation of Omniscient Debugging. This technique records all data values associated with each instruction in Java bytecode, forming a comprehensive execution trace. Specifically, it captures actual arguments, return values, and exceptions at the method entry and exit, as well as target objects, subscripts, and values read/written during field and array read/write instructions in chronological order. To distinguish references for objects, ID values are stored. SELogger can run as a Java agent within a Java Virtual Machine. It utilizes ASM [4], a framework for manipulating Java bytecode, to inject logging instructions into the loaded class of the program under analysis and collect an execution trace.

B. Merkle Tree Construction

We construct a Merkle tree from the execution traces collected both before and after the library update. The structure of the Merkle tree consists of hash values at each node. The hash values of non-leaf nodes are calculated based on the hash values of their child nodes. By comparing the hash values of the roots of two different Merkle trees, the entire tree can be efficiently compared.

In our method, method calls are treated as nodes, and we construct a Merkle tree with the calling relation represented by a directed edge. For example, when calls from *method A* to *method B*, *method C* are recorded in this order in the execution trace, the Merkle tree constructed from this execution trace has three nodes (*method A*, *method B*, *method C*) and two directed edges (*method A to method B* and *method A to method C*.) Each method call is regarded as a node, so it does not have a closed path and is a non-cyclic graph. Each node has hash values calculated from the execution trace which is recorded from the method start to end and the hash value of the method call within the method.

[3]https://github.com/yebityon/POLDAM

[4]https://asm.ow2.io/

C. Calculation of Hash Values

A hash value is calculated from the execution trace for each method call from the method start to end. The hash value is a summary of the execution trace, and method calls for different hash values are considered to have differed during execution. In our implementation, when a method call instruction is recorded, a hash value is calculated using the execution trace up to the corresponding end-of-execution instruction for that method.

In this study, we define four hash values for the execution trace. For each method call, we calculate a hash value for bytecode instructions collected from the start to the end of method execution (Flow Hash), a hash value for the method return value recorded in the execution trace (Param Hash). For each of them, we calculate hash values (Control Flow Hash and Control Param Hash) for Merkle tree comparison. The Control Flow Hash and Control Param Hash for a given method call is calculated using the Param Hash and Flow Hash on the method and the Control Flow Hash and Control Param Hash of the methods which is called within the target method.

Flow Hash

Flow Hash is calculated from the Java bytecode event names recorded from the start to the end of execution in the target method. The events executed in another method within the target method are not included in the calculation of the hash value.

Param Hash

Param Hash is calculated from the Java bytecode events which is related to the return value of the method. Specifically, it is a hash value calculated using only the value of the $METHOD\_EXIT$ instruction in SELogger when it is recorded.

Control Flow Hash

Control Flow Hash is calculated using the Flow Hash of the target method and the Control Flow Hash of the methods which is called within the target method. A method call with equal Control Flow Hash means that the execution traces of the methods invoked within the method are exactly the same.

Control Param Hash

Control Param hash is calculated using the Param Hash of the target method and the Control Param hash of the methods which is called within the target method. A method call with equal Control Param Hash means that the method calls return the same value.

D. Detect Backward Incompatibilities

We detect the candidate methods of the backward incompatibilities by comparing the hash value in the Merkle tree before and after the library update. We can find the method calls that record different hash values by tracing each hash value manually.

Due to the nature of Merkle trees, when different hash values are recorded in a method call, there is a runtime difference in either the method call or the methods called within the method. We examined whether the Flow Hash, Param Hash, Control Flow Hash, and Control Param Hash changed for each method commonly invoked by the updated library methods before and after the library update. During this phase, the Control Flow Hash or Control Param Hash is verified. If both hash values are identical, the instructions and return value within the method, or the methods transitively invoked by the method, are the same. Then, we can stop tracing the method call in the child nodes, leading to a reduction in the number of methods that need to be checked.

If the Flow Hash or Param Hash is changed, we further check whether a code change has occurred in the method. Then, we can identify the candidate methods whose execution has differed due to a code change.

IV. Case Study

We apply our proposed method to the cases of library updates to verify whether it is possible to detect the cause of backward incompatibilities. From the Compsuite dataset [14], we selected three incompatibilities (i-2, i-7, and i-49). These incompatibilities were chosen because we were able to reproduce them and identify their cause.

The dataset contains the cases where incompatibilities occurred during a library update in Java OSS. Table I shows the details of the incompatibilities. Each case comprises the name of the client software, the name of a library, the incompatibility-induced test case, the revision of the client whose unit test failed, and the library version information before and after the update. These data can be used to reproduce incompatibilities caused by library updates. As the dataset does not include the source code that causes incompatibility, we manually identified the cause of the incompatibilities.

We apply our method to the three incompatibility cases and evaluate the number of candidate library methods that may cause backward incompatibilities in client-side library updates using Merkle tree. The number of candidate library methods approximates the manual effort for source code reading to investigate the actual cause in code. In addition, we measure the time and storage costs associated with this process. To measure execution times, we collect execution traces and build the Merkle tree 10 times, calculating the average. We collected execution traces within a Docker container whose image is Ubuntu 22.04 with 16 GB memory and CPU 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80 GHz 2.80 GHz. Then, we constructed the Merkle-Tree on the CentOS Linux, which has Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60 GHz and 264 GB memory to avoid Out-of-Memory error.

A. Result

Table II shows the number of methods detected by the proposed method. It shows the number of value changes in Flow Hash (FH), Param Hash (PH), Control Flow Hash (CFH), and Control Param Hash (CPH). M_{com} means the number of methods in the entire library that are executed both before and after the update. M_{diff} is the number of methods whose code changes before and after the update. M_{FHorPH} and $M_{FHandPH}$ means the number of Merkle tree methods with code change whose Flow Hash or/and Param Hash changes.

TABLE I
DETAILS OF INCOMPATIBILITIES IN THE DATASET

ID	client	library	test	client sha	version$_{before}$	version$_{after}$
i-2	square/retrofit	square/moshi	MoshiConverterFactoryTest#failOnUnknown	b8d593c	1.8.0	1.9.2
i-7	apache/druid	qos-ch/slf4j	Slf4jLogFilterTest#test_slf4j	1d784d1	1.7.9	2.0.0-alpha1
i-49	intuit/wasabi	apache/httpcomponents-client	DefaultRestEndPointTest#testGetRestEndPointURI	9f2aa5f	4.5.1	4.5.10

At first, we confirmed that the Flow Hash and Param Hash have changed for all methods that are actually the cause of the library incompatibility for each ID.

The incompatible methods differ in the instructions executed internally and result in different return values. Therefore, to focus on them, we focus on changes in the Flow Hash or Param Hash. Using the proposed method, we have succeeded in narrowing down the candidate methods that caused the incompatibility (M_{FHorPH}) with about 80 to 90% less effort than simply tracing methods in the dynamic call graph (M_{com}). Furthermore, in all three cases, focusing on both Flow Hash and Param Hash, which indicate the execution path and return value, led to the identification of candidate methods that caused the incompatibility ($M_{FHandPH}$) compared with simply focusing on code difference (M_{diff}).

For example of i-49, 12 methods are commonly executed before and after the library is updated. Figure 2 shows a part of the Merkle tree of version 4.5.1, and Figure 3 shows a part of the Merkle tree of version 4.5.10. In this update, the source code in the *buildString* and *encodePath* methods have been changed in these displayed methods.

In Figure 3, we display methods where either the Control Flow Hash or the Control Param Hash has changed in red. To detect incompatibilities, we focus on the Control Flow Hash and Control Param Hash, which also have different parent hash values, and search for nodes where the Flow Hash and Param Hash are different until the hash values of the children are different. Then, we can find that by checking the Flow Hash and Param Hash in the child elements of the *buildString* method, the final return value is affected by the return value change due to the *normalizePath* method no longer being called, which was called before the update. Thus, we can see that the code is detected as a hash value where the code has been changed. The changes made in the update are shown in Listing 1.

Listing 1. Difference in *buildString* method

```
1  private String buildString() {
2  ...
3  - sb.append(normalizePath(this.encodedPath)
       );
4  + sb.append(normalizePath(this.encodedPath,
       sb.length() == 0));
5  - } else if (this.path != null) {
6  + } else if (this.pathSegments != null) {
7  - sb.append(encodePath(normalizePath(this.
       path)));
8  + sb.append(encodePath(this.pathSegments));
```

Therefore, our proposed method successfully detects the method that caused the library incompatibility from the library calls in the execution of the test case in the i-49 case.

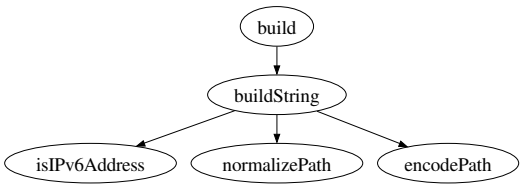

Fig. 2. Subtree of Merkle tree in version 4.5.1

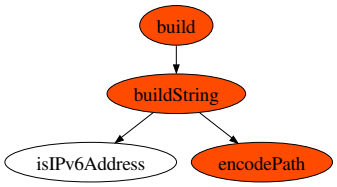

Fig. 3. Subtree of Merkle tree in version 4.5.10

B. Performance

The performance is described in Table III. The amount of execution trace is at most a few MB for the three cases. The execution time is shorter when the execution trace is obtained for the i-7 data than when the unit test is performed alone, but this is thought to be due to the large variation in time for each execution. The trace collection overhead is at most about 1 minute. This is because we need to collect traces only for a single failed test case due to incompatibilities. The construction of the Merkle tree is shorter than the overhead of trace collection. Collecting execution traces before the update and constructing Merkle tree can be automatically done by integrating into an environment such as CI. The proposed method will be automatically executed when the library is updated and the test fails and can report the candidate of the method of the cause of the incompatibilities to the developer. In this case, the cost for checking incompatibilities when updating the library is the collection of execution traces after the update and the construction of the Merkle tree. The total overhead for both processes is at most two minutes, so we can consider this to be a reasonable execution time for incompatibility analysis. The trace size is at most a few MB per test case, so it can be said that the storage cost is not that high for analysis of the incompatibility. Therefore, it can be said that the proposed method can be implemented at a reasonable cost.

TABLE II
NUMBER OF METHODS DETECTED BY PROPOSED METHOD

ID	M_{com}	FH	PH	CFH	CPH	M_{diff}	M_{FHorPH}	$M_{FHandPH}$
i-2	81	17	24	21	32	9	9	8
i-7	11	9	4	9	8	4	4	1
i-49	12	3	3	4	4	3	3	2

TABLE III
PERFORMANCE OF THE PROPOSED METHOD

DataID	#Event		Trace Size[MB]		$Time_{normal}$[ms]		$Time_{trace}$[ms]		$Time_{merkle}$[ms]	
Update	before	after	before	after	before	after	before	after	before	after
i-2	98,988	257,451	1.5	3.9	267,433	257,761	320,081 (+20%)	287,425 (+12%)	4,207	4,360
i-7	51,867	99,682	0.79	1.5	576,884	558,925	575,119 (±0%)	567,702 (+2%)	5,831	5,510
i-49	31,876	151,911	0.49	2.3	20,322	21,832	29,053 (+43%)	31,412 (+44%)	13,868	13,837

V. THREATS TO VALIDITY

In this study, our method only focuses on libraries that have updated the version they directly depend on in pom.xml. Therefore, if the transitively dependent library version is updated, our proposed method cannot be used as is.

The Param Hash concentrates on the execution trace of the return value and may not precisely reflect the state of the object. If the return value is unchanged but the internal state of the object has changed, the Param Hash will have the same value before and after the update.

The Param Hash does not take into account instructions such as random numbers, which have different values for each execution. Therefore, even in methods that are not impacted by library updates, if the return value is affected by an instruction whose value is different with each execution, the Param Hash may change, potentially resulting in false positives.

In all incompatibilities in the case study, the structure of the Merkle tree has not changed significantly in the library update. If the update includes a large number of changes in the execution, it is difficult to apply the proposed method.

VI. CONCLUSION AND FUTURE WORK

We propose a method to detect the candidates of the backward incompatible methods when the library is updated. The structure of Merkle trees made it possible to effectively compare and identify the library methods whose execution traces changed in the update. In the case study, we confirmed that the proposed method can be used to check incompatibilities in three incompatibilities.

Our future work includes applying the proposed method to a broader range of datasets to demonstrate its effectiveness in detecting incompatible methods. We also would like to improve our method using automated test generation and considering randomness and flaky tests. Additionally, we would like to develop a tool to automate the entire process of detecting incompatibilities in the proposed method and assess the effectiveness of our method.

ACKNOWLEDGEMENT

This research has been supported by JSPS KAKENHI Nos. JP20H05706, JP22K21279 and JP23K16862.

REFERENCES

[1] D. Foo, H. Chua, J. Yeo, M. Y. Ang, and A. Sharma, "Efficient static checking of library updates," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, p. 791–796.

[2] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? A case study of Java Projects," *Journal of Systems and Software*, vol. 183, p. 111097, 2022.

[3] K. Jezek and J. Dietrich, "Api evolution and compatibility: A data corpus and tool evaluation," *Journal of Object Technology*, vol. 16, no. 4, pp. 2:1–23, Aug. 2017.

[4] D. Konstantopoulos, J. Marien, M. Pinkerton, and E. Braude, "Best principles in the design of shared software," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, vol. 2, 2009, pp. 287–292.

[5] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378.

[6] G. Mezzetti, A. Møller, and M. T. Torp, "Type Regression Testing to Detect Breaking Changes in Node.js Libraries," in *32nd European Conference on Object-Oriented Programming*, vol. 109, Dagstuhl, Germany, 2018, pp. 7:1–7:24.

[7] A. Møller, B. B. Nielsen, and M. T. Torp, "Detecting locations in javascript programs affected by breaking library changes," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020.

[8] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: A study on behavioral backward incompatibilities of java software libraries," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 215—225.

[9] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 466–476.

[10] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo, "Out of sight, out of mind? how vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, no. 4, p. 59, Apr 2021.

[11] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.

[12] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 215–224.

[13] K. Shimari, T. Ishio, T. Kanda, N. Ishida, and K. Inoue, "Nod4j: Near-omniscient debugging tool for java using size-limited execution trace," *Science of Computer Programming*, vol. 206, p. 102630, 2021.

[14] X. Xu, C. Zhu, and Y. Li, "Compsuite: A dataset of java library upgrade incompatibility issues," https://arxiv.org/abs/2305.08671, 2023.

[15] C. Zhu, M. Zhang, X. Wu, X. Xu, and Y. Li, "Client-specific upgrade compatibility checking via knowledge-guided discovery," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, may 2023.