Client-Specific Upgrade Compatibility Checking via Knowledge-Guided Discovery

CHENGUANG ZHU*, The University of Texas at Austin, USA
MENGSHI ZHANG, Meta Platforms, USA
XIUHENG WU, Nanyang Technological University, Singapore
XIUFENG XU, Nanyang Technological University, Singapore
YI LI†, Nanyang Technological University, Singapore

Modern software systems are complex and they heavily rely on external libraries developed by different teams and organizations. Such systems suffer from higher instability due to incompatibility issues caused by library upgrades. In this paper, we address the problem by investigating the impact of a library upgrade on the behaviors of its clients. We developed CompCheck, an automated upgrade compatibility checking framework which generates incompatibility-revealing tests based on previous examples. CompCheck first establishes an offline knowledge base of incompatibility issues by mining from open source projects and their upgrades. It then discovers incompatibilities for a specific client project, by searching for similar library usages in the knowledge base and generating tests to reveal the problems. We evaluated CompCheck on 202 call sites of 37 open-source projects and the results show that CompCheck successfully revealed incompatibility issues on 76 call sites, 72.7% and 94.9% more than two existing techniques, confirming CompCheck's applicability and effectiveness.

CCS Concepts: • **Software and its engineering → Software evolution**; **Maintaining software**; *Software libraries and repositories.*

Additional Key Words and Phrases: Software upgrade, compatibility, test generation

1 INTRODUCTION

Modern complex software systems are often built by integrating components created by different teams or even different organizations, and these components inevitably evolve over time. With little understanding of changes happening in external components, it is very challenging to ensure that the new upgrades do not introduce incompatibility issues to the existing software system. This is due to the fact that the specifications of third-party components are usually limited to the descriptions of their Application Programming Interfaces (APIs) only. Yet, even upgrades which do not alter the APIs can still cause behavioral differences, hindering the stability of dependent components [77]. A recent study [41] shows that 202 out of 408 sampled open source Java projects break after an upgrade of library components, and 41.3% of the breakages were runtime test failures (not caught by compilers). Such incompatibility issues are difficult to detect statically, and there is no easy way to understand their cascading effects on the whole system.

Managing upgrades to third-party software packages is challenging and many developers choose to avoid it as much as they can. Another empirical study [50] indicates that 81.5% of the studied systems keep their outdated dependencies. When it comes to security patches, 69% of the interviewed

*This work was done in part while the first author was a research assistant at Nanyang Technological University.
†Yi Li is the corresponding author.

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-040).
Authors' addresses: Chenguang Zhu, The University of Texas at Austin, Austin, Texas, USA, cgzhu@utexas.edu; Mengshi Zhang, Meta Platforms, Menlo Park, California, USA, mengshizhang@meta.com; Xiuheng Wu, Nanyang Technological University, Singapore, Singapore, XIUHENG001@e.ntu.edu.sg; Xiufeng Xu, Nanyang Technological University, Singapore, Singapore, XIUFENG001@e.ntu.edu.sg; Yi Li, Nanyang Technological University, Singapore, Singapore, yi_li@ntu.edu.sg.

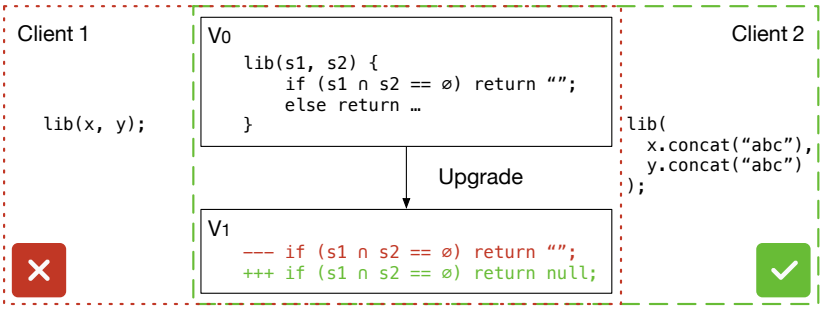


Fig. 1. A client-specific compatibility issue.

developers were unaware of the vulnerabilities in the libraries that they use and therefore were reluctant to upgrade due to extra efforts and added responsibilities. This raises critical security concerns and may in turn leave their systems open to zero-day attacks [13].

To upgrade, or not to upgrade: that is the question. An *upgrade incompatibility* issue manifests mismatches between two independently managed software components, i.e., the *client* and the *library*. Naturally, the compatibility checking problem has also been studied along these dimensions. Existing compatibility checking techniques can be classified based on (1) whether they target client developers or not (i.e., *client-oriented*), and (2) whether they treat different client applications separately when considering the impacts from a library upgrade (i.e., *client-specific*).

Client-oriented techniques are designed from the client's perspective, to provide client developers information and actionable recommendations on detected incompatibilities. Examples include smart alerting [83], which raise an alarm if a library upgrade may affect the client's original functionalities, and framework/API migration techniques such as AURA [88] which generates change rules mapping target API methods (old version) to replacement methods (new version). On the other hand, library-oriented techniques help library developers to better test and improve their library code. A recent work advancing towards this direction is DeBBI [15].

Not all client-oriented techniques are client-specific. For example, package compatibility checking tools rely on historical upgrade failure information to make recommendations to developers, including Dependabot [19] and GemChecker [17]. The downside of these techniques is that the information used in defining compatibility is too generic and does not consider how upgraded libraries are used by specific client applications, thus being imprecise when making recommendations. Regression testing [90], being client-specific, can be used to verify changes to client code, but often fails to capture issues caused by library upgrades. This is because regression tests are created by client developers, who have incomplete knowledge about library changes.

Client-Specific Compatibility Checking. To break the information barrier between client and library developers, and to precisely evaluate the impact of a library upgrade on specific client applications, we develop a *client-specific* compatibility checking framework—CompCheck. Different from existing techniques, CompCheck is *client-oriented* and *client-specific*.

Figure 1 shows an example of a client-specific compatibility issue. The upgrading of an API method *lib()* has different impacts on its two clients due to their different usage contexts, e.g., passing different arguments. The function *lib()* computes the longest common substring of two input parameters, *s1* and *s2*. In the upgraded version (*V1*), *lib()* returns *null* instead of "" (in *V0*) when there is no common string. In this example, a behavioral incompatibility is exposed in *Client1*

(*NullPointerException* may be thrown), while *Client2* remains unaffected. A client-agnostic technique would miss the subtlety and raise a false alarm for *Client2*.

To detect client-specific incompatibility issues, CompCheck first establishes an offline *knowledge base* of previously discovered incompatibility issues. The knowledge base records the *client-library pairs* before and after the upgrades, the *context summaries* indicating how the libraries are used by the corresponding clients, and the runtime *program traces* from both the libraries and clients. During the online incompatibility detection, CompCheck matches a given client with an entry from the knowledge base, if they both depend on the same library upgrades and the contexts of which are similar. We then use the recorded information from the knowledge base to guide the generation of *incompatibility-revealing tests* for the new client. Upon completion of test runs, CompCheck either discovers incompatibility issues manifested by concrete tests, or produces an enhanced regression test suite with the ability to detect potential incompatibilities in future upgrades. Client developers can directly act upon the revealed issues when performing library upgrades.

We evaluated the effectiveness of CompCheck on 202 call sites of 37 open-source projects, performing an end-to-end comparison with two existing techniques: (1) Sensor [84], a state-of-the-art technique that detects dependency (library) conflicts using test generation guided by client code analysis, and (2) CIA+SBST, a technique that detects library incompatibility by combining change impact analysis with search-based test generation. These are the most relevant competitor of CompCheck, as other existing techniques target fundamentally different problems. The experimental results show that CompCheck successfully revealed incompatibility issues on 76 call sites, 72.7% more than Sensor and 94.9% more than CIA+SBST, confirming CompCheck's applicability and efficiency. Furthermore, by comparing CompCheck's incompatibility discovery with plain search-based test generation, we show that CompCheck's object reusing significantly improves the effectiveness of incompatibility revealing.

Contributions. CompCheck is designed to handle real-world client and library programs and it produces no false positive in revealing incompatibility, since every incompatibility reported is accompanied with a concrete test case. We have implemented a prototype targeting Java projects using Maven [4] as their build system. To facilitate reusing and reproducing our research, we make CompCheck publicly available, together with additional experimental data and results.[1] To summarize, we make the following contributions in this paper.

- A dataset containing 758 incompatible client-library pairs. To the best of our knowledge, it is the largest dataset of library upgrade incompatibility issues.
- An empirical study of library upgrade incompatibility issues in open source projects, summarizing common incompatibility manifestation patterns.
- A novel technique generating incompatibility-revealing tests for new clients, guided by the summaries of previously discovered incompatibility issues from an offline knowledge base. A key advantage of CompCheck is its client-specificity: it focuses on target clients similar to the ones that are known to fail previously.

We envision CompCheck to be deployed as a long-running and self-evolving online service, ideally with tight integration with project hosting platforms, such as GitHub [34]. This will enable it to expand the knowledge base continuously by learning from new clients and tests in a scalable manner.

2 EXAMPLE

In this section, we illustrate the workflow of CompCheck on real-world project examples. Code snippets in the example are simplified for demonstration purposes.

[1] https://sites.google.com/view/compcheck

```
1 public void testKryo() {
2   Kryo kryo = new Kryo();
3   ByteArrayOutputStream bytes = new
          ByteArrayOutputStream();
4   Output output = new Output(bytes);
5   Boo b = new Boo("hello");
6   kryo.writeObject(output, b);
7   output.close();
8   ...
9 }
```

(a) Affected method M1 (Rtree).

```
1 public void testSerializeAndDeserializeOfPerson
2       StreamUsingKryo() {
3   Person source = new Person("fred", 24);
4   Serialized.kryo().write(source, file);
5   ...
6 }
7
8 private final Kryo kryo = new Kryo();
9 ...
10 public Observable write(Object source, File file) {
11   Output output = new Output(new
          FileOutputStream(file, false), 4096);
12   kryo.writeObject(output, source);
13   output.close();
14   ...
15 }
```

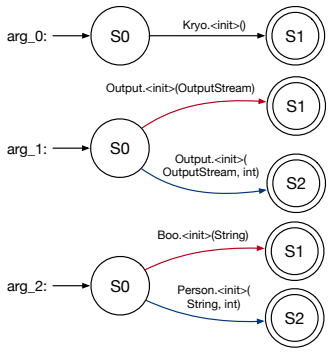
(b) Affected method M2 (Rxjava-extras).

M1:

kryo : *Kryo.<init>()* → ⊙
output : *Output.<init>(OutputStream)* → ⊙
b : *Boo.<init>(String)* → ⊙

M2:

kryo : *Kryo.<init>()* → ⊙
output : *Output.<init>(OutputStream, int)* → ⊙
object : *Person.<init>(String, int)* → ⊙



(c) Argument traces and their summary.

```
1 public byte[] serialize(final Object obj) {
2   ByteArrayOutputStream outputStream = new
          ByteArrayOutputStream();
3   Output output = new Output(outputStream);
4   Kryo kryo = new Kryo();
5   kryo.writeObject(output, obj);
6   bytes = output.toBytes();
7   output.flush();
8   return bytes; }
```

(d) Matched client method in [23].

```
1 public void save(OutputStream os,
          SaveFileFormat save) {
2   ...
3   kryo = new Kryo(resolver);
4   kryo.setRegistrationRequired(false);
5   Output output = new Output(os);
6   kryo.writeObject(output,
7                    save.componentIdentifiers); }
```

(e) Unmatched client method in [6].

```
1 @Test
2 public void test0() {
3   Boo boo0 = loadObj(arg_2_S1);
4   KryoSerializer kryoSerializer0 = new
          KryoSerializer();
5   kryoSerializer0.serialize(boo0); }
```

(f) A test generated for client (d), *loadObj* is a special function for loading a previously cached object.

Fig. 2. Clients affected by the Kryo library upgrade, their argument trace summary, and the test generated by CompCheck.

Kryo [24] is a popular binary object graph serialization framework for Java, which has over 5.7K stars on GitHub. Its artifact ID on Maven Central [55] is *com.esotericsoftware:kryo*. Two client projects, Rtree [76] and Rxjava-extras [78], depend on it and work well with the previous stable release v3.0.3. Yet, if the Kryo version is bumped to v5.0.0-RC4, they both suffer from regression test failures. The code snippets of the affected tests are shown in Figures 2a and 2b, respectively, where the API calls causing incompatibility are highlighted (Line 6 and Line 12, respectively). The API method affected by the upgrade is *Kryo.writeObject(Output,Object)*, which writes the serialized object (the *Object* argument) into an output stream (the *Output* argument). It throws an exception internally once upgraded to the newer version. This is caused by a change of the initial value of a Boolean field, which controls whether a class is required to be registered with the library before serialization can be performed. It was initialized to *false* in v3.0.3 [21] while it is then set to *true* by default in v5.0.0-RC4 [22]. Kryo's official documentation also confirms that many changes made between its major releases are backward incompatible [49].

Incompatibility Knowledge Mining. An upgrade to the Kryo library may not always cause a problem. It also depends on how the library is used by the client program. For example, client programs which always register before calling *writeObject* would not be affected by this upgrade. Therefore, compatibility checking methods should be *client-specific* [61] and the key step of our approach is to collect and summarize the usage contexts of the target library in the clients. Specifically, CompCheck collects runtime traces for each argument passed into the library API.

Figure 2c shows the *argument traces* we collected for the backward incompatible API in the affected clients M1 and M2. There are in total three arguments for *writeObject*, including the implicit one, i.e., the class instance *kryo* to which the API method belongs. For each argument, the traces record all operations on the argument with potential side-effects, preceding the API call. This includes method calls on arguments of object types. For example, the first argument (*arg_0*) of M1 is the implicit *kryo* object. Before it is passed to the API (shown as ⊙ in Fig. 2c), the method *Kryo.<init>()* is called on it, which is a no-argument constructor of class *Kryo*. Traces of other arguments in M1 and M2 are also listed in Fig. 2c.

We summarize argument traces of all previously observed backward incompatible API calls to form a knowledge base, which can be used to discover similar issues in new projects. The summary of the argument traces for M1 and M2 is shown in Fig. 2c, which is obtained by merging traces of each argument into a Finite State Machine (FSM). This way, we obtain an FSM for each argument. For instance, the FSM for *arg_1* has two branching transitions coming out of the initial state *S*0: "*S*0 → *S*1" from M1 (red) and "*S*0 → *S*2" from M2 (blue).

Knowledge-Guided Incompatibility Discovery. Given a new client program, CompCheck searches for any usage of library API calls matching the summaries stored in the knowledge base. The high-level idea of the matching procedure is to examine the new client method, and check if there exists any program path which may lead to the same argument being passed into the library API. Technically, this is done by enumerating paths of the new client and deriving traces (i.e., operation sequences) for specific arguments, which are then checked against the FSM summaries for acceptance. If an argument trace is accepted by the FSM, then it is possible for the new client to reproduce the same argument value causing incompatibilities previously. Note that it may not necessarily require the arguments to have the exact same values, in order to trigger the same incompatibility issues. We design a number of heuristics to balance the matching accuracy and the ability to discover more issues, which will be discussed further in Section 4.2.1.

Figure 2d shows an example client method that is successfully matched with the knowledge base. The *serialize* method from the project Myth [23] invokes the same API as highlighted (Line 5). The first argument (*kryo*) has a potential trace: "*Kryo.<init>()* → ⊙", which is accepted by the FSM of *arg_0*. Similarly, the second argument (*output*) has a potential trace: "*Output.<init>(OutputStream)* → ⊙ → *Output.flush()*", which is accepted by the FSM of *arg_1* when ignoring the operation after the API call (⊙). The third argument (*obj*) is passed directly from outside, thus assumed to match with any summary. In this case, all three arguments have traces accepted by the stored summaries, which is a strong indication that the same incompatibility issue exists. Thus, we give higher priority to such client and generate tests to reproduce and confirm the incompatibility.

Figure 2e shows an example client that is not matched. The *save* method is from Artemis-odb [6] and the reason for not matching is highlighted in red (Line 4). The only possible trace for *kryo* is: "*Kryo.<init>(ReferenceResolver)* → *kryo.setRegistrationRequired(false)* → ⊙", which is not accepted by the FSM of *arg_0*. The key issue here is the extra method call *setRegistrationRequired* which changes the calling contexts of *writeObject* and makes it less likely to trigger the same incompatibility issue than the previous examples. We do not attempt to generate tests for such clients. This is also confirmed by our observation that the same issue does not happen here when *registrationRequired* is *false*.

CompCheck applies the stored argument values in the knowledge base to reproduce the incompatibility on matched clients with best efforts. Figure 2f shows a test generated by CompCheck to manifest the incompatibility in *serialize* (Fig. 2d). The test first loads a stored object argument, namely, *boo0*, and then invokes the target API with it (Line 5). *KryoSerializer* is the class containing the *serialize* method. The oracle used for the generated tests is that the return value of the client (if

$$
\begin{array}{lcl}
P & ::= & \overline{L} \\
L & ::= & \texttt{class } C \texttt{ extends } C\{\overline{C\ f};\ \overline{K}\ \overline{M}\} \\
K & ::= & C(\overline{C\ f})\{\texttt{super}(\overline{f});\ \texttt{this}.f = f;\} \\
M & ::= & C\ m(\overline{C\ x})\{\overline{S}\} \\
S & ::= & x := e \mid e.f := e \mid e.m(\overline{e}) \mid \texttt{return } e \mid \\
 & & \texttt{if}(b)\texttt{then}\{\overline{S}\} \mid \texttt{goto } n \\
e & ::= & x \mid e.f \mid e.m(\overline{e}) \mid \texttt{new } C(\overline{e}) \mid (C)e \\
b & ::= & \texttt{true} \mid \texttt{false} \mid b \ \&\& \ b \mid !b \mid e == e
\end{array}
$$

Fig. 3. Syntax rules for a simple Java-like language.

one exists) should stay the same, and there should not be uncaught exception thrown. We elaborate more on how we establish the test oracles in Section 4. In this case, the test reveals incompatibility when the *kryo* library is upgraded by triggering a runtime exception.

3 BACKGROUND

In this section, we provide the necessary background and define the terminologies which will be used in the remainder of the paper.

To keep the presentation of our technique concise, we step back from the complexities of the full Java language and concentrate on its core features. We adopt a simple functional subset of Java based on *Featherweight Java* [43]. The syntax rules of the language are given in Fig. 3. A program P consists of a list of class declarations ($\overline{L}$), where the overhead bar $\overline{L}$ stands for a (possibly empty) sequence $\langle L_1, \ldots, L_n \rangle$. We use $\langle\rangle$ to denote an empty sequence and comma for sequence concatenation. Every class declaration has *members* including *fields* ($\overline{C\ f}$), *methods* ($\overline{M}$), and *constructors* ($\overline{K}$). A method *body* consists of a sequence of statements, which can either be *local assignment*, *field assignment*, *method call*, *return*, *conditional branch*, or *goto* (loops are omitted). The expression assigned or returned can be a variable, a field access, a method call, an instance creation, or a typecast.

In a program P, some of the classes, $L^l \in \overline{L}$, are designated as libraries (e.g., a jar file in Java), whose methods are called by the methods of the client classes L^c. A *client-library (method) pair*, $(M^c, (M^l_1, M^l_2))$, consists of a client method M^c and two versions of the same library method, namely, M^l_1 and M^l_2. The two library methods may have different bodies but share the same signature, thus, can be called interchangeably from the client method. A program P may contain multiple client-library pairs. In our scenario, the source code of libraries is not necessarily available, but the Application Programming Interface (API) is always known in advance.

We also assume standard program semantics and are particularly interested in the *program traces* generated during runtime.

Definition 1 (Program Trace). *Given a program p, a program trace π is a sequence of statements, $\pi = \langle s_1, \ldots, s_m \rangle$, traversed during an execution, where $s_i \in S$.*

We say a trace π_2 is a *subtrace* of π, if there exist traces, π_1 and π_3, such that $\pi = \pi_1, \pi_2, \pi_3$. We say π_2 is a *suffix* of π, when π_3 is empty. Let V be the set of all values evaluated during an execution.

Definition 2 (Method Trace). *Let π be a program trace. A method trace $m(\pi)$ can be derived from π by retaining only the method call statements, i.e., $m(\pi) = \langle \overline{y_i} := m_i(\overline{x_i}) \rangle$, where m_i is the method signature, $\overline{x_i} \in V^n$ is the list of input values (including accessed fields), and $\overline{y_i} \in V^m$ is the list of output values (including fields).*

The method calls appear in the method trace according to the order they are pushed to the call stack. Then the *observable behaviors* of a method M on a program trace π are the set of all observed

input/output value pairs of M, i.e., $\text{Ob}(M, \pi) = \{(\overline{x_{M_i}}, \overline{y_{M_i}})\}$. We use $\text{Ob}(M, P) = \cup_{\pi \in P} \text{Ob}(M, \pi)$ to denote the set of all observable behaviors of M in all possible traces of P. Similar subtraces can be defined for object instances as well.

Definition 3 (Object Trace). *Let π be a program trace and x be an object instance. An object trace, $o(\pi, x)$, can be derived from π by retaining only the assignment statements to and method calls on x, i.e., $o(\pi, x) = \langle s_i \in \pi \mid s_i \in \{\text{`}x := \_\text{'}, \text{`}\_ := x.m(\_)\text{'}, \text{`}x.m(\_)\text{'}\}\rangle$.*

Intuitively, an object trace contains all operations which may have an side-effect on the state of the instance. Given a method call, $\overline{y_i} := m_i(\overline{x_i})$, we can collect object traces for all its input arguments $\overline{x_i}$ and call it the *argument context* of m_i. Now we define the problem of *library upgrade incompatibility* as follows.

Definition 4 (Library-Upgrade Incompatibility). *We say a program P is library-upgrade incompatible if there exists a client-library pair, $(M^c, (M^l_1, M^l_2))$, such that,*

$$\exists(\overline{x_i}, \overline{y_i}) \in \text{Ob}(M^c, P), \exists(\overline{x_j}, \overline{y_j}) \in \text{Ob}(M^c, P[M^l_1 \mapsto M^l_2]) \cdot \left(\overline{x_i} = \overline{x_j} \wedge \overline{y_i} \neq \overline{y_j}\right),$$

where $P[M^l_1 \mapsto M^l_2]$ means the program with M^l_1 substituted by M^l_2.

Intuitively, a library upgrade is incompatible with respect to a client, if the change in library results in an observable behavioral difference of the client, i.e., same inputs producing different outputs. In general, a library upgrade incompatibility can be a compilation error or a behavioral incorrectness. In this paper, we only focus on the latter, which is much harder to detect.

4 TECHNIQUE

Figure 4 shows an overview of CompCheck. CompCheck consists of two major components: knowledge mining and incompatibility discovery. For knowledge mining, CompCheck maintains and grows a knowledge base of incompatible library upgrades, initially bootstrapped from a small portion of open-source projects on GitHub. The incompatibility issues are mined by running accompanied regression test suites while upgrading all library dependencies one-by-one (details discussed in Section 4.1.1), which is then stored along with contextual information such as execution traces. The collected information is further consolidated to form the knowledge base. Next, in the incompatibility discovery component, given a new target client project, CompCheck tries to match it with the stored client contexts from the knowledge base. If a client method is matched, CompCheck attempts to generate unit test cases, guided by the knowledge observed previously, to reveal and confirm the incompatibility in the next client. The generated test cases help developers to locate and fix the incompatibility, and once confirmed in the new project, can be used to enhance the existing regression test suite as well as the knowledge base.

Our envisioned usage scenario of CompCheck is a long-running and self-evolving service deployed on the cloud, analogous to commercial solutions (e.g., Dependabot [19]). This way, the knowledge base of CompCheck may continuously grow by learning from new clients and new tests in a scalable manner. To expand the knowledge base, one needs to run available regression tests at the client sides, for every newly discovered library version. If integrated with continuous integration (CI) tools, CompCheck does not introduce much overhead beyond the regular regression testing. Over time, the knowledge base is to be maintained when some library versions become obsolete and are no longer used by any client.

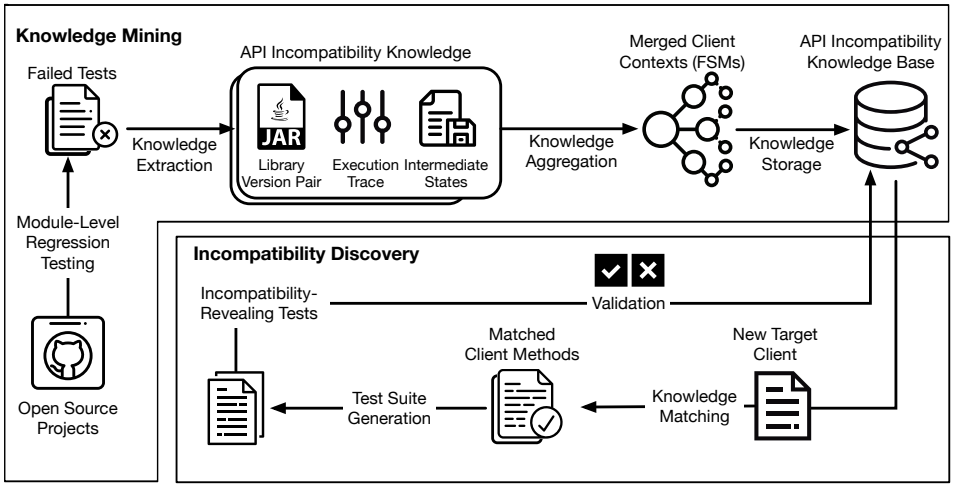


Fig. 4. Overview of the CompCheck workflow.

4.1 Knowledge Mining

The knowledge discovery component consists of three main phases: module-level regression testing, knowledge extraction, and knowledge aggregation.

4.1.1 Module-level Regression Testing. To obtain incompatible API upgrades, we perform module-level regression testing, i.e., upgrading libraries and checking if any previously passing client tests flip to fail, on a set of collected GitHub projects. The initial dataset was collected from 1,225 top-rated Java projects on GitHub which use Maven as their build system. We focused on Maven projects because of the technology maturity. We then performed module-level regression testing on this dataset. For each project in the dataset, we first ran its entire test suite, ensuring that all the tests pass in the base version. A project was dropped if it encountered any compilation error or test failure. To alleviate the harm of potential flaky tests, we ran the test suite three times, dropping a project if the test suite fails in any single run.

For the remaining projects, we checked if any of their external dependencies (i.e., jar files) are upgradable. For each upgradable jar file, we upgraded it to the latest version on Maven Central [55] and reran the test suite. If any test failed in this run, we considered the library has a backward incompatibility issue with the client project. Note that we only upgrade a single library each time to isolate failures caused by different libraries. The result of this step is a set of test cases which flip to fail after upgrading. Although CompCheck relies on regression testing to mine knowledge at a massive scale, the regression test suite of a sole client is usually not sufficient to detect potential incompatibility without prior knowledge about library upgrades.

4.1.2 Knowledge Extraction. With tests failing due to upgrades, CompCheck then extracts information about the nature and context of the incompatibility issues from method traces (Definition 2) collected from test executions. The traces were collected by instrumenting the client code and recording necessary execution information. For each failed test execution, the recorded method trace includes the method signatures and input/output values, in the execution order.

To identify the library method which triggers the test failure in the client, we consider two cases: (1) when the failure is caused by a runtime exception, the incompatible API call is the latest library method along the trace called from the client; and (2) when the failure is caused by an assertion

violation, the culprit can be determined by comparing the two method traces obtained before and after the upgrade. More specifically, an assertion violation may not always be caused by the closest library call, because the client's behavior difference can also be introduced much earlier on. The assertion violation indicates a behavioral difference of the client. Since there is no change in client code, we must have at least one library call along the trace which takes the same inputs, but returns different outputs. We consider the first such library call as an incompatible API call. We record the incompatible APIs and their client method in the form of client-library pairs, which by definition satisfy the criteria in Definition 4.

After identifying a client-library pair, the next step is to extract contextual information from the client, which can be used later to search for clients with similar issues. We first record the argument context $\overline{\phi}$ (C.f. Section 3) of the incompatible API. Note that if the API method is not static, the receiver object is treated as its first argument (*arg_0*). For each argument, we record the values passed into the API if it is of the primitive type, and the object traces (see Definition 3) if it is of an object type. In practice, we also record the intermediate states of the object, between each operation along the object trace. Therefore, for an object argument, we record the method call sequence on it and the value (i.e., state) of the object after each method call in that sequence. The output of this step is a *knowledge record k* specific to one incompatible API occurrence, which includes a client-library pair along with its argument context.

4.1.3 Knowledge Aggregation. Next, we merge the extracted knowledge record with the accumulated knowledge base so far. This is essentially merging the argument contexts of all occurrences of the same API. Specifically, for each argument, we merge values for primitive types into a set and compactly represent object traces as an FSM. The merging of object traces is performed following the automaton learning algorithm proposed by Oncina and Garcia [65]. Given a set of traces, the algorithm recursively merges the sequences that have common prefixes, and generates a merged FSM as the outcome in polynomial time. It does not attempt to minimize the merged FSM though. Therefore, the resulting knowledge base contains aggregated knowledge records in the form of $k = (M^c, (M^l_1, M^l_2), \overline{\Phi})$, where $\overline{\Phi}$ is the merged argument contexts.

4.2 Incompatibility Discovery

The incompatibility discovery component leverages the knowledge base to check potential incompatibility issues in new target client projects. It consists of three phases: knowledge matching, test generation, and test validation. Algorithm 1 describes the first two phases. Given the incompatible API M^l, its target caller's control flow graph G, argument context Φ, intermediate states Λ (extracted from Φ), and matching strategies S (we introduce it later in this section), CompCheck attempts to generate incompatibility-revealing tests for the target API call site.

4.2.1 Knowledge Matching. In knowledge matching, CompCheck scans the target client while going through the knowledge base, to locate the call sites of all the known incompatible APIs. It performs context matching when finding such a call site. For matching, CompCheck builds a control flow graph of the client method which is used to statically compute all possible object traces for each argument of the API. The control flow graph is intra-procedural. It only analyzes the client method that directly contains the target call site. We limit the analysis scope to intra-procedural to reduce costs, but leave it configurable. In practice, one could perform more fine-grained analysis, e.g., field values, by modifying this configuration. We then check if any of these object traces is accepted by the FSM in the knowledge base. For each object argument of API, we perform a *suffix matching* on its object trace in the caller with the corresponding FSM.

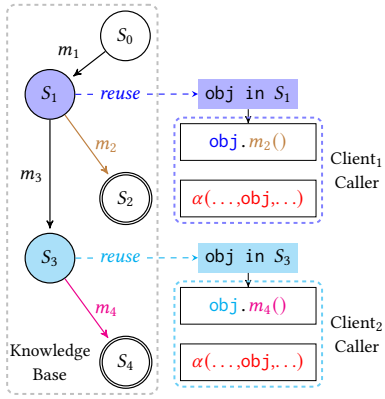


Fig. 5. Suffix matching and object reusing.

Figure 5 illustrates suffix matching together with object reusing. Here, two client methods (Client$_1$ and Client$_2$), having different object traces, are to match with an FSM in the knowledge base. Starting with the acceptance state, we do a backward matching between the object trace and the stored argument context, following the inverse of the transitions, until we end up with a state in the stored FSM (indicating a match) or some unseen states (indicating a mismatch). The rationale of suffix matching is that by connecting a previously stored intermediate state to the matched suffix, we can accurately replay the state of an incompatibility-triggering object argument.

For instance, in Fig. 2d, the object trace of *arg_1* of *writeObject* in the new client is accepted by the FSM: *Output.<init>(OutputStream)* $\rightarrow \odot$. For *arg_0*, it is an argument of the caller, and is directly passed to the API. Therefore, it matches with the acceptance state directly, as we can load any stored final states of *Output* to create a known incompatibility-triggering argument. In contrast, in Fig. 2e, the object trace of *arg_0* fails to match with the corresponding FSM. As indicated by the red line, it executes one more transition *Kryo.setRegistrationRequired(false)* after reaching the end state (via transaction *Kryo.<init>()*). Semantically, the *Kryo* object explicitly sets no registration required, thus the API *writeObject(Output,Object)* will not trigger the exception after an upgrade. These examples confirm the value of our client-specific approach. Compared with client-agnostic techniques, we can differentiate that Fig. 2d is a similar usage as the knowledge (Figs. 2a and 2b), while Fig. 2e is a different usage.

Matching Strategies. Multiple matching strategies can be applied in context matching (Algorithm 1, Line 16). They collectively determine whether a call site of an incompatible API matches the corresponding knowledge. Intuitively, the most strict matching (requiring an exact match on the argument context) may not be necessary, as reasonably different argument values might still be able to trigger the issue. In such cases, tolerating certain differences of argument context can improve recall with little or no loss of precision. As an example, in Section 2, an *Output* object, either created by *Output.<init>(OutputStream)* or by *Output.<init>(OutputStream, int)*, triggers the same incompatibility issue. Our experiment also confirms that this intuition holds in many cases in practice.

Our matching strategies have two dimensions: (1) for each individual argument, we use two switches to control their matching; and (2) for the API call site as a whole, we compute its confidence score and empirically set a matching threshold. Specifically, the two switches controlling the matching of an individual argument are: **(S1) Relax Polymorphism**: if enabled, when matching the object traces of two arguments, we allow matching other objects under inheritance relationship,

Algorithm 1: Algorithm for incompatibility discovery

input : M^l: the incompatible API; G: the control flow graph of M^c; Φ: the merged argument context of M^l; Λ: the stored intermediate states of M^l; S: matching strategies;
output : t – an incompatibility-revealing test of M^l

```
G⁻ ← G with all the edges reversed;
foreach o ∈ GetObjectArgs(M^l) do
    G_o⁻ ← G⁻(o); Φ_o ← Φ(o); Λ_o ← Λ(o);
    stored_state ← MatchArgumentContext(G_o⁻, Φ_o, S);
    if s ≠ null then
        s ← LoadState(stored_state);
        t ← GenerateTest(s, M^l);
        return t;
    return ∅;
Function MatchArgumentContext(G_o⁻, Φ_o, Λ_o, S):
    ACC ← {∀φ ∈ Φ_o s.t. φ = acceptance};
    foreach acc ∈ ACC do
        current ← acc; i ← IndexOf(current, Φ_o);
        foreach j ∈ |G_o⁻| do
            m ← G_o⁻[j];
            if Match(current, m, S) then
                if m = G_o[0] then return Λ_o[i] ;
                − − j;
                current ← Φ_o[− − i];
            else break ;
    return null;
```

overriding methods, and overloading methods; otherwise, we require an exact match on object types and methods. **(S2) Relax Primitive**: if enabled, when matching two primitive arguments (including string and class literal), we do not require the values to be exactly the same. As an example usage of the switches, assume our knowledge only contains a single transition, *Output.<init>(OutputStream)*, for *arg_1* in Fig. 2c. When matching a new client calling *Output.<init>(OutputStream, int)*, if S1 is enabled, CompCheck considers it as a match, otherwise, a mismatch.

We quantifies the confidence that the whole API call site matches another using a confidence score, which is calculated as follows:

$$\text{Confidence} = N_{\text{matched}}/N_{\text{total}} \tag{1}$$

where N_{matched} and N_{total} denote the number of matched and the total number of arguments of method, respectively. By adjusting the threshold of the confidence score, we control how many arguments are required for the target call site to match.

Besides the strategies introduced above, based on our empirical studies (C.f. Section 5), we also found that project configurations (e.g., *pom.xml* files) provide useful insights which are helpful for context matching. We investigate the impact of these strategies in Section 6.1.

4.2.2 Test Generation. With a matched call site of incompatible API, CompCheck attempts to generate incompatibility-revealing tests utilizing the corresponding knowledge. Our test generation depends on whether CompCheck has control over any API argument from the outside of its caller. There are three cases: (1) When CompCheck *directly* controls an argument of the API outside the caller, i.e., an argument of API is also an argument of the caller, which is passed from caller to API after a sequence of operations. In such cases, our test case can reuse a stored object state

by passing it to the caller, as shown in Fig. 5. (2) When CompCheck has only indirect control on an argument of the API outside the caller, i.e., an argument of API has data dependency with a caller argument, but conversion is needed for obtaining API argument from the caller argument. In such cases, we cannot directly reuse stored states. As such, we developed an optimization, *type conversion table*, for CompCheck to handle such cases if the conversion is simple and has been observed previously. (3) When CompCheck has no control over any argument of the API outside the caller, then CompCheck performs standard search-based test generation.

CompCheck's test oracle generation is two-fold. (1) If the incompatibility issue is manifested as a runtime exception, we generate a test with no explicit oracle. During the test execution, if any exception is thrown, the test is considered failing. (2) If the incompatibility issue is manifested as an assertion violation, i.e., different output values are produced, we determine the test oracle based on heuristics. Specifically, we invoke the client method with the generated test inputs under the old library version, collect its output values, and use them as oracles.

The caller method of the target API may have some extra arguments that are irrelevant to the API call site, but necessary for it to be invoked. CompCheck performs standard search-based generation to generate such arguments. CompCheck also employs another optimization—*caller slicing*—to eliminate some irrelevant complex inputs by slicing off arguments and statements with no dependency on the API call site.

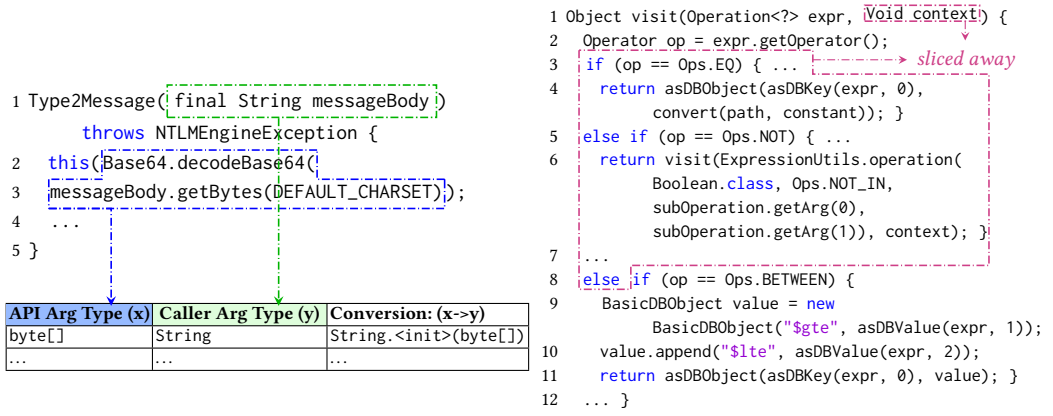
4.2.3 Test Validation. For each generated test, CompCheck executes it with both the old and new libraries. If a test case passes with the old library but fails with the new one, it is a valid incompatibility-revealing test. Finally, CompCheck performs knowledge mining on the generated incompatibility-revealing tests, merging it with the existing knowledge base.

By consolidating newly obtained knowledge, CompCheck keeps evolving its knowledge base. This could be extremely beneficial in an industrial environment, e.g., setting up CompCheck on a centralized code repository of a company and iterating from time to time. For scalability, we constantly merge the summary of argument traces and drop redundant ones. In the future, we also plan to compute the similarity of object states and drop a state if a similar one already exists in the knowledge base.

4.2.4 Illustrations of the Optimizations. As stated in Section 4.2.2, CompCheck employs two optimizations for improving the effectiveness of knowledge matching and test generation. Here we provide details on these optimizations together with illustrative examples.

Type Conversion Table. The first optimization is building a type conversion table for each incompatible API in the knowledge base, to make the mined knowledge applicable to more similar contexts. The transformation table stores the mapping between each argument type of the API and the types which can be directly (i.e., in one step) obtained from the argument. This way, even if a new API call site requires a different type of value to be passed in, we can still utilize the knowledge with proper conversion in place. We build the type conversion table by scanning the bytecode of clients. We then store the transformation and the values before/after the conversion. For primitive values, we store the standard conversions.

Figure 6a shows an example where the type conversion table helps in the project HttpComponents [32]. The caller method *Type2Message.<init>(String)* takes a string argument and converts the string into a byte array, before passing it into the API *Base64.decodeBase64(byte[])*. In the type conversion table, CompCheck stores the API's argument type (highlighted in blue), the caller's argument type (highlighted in green), and the conversion method from the API argument type to the caller's argument type. In this case, the API takes an argument of *byte[]* type, while the caller takes a *String* type. Therefore, when generating tests, CompCheck reuses a byte array saved in the



(a) Optimization with type conversion table

(b) Optimization with caller slicing

Fig. 6. Illustrations of the optimizations employed by CompCheck.

knowledge base, using the method *String.<init>(byte[])* to do the conversion, and then invokes the caller method with the obtained *String* object to reveal the incompatibility issue.

Caller Slicing. The second optimization is to perform slicing for the caller of the target API, dropping arguments and statements with no effect on the API call site. It enables us to (1) avoid generating some complex objects that are irrelevant to the target call site, e.g., a special file that is not used by the call site, and (2) reach a certain branch containing the target call site without guessing inputs to bypass complex branch conditions. The sliced caller method preserves the same calling context, when the API call site is reached.

Figure 6b illustrates how caller slicing helps with test generation in the project Querydsl [72]. In this example, the target incompatible API is the constructor of *BasicDBObject* (Line 9). The caller method *visit* takes two arguments. The second argument *context* is used only at Line 6, which is on a different conditional branch thus has no data dependency with the API call site. Without caller slicing, to generate a test invoking the *visit* method, CompCheck's test generator has to first generate three object instances, including an instance of the class containing the *visit* method (*MongodbSerializer* in this case), an instance of *Operation* class, and an instance of *Void* class. In contrast, with caller slicing enabled, as illustrated by Fig. 6b, CompCheck can drop the *context* argument as it has no effect on the target API call site (Line 9), and slice away all the branches that have no effect on the target API call site, including the branches at Lines 3–4 and Lines 5–6. In this way, the test generator only needs to search for the first two objects, the search space is therefore reduced.

5 A STUDY ON THE LIBRARY UPGRADE INCOMPATIBILITY KNOWLEDGE BASE

To understand the distribution and manifestation of library upgrade incompatibility issues, we conduct a manual empirical study on our knowledge base. Specifically, we aim to answer the research question: **RQ1**: What are the manifestation patterns of incompatibility issues in Java projects?

The knowledge base for the study is built on 651 client projects collected from GitHub, which are written in Java and use Maven as build system. Starting with 1,225 top-starred Maven projects on GitHub, we filtered out 574 projects that have compilation errors or test failures in the initial

execution, resulting in 651 projects for extracting knowledge. We collected backward incompatible client-library pairs through the module-level regression testing process, which is stated in Section 4.1.1, For each client project, we checked if any of its libraries are upgradable with Maven's Versions Plugin [69]. For each upgradable library, we upgraded it to the latest version on Maven Central [55] and reran the test suite. If any test failed in this run, it means that the library has a backward incompatibility issue with the client. We only upgrade a single library each time to isolate failures caused by different libraries.

In total, we discovered 758 backward incompatible client-library pairs $(M^c, (M_1^l, M_2^l))$. Of these pairs, 386 only have runtime errors, 265 only have assertion violations, and 107 have both types of failures. The 758 incompatible client-library pairs include 175 unique clients and 430 unique libraries in total. On average, each client uses 4.3 incompatible libraries, while each incompatible library corresponds to 1.8 clients. There are 388 incompatible client-library pairs having behavioral incompatibility issues that cannot be simply detected by recompiling, which correspond to 114 unique clients and 266 unique libraries. The affected clients have 2.5 K stars on GitHub on average, being popular in the developers' community and can influence many potential users. The total number of tests failed due to incompatibility is 58.8 K. The average LOC of the tests that failed due to incompatibility is 17.9. In the knowledge base, the average number of arguments of each incompatible API is 1.8. The statistics show that an incompatibility issue of a library can affect multiple clients, and the incompatibility issues are often complex and can be time-consuming for developers to diagnose. In this study, the time overhead for running CompCheck is 5.6 seconds per test method and 91.3 hours in total, respectively. The knowledge base takes 211.3 KB per context on average and 11.9 GB of disk space in total respectively.

We further inspected the failed tests to study the characteristics of incompatibility issues. For feasibility, we randomly sampled 200 client-library pairs, and randomly selected one failed test from each pair for inspection. The goal of the inspection is to determine and categorize the incompatibility manifestation pattern of each pair. Two authors inspected these cases independently and discussed the findings. We dropped 23 pairs as due to the complexity of the client code, we were unable to identify the root causes of these incompatibility issues, thus were uncertain which manifestation pattern should be assigned to them. In the end, we had inspection results of 177 failed tests, of which 120 are runtime errors and 57 are assertion violations.

From the observations, we summarized three common incompatibility manifestation patterns: (1) *direct incompatibility* happens when the incompatible API is directly invoked by the client, e.g., Fig. 2; (2) *transitive incompatibility* happens when the incompatible API is not invoked by the client, but by another dependant library of the client; and (3) *co-evolution incompatibility* happens when multiple dependent libraries of a client are supposed to maintain the same version number or follow some constraints on their versions. Upgrading one library without the others accordingly triggers incompatibility, in either the upgraded library or the remaining ones.

Co-Evolution Incompatibility. Figure 7 shows the *pom.xml* file of an example of co-evolution incompatibility. The client is JVM-Profiler [70]. Jackson-core [27] and Jackson-databind [28] are two dependent libraries that are supposed to evolve together (always using the same version) when upgrading. Only upgrading Jackson-databind from the old version (2.9.10.1, the value of *jackson.version*) to the new version (2.10.1) makes Jackson-databind throw a runtime exception, as it uses the method *JsonGenerator.writeStartArray* of Jackson-core, which does not exist in the old version. However, if upgrading both libraries to 2.10.1, the incompatibility issue disappears, as their co-evolution constraint is satisfied.

Usually, co-evolving libraries are distributed under the same organization. However, to the best of our knowledge, there is no standard rule stating how these version constraints should be

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>${jackson.version}</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>${jackson.version}</version>
</dependency>
```

Fig. 7. Co-evolving dependencies declared in *pom.xml*.

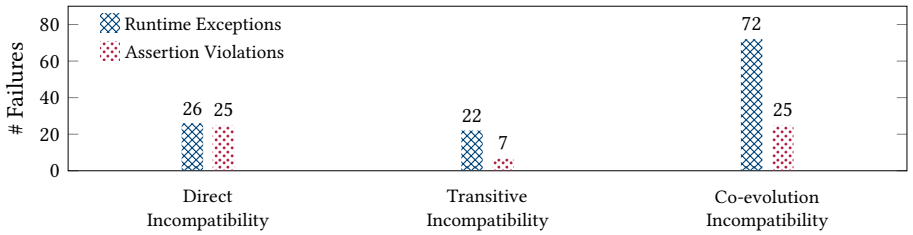


Fig. 8. Distribution of incompatibility patterns.

documented and maintained. Based on our observation, sometimes the constraints are explicitly documented by library developers [40], while in some other cases the constraints are implicitly treated as a convention by experienced users [35]. Therefore, novice users may miss such requirements when integrating such a library into their projects, which potentially causes incompatibility issues in the future. Note that the sets of instances for the three manifestation patterns are disjoint. Although the instances of the co-evolution incompatibility pattern can also be assigned to direct or transitive incompatibility, their distinguishable characteristics (e.g., sharing the same group ID) make them easier to identify and fix automatically, compared with regular incompatibility issues. Therefore, we assign them to a separate pattern.

Figure 8 shows the distribution of different patterns in the dataset. Of all the inspected incompatibility issues, 67.8% manifest themselves as runtime exceptions, while 32.2% are assertion violations. Direct Incompatibility, transitive incompatibility, and co-evolution incompatibility constitute 28.8%, 16.4%, and 54.8% of the failures, respectively.

As it indicates, more than half of the inspected issues are co-evolution incompatibility. This has two implications. First, co-evolution information can help with context matching, since some incompatibility issues can only be triggered if both co-evolving libraries are present in the same project. As an evidence, in Section 6.1, we show that incorporating co-evolution information in context matching can achieve the best matching effectiveness. Second, if a library consists of multiple individual artifacts which only operate correctly with the right combinations of versions, then such constraints should be explicitly documented, to avoid misuses by client developers.

Based on these implications, we added the consideration of co-evolution issues into our matching strategies, which can potentially improve the matching accuracy. Thus, we extended our basic confidence score formula (Eq. (1)) to incorporate this factor (see Eq. (2)).

$$\text{Confidence} = 0.5 \times (N_{\text{matched}}/N_{\text{total}}) + 0.5 \times F_{\text{coevo}}. \tag{2}$$

In Eq. (2), $F_{\text{coevo}} = 1$ when the configuration file (*pom.xml*) of the target client contains the same co-evolving libraries as a known failing client in the knowledge base. Otherwise, $F_{\text{coevo}} = 0$. In Section 6.1, we investigate the improvement of Eq. (2) compared with Eq. (1) on matching accuracy.

Answer to RQ1: Our study discovered that there are three manifestation patterns of incompatibility issues in Java projects: direct incompatibility, transitive incompatibility, and co-evolution incompatibility. They constitute 28.8%, 16.4%, and 54.8% of the failures, respectively.

6 IMPLEMENTATION AND EVALUATION

We implemented CompCheck with a combination of Java programs and Python scripts. We used the ASM [7] Java bytecode instrumentation framework to collect and analyze execution traces, and the XStream [89] library for saving object states. We stored the knowledge base and object traces in the JSON and XML formats, respectively. We built control flow graphs of clients and implemented caller slicing using Soot [39]. We implemented the test generation component as an extension of the EvoSuite [33] search-based test generator. When generating an argument of the caller method of target API, if the context matches knowledge and CompCheck has control over an argument, then CompCheck loads a stored object, instead of searching from scratch. Besides, we customized EvoSuite to not use any code entity from libraries that are removed during the upgrade. This is to prevent the target call site from being hidden by shallow errors, e.g., *NoClassDefFoundError*, which are less interesting as they can be easily located and fixed. When executing test generation, we set the target class/method for EvoSuite as the class/method containing the call site, using default values for other parameters.

Research Questions. We aim to answer the following research questions. **RQ2**: How do different matching strategies affect the effectiveness of context matching? **RQ3**: How effective is CompCheck and its incompatibility discovery in revealing incompatibility issues on new target call sites? **RQ4**: How effective is CompCheck compared with existing techniques in revealing incompatibility issues? **RQ5**: What are the contributions of CompCheck's object reusing and optimizations, to CompCheck's overall effectiveness?

6.1 Impact of Matching Strategies

To study the impact of different matching strategies, we first sampled a set of incompatible APIs from the knowledge base, then manually labeled a set of call sites for each API. Finally, we ran CompCheck's context matching on these call sites with different matching strategies, comparing the precision and recall of each strategy.

Table 1 shows the backward incompatible APIs used in our experiments. From left to right, the columns show the API IDs, API names and argument types, library names, old version numbers, new version numbers, and the number of known call sites of the APIs in the knowledge base, respectively. Some names are shortened to save space, and an expanded table is available online [18]. We used all the 24 incompatible APIs as the knowledge for the matching experiment, requiring that each knowledge must have at least two known failing call sites.

We extracted call sites of APIs from the projects we collected in Section 4.1.1, i.e., the 1,225 top-starred Maven projects on GitHub. For each API, we went through the client projects and extracted its call sites from the projects that use the API. We kept adding call sites until most APIs in Table 1 have no less than five call sites. In total, we collected 202 call sites from 37 client projects, which is shown in Table 2. During the collection, we found that many call sites did not expose incompatible behaviors in the module-level regression testing: they were either not covered by the developers' test cases, not checked by the existing test assertions, or their execution did not trigger

Table 1. Backward Incompatible APIs.

Id	Incompatible API	Library	(lib, lib')	$\#M^c$
k1	Kryo.writeObject(Output,Object)	Kryo [24]	(3.0.3, 5.0.0-RC4)	3
k2	CmdLineParser.<init>(Object)	Args4j [46]	(2.0.23, 2.33)	2
k3	ObjectMapper.writeValueAsString(Object)	Jackson [28]	(2.9.10.1, 2.10.1)	2
k4	MethodVisitor.visitFrame(int,int,Object[],int,Object[])	ASM [7]	(5.1, 7.2)	3
k5	Type.getType(String)	ASM [7]	(5.0.4, 7.2)	2
k6	Guice.createInjector(Module[])	Guice [38]	(4.0, 4.2.2)	4
k7	Reflections.<init>(String,Scanner[])	Reflections [73]	(0.9.9-RC1, 0.9.11)	2
k8	ObjectMapper.readValue(InputStream,Class)	Jackson [28]	(2.8.11.3, 2.10.1)	3
k9	Base64.decodeBase64(byte[])	Codec [31]	(1.12, 1.13)	2
k10	Schema.writeTo(Output,Object)	Protostuff [71]	(1.5.9, 1.6.2)	2
k11	BasicDBObject.<init>(String,Object)	MongoDB [60]	(3.5.0, 3.12.0)	3
k12	LoggerFactory.getLogger(String)	SLF4J [82]	(1.7.9, 2.0.0-alpha1)	4
k13	EqualsVerifier.forClass(Class)	EqualsVerifier [66]	(1.7.5, 3.1.11)	2
k14	DateTimeFormatter.parseDateTime(String)	Joda-Time [45]	(2.8.1, 2.10.5)	2
k15	DateTimeFormatter.parseMillis(String)	Joda-Time [45]	(2.8.1, 2.10.5)	2
k16	HttpClient.execute(HttpUriRequest)	HttpComponents [3]	(4.5.3, 4.5.10)	3
k17	Json.jsonPath(String)	WebMagic [87]	(0.6.1, 0.7.3)	3
k18	Logger.getName()	SLF4J [82]	(1.7.25, 2.0.0-alpha1)	3
k19	HtmlCleaner.clean(String)	HtmlCleaner [42]	(2.5, 2.23)	3
k20	Client.handle(Request)	Restlet [75]	(2.2.1, 3.0-M1)	2
k21	JSONObject.toJSONString()	Fastjson [1]	(1.1.41, 1.2.62)	2
k22	DefaultHttpClient.getCookieStore()	HttpComponents [3]	(4.2.5, 4.5.10)	3
k23	ObjectMapper.readTree(String)	Jackson [27]	(2.9.9, 2.10.1)	2
k24	URIBuilder.build()	HttpComponents [3]	(4.5.1, 4.5.10)	2

errors with the developer-provided inputs. This further motivates our work. When incompatibility issues cannot be exposed by the developer's existing tests, client developers may utilize CompCheck to enhance their test suite to test the compatibility of their libraries. For each client project, we show its URL on GitHub, LOC, the number of stars, and the version (SHA) analyzed. We used all the 202 call sites as the targets for the matching experiment.

We manually labeled each call site through inspection and test creation. If we could manually create an incompatibility-revealing test for a call site, we labeled it as revealable (+); otherwise, if we failed to create a test to expose its incompatibility, we labeled it as unrevealable (-). As a result, of the 202 call sites, we labeled 104 as revealable and 98 as unrevealable. A call site is revealable means that its incompatibility can be exposed by a manually created test. Therefore, an unrevealable call site is "compatible" in theory, except that there could be some noises due to mislabeling. We discuss the limitation of the labeling in Section 6.4. Intuitively, a call site is unrevealable if given its context, the API's incompatibility cannot be exposed. A typical example of an unrevealable call site is Fig. 2e. The client calls *kryo.setRegistrationRequired(false)* (Line 4) to explicitly disable the registration checking, thus the API call site (Line 6) can never trigger the incompatibility ("IllegalArgumentException: Class is not registered"), no matter what arguments are provided to the API.

$$\text{Precision} = \frac{\text{tp}}{\text{tp} + \text{fp}} = \frac{|\{\text{CompCheck matched}\} \cap \{\text{revealable}\}|}{|\{\text{CompCheck matched}\}|} \tag{3a}$$

$$\text{Recall} = \frac{\text{tp}}{\text{tp} + \text{fn}} = \frac{|\{\text{CompCheck matched}\} \cap \{\text{revealable}\}|}{|\{\text{revealable}\}|} \tag{3b} \quad \text{and} \quad F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3c}$$

Table 2. Client Projects for the Experiment.

URL (https://github.com/)	$\mathcal{K}$	KLOC	#Stars	SHA
alibaba/druid	k12	295.7	25,234	1d784d12
apache/hive	k9	1,704.5	4,587	358e5a93
apache/httpcomponents-client	k9	284.6	931	a3f24e97
apache/incubator-pinot	k13	278.8	3,849	bbf29dc6
apache/servicecomb-java-chassis	k10	203.6	1,554	2701f56f
aws/aws-sdk-java	k14, k15	7,340.2	3,707	28f5c5f3
bazaarvoice/jolt	k6	172.7	929	a07d777c
bguerout/jongo	k7, k11	36.3	558	7f15d205
brianway/webporter	k17	19.6	2,611	01795a07
BroadleafCommerce/BroadleafCommerce	k24	188.9	1,513	6129f6ed
code4craft/webmagic	k19	12.7	10,282	1b6394be
ctripcorp/apollo	k6	1,104.5	20,593	6b0a06a4
davidmoten/rtree	k1	33.1	946	7c6f2001
DigitalPebble/storm-crawler	k8	27.8	634	43d05f80
dynjs/dynjs	k2	366.0	613	785b2039
electronicarts/ea-async	k5	13.9	870	96962312
EsotericSoftware/reflectasm	k4	3.4	1,066	7cab65bb
fengjiachun/Jupiter	k4, k10	44.0	1,149	2b16d7ab
galenframework/galen	k3	561.2	1,373	11e4053c
glowroot/glowroot	k18	139.1	972	a3a63617
google/closure-templates	k2	189.4	433	44e80ce9
hcoles/pitest	k13	67.7	1,364	9f838bdc
internetarchive/heritrix3	k9	84.9	1,710	83c7044b
intuit/wasabi	k6, k24	573.8	922	9f2aa5f9
jamesdbloom/mockserver	k23	74.8	3,593	aec1fbf1
junkdog/artemis-odb	k5	41.5	581	d03b4650
magro/kryo-serializers	k1	5.4	368	c6b544ea
NanoHttpd/nanohttpd	k22	6.8	6,267	e7935c56
querydsl/querydsl	k7, k11	686.1	2,784	8d6b7382
raphw/byte-buddy	k5	111.8	5,337	dcb89da2
rhuss/jolokia	k16	30.8	737	fffeeee7
stanford-futuredata/macrobase	k1	56.2	590	f729b896
stoicflame/enunciate	k3, k8	407.8	445	780bcf77
uber/uReplicator	k20, k21	7.8	817	629e3a9e
uber-common/jvm-profiler	k3	7.5	1,300	03a21825
x-stream/xstream	k14	59.3	637	807e21fb
zanata/zanata-platform	k2	16.9	308	c679fd0a

We experimented with the following matching strategies: S^*, disabling switches S1 and S2, enforcing exact match on argument context and primitive values; S^*_{poly}, enabling S1 and disabling S2; S^*_{prim}, disabling S1 and enabling S2; $S^*_{\text{poly,prim}}$, enabling both S1 and S2; and $S^*_{\neg\text{coevo}}$, same as S^* except using Eq. (1) to compute the confidence score, i.e., the calculation is solely based on the number of matched arguments, not utilizing co-evolution information. Note that all the strategies use Eq. (2) to compute the confidence score except $S^*_{\neg\text{coevo}}$. The strategies S^*, S^*_{poly}, S^*_{prim}, and $S^*_{\text{poly,prim}}$ apply different combinations of switches. $S^*_{\neg\text{coevo}}$ is designed to study whether the co-evolution information can help in matching (by comparing the curves of S^* and $S^*_{\neg\text{coevo}}$).

Table 3. Comparison of Matching Strategies: Precision, Recall and F_1 Scores under Different Strategies and Confidence Thresholds (The maximal F_1 scores in each row are colored in orange, the global maximal F_1 score is colored in blue).

Confidence Threshold	S^*			S^*_{poly}			S^*_{prim}			$S^*_{\text{poly,prim}}$			$S^*_{\neg\text{coevo}}$		
	PR	RC	F_1	PR	RC	F_1	PR	RC	F_1	PR	RC	F_1	PR	RC	F_1
0.0	0.51	1.00	0.675	0.51	1.00	0.675	0.51	1.00	0.675	0.51	1.00	0.675	0.51	1.00	0.675
0.1	0.62	1.00	0.765	0.61	1.00	0.758	0.53	1.00	0.693	0.53	1.00	0.693	0.70	0.42	0.525
0.2	0.62	1.00	0.765	0.62	1.00	0.765	0.53	1.00	0.693	0.53	1.00	0.693	0.70	0.42	0.525
0.3	0.62	1.00	0.765	0.62	1.00	0.765	0.53	1.00	0.693	0.53	1.00	0.693	0.70	0.42	0.525
0.4	0.62	1.00	0.765	0.62	1.00	0.765	0.53	1.00	0.693	0.53	1.00	0.693	0.70	0.40	0.509
0.5	0.62	1.00	0.765	0.62	1.00	0.765	0.53	1.00	0.693	0.53	1.00	0.693	0.70	0.40	0.509
0.6	0.70	0.42	0.525	0.70	0.57	0.628	0.76	0.83	0.793	0.75	0.97	0.846	0.96	0.21	0.345
0.7	0.70	0.40	0.509	0.74	0.55	0.631	0.77	0.82	0.794	0.78	0.96	0.861	1.00	0.12	0.214
0.8	0.96	0.21	0.345	0.90	0.37	0.524	0.87	0.53	0.659	0.87	0.68	0.763	1.00	0.12	0.214
0.9	1.00	0.12	0.214	0.90	0.27	0.415	0.88	0.48	0.621	0.87	0.62	0.724	1.00	0.12	0.214
1.0	1.00	0.12	0.214	0.90	0.27	0.415	0.88	0.48	0.621	0.87	0.62	0.724	1.00	0.12	0.214

Table 3 shows the precision, recall, and F_1 scores of context matching with different matching strategies. A higher F_1 scores indicates a better balance between the precision and recall. The precision, recall, and F_1 scores are computed according to formula Eqs. (3a) to (3c) respectively. If a call site is manually identified as unrevealable, but CompCheck matches it, we consider it as a false positive of matching; if a call site is manually identified as revealable, but CompCheck does not match it, then we consider it as a false negative of matching. From top to bottom in Table 3, each row shows the scores given a certain confidence threshold value (from 0.0 to 1.0, at the step of 0.1). The maximal F_1 scores in each row are colored in orange, while the global maximal F_1 score is colored in blue.

The result indicates that the best strategy is $S^*_{\text{poly,prim}}$ with the confidence threshold set to 0.7 (F_1 score = 0.861). It confirms our intuition that tolerating reasonable difference in context matching could improve the recall without much loss of the precision. We used it as our default matching strategy for the experiment for RQ3–RQ5. Another observation is that S^* generally performs better than $S^*_{\neg\text{coevo}}$, indicating the library co-evolution information can improve the effectiveness of context matching. For S^* and $S^*_{\neg\text{coevo}}$, as the threshold value increases from 0.0 to 1.0, recall keeps decreasing while precision keeps increasing. This is because as the threshold goes higher, fewer call sites can be matched, but each call site is matched with higher confidence. For the other strategies, since they all tolerate differences in the contexts, they may have false positives even when the confidence threshold is 1.0, thus can have precision scores lower than 1.00 at the bottom row.

Answer to RQ2: Matching strategies impact the effectiveness of context matching. On our dataset, with switches S1 and S2 enabled and confidence threshold set to 0.7, context matching achieves the best effectiveness (precision = 0.78, recall = 0.96). Library co-evolution information helps context matching.

6.2 Effectiveness of Incompatibility Discovery

To answer RQ3 and RQ4, we ran CompCheck's incompatibility discovery on 202 call sites collected from the clients in Table 2. For RQ3, we evaluated CompCheck's effectiveness based on how many call sites of incompatible APIs that it could reveal successfully. For RQ4, to show CompCheck's effectiveness compared with existing techniques, we performed an end-to-end comparison on

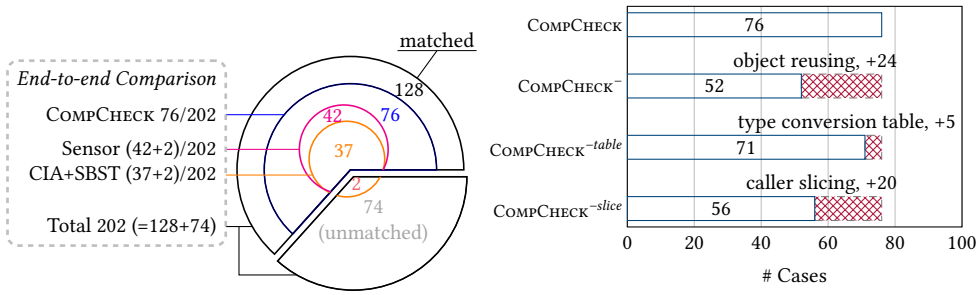
CompCheck and two other techniques: (1) Sensor [84], a state-of-the-art technique that detects dependency (library) conflicts by test generation guided by client code analysis, and (2) CIA+SBST, a technique that detects library incompatibility by combining change impact analysis with search-based test generation. In these experiments, we configured CompCheck to use the default strategy $S^*_{\text{poly,prim}}$ for knowledge matching. We give details of each experiment setup later in this section. The experiments were performed on a 4-core Intel(R) Core(TM) i7-8650 CPU @ 1.90 GHz machine with 16GB of RAM, running Ubuntu 16.04, with Java 1.8.0_151 and Python 3.7.3.

The end-to-end comparison experiment (RQ4) was conducted on CompCheck and two other techniques—Sensor and CIA+SBST. Our rationale for selecting these techniques for comparison is as follows. First, Sensor is the most recent state-of-the-art technique for detecting library incompatibilities (named dependency conflicts in [84]) in a Java project. From the technical perspective, both CompCheck and Sensor use program analysis to guide test generation, where the main difference is that when generating tests, CompCheck utilizes runtime input values recorded in the knowledge base, which are known to cause incompatibility in some other clients previously; while Sensor relies on the parameter values extracted from the client code under test. The implementation of Sensor is also publicly available. Second, the goal of *differential regression test generation* [16, 25] is very similar to CompCheck's. In fact, it can be used to reveal changes by generating a set of targeted tests. Yet, to the best of our knowledge, the only available tool implementation, EvoSuiteR [25] is not client-oriented: it is able to generate tests for changed (library) classes, but not for target clients relying on them. Therefore, for a fair comparison, we implemented an end-to-end compatibility checking technique on top of EvoSuite, based on the idea of EvoSuiteR and made it client-oriented with the help of change impact analysis (CIA) [74]. We name this combination CIA+SBST and use it as another baseline to compare with CompCheck.

Note that many other existing compatibility checking tools are not comparable with CompCheck by their nature. CompCheck is client-oriented and client-specific, i.e., it is designed for developers of client applications and only focuses on incompatibility issues which can be manifested at a given client context. Smart alerting [83] and Dependabot [19] are client-oriented but not client-specific. They warn client developers whenever there is a known severe bug in the new library version without providing concrete test cases, even if the buggy code is not used by the client. DeBBI [15] is library-oriented and aims to help library developers test their code more effectively with the support of additional test suites. GemChecker [17], AURA [88], and HiMa [57] aim to detect deprecated APIs in new library versions or discover API migration rules, rather than revealing behavioral incompatibility.

The experiment was performed on 202 call sites collected from the clients in Table 2. For CompCheck's call site matching, we used its default strategy $S^*_{\text{poly,prim}}$. For Sensor, we firstly modified the *pom.xml* files of each client project to include both the old and the new versions of the target library, then enforced Maven to load both library files, and manually ran Maven's debugging mode in each client to make sure both *.jar* files are present on the class path. For single-module Maven projects, we ran Sensor in the root directory; for multi-module Maven projects, we ran Sensor in both the root and the sub-module directories containing the call site of the target API and reported the combined results. For cases that Sensor detects conflict libraries but does not attempt to generate tests, we ran the standard EvoSuite (the test generator used by Sensor) on top of Sensor's detection result. If the standard EvoSuite generates any incompatibility-revealing tests, we consider it as a success of Sensor as well.

For CIA+SBST, we ran change impact analysis on both library and client at the Java bytecode level. Given an old version and a new version of the library, CIA+SBST uses Chianti's impact analysis algorithm to identify the impacted code entities of the client, obtaining a list of public



(a) Effectiveness of incompatibility discovery: CompCheck, Sensor, and CIA+SBST.

(b) The effectiveness contributions of CompCheck's object reusing and optimizations.

Fig. 9. Effectiveness of incompatibility discovery.

methods affected by the changes of the library. Next, the list is passed to the search-based test generator (i.e., EvoSuite), which in turn uses these methods as test generation targets and finally outputs of a set of generated tests.

To compare the tools' effectiveness, the tests generated by all three tools were executed on the old and new library versions separately, and we compared their numbers of generated incompat ibility-revealing tests, which pass with the old version but fail with the new version. The more incompatibility issues a tool can reveal with its generated incompatibility-revealing tests, the more effective it is. We set a time budget of 30 min for each target call site for all the tools. To avoid fluctuations caused by randomness, we ran each tool 10 times and reported the average.

Figure 9a shows the results of the experiment. By knowledge matching, CompCheck matches 128 call sites as targets, of which it successfully generated incompatibility-revealing tests for 76 call sites. In contrast, Sensor was able to detect incompatibility on 44 call sites in total. CIA+SBST treats 202 call sites as targets because all these call sites are affected by the library changes, but it managed to generate incompatibility-revealing tests for only 39 of them. Recall that for incompatibility discovery, an incompatibility issue is considered successfully discovered by a tool if the tool can generate an incompatibility-revealing test on the call site of the incompatible API. Therefore, in this experiment, CompCheck successfully discovered 76 incompatibility issues, while Sensor and CIA+SBST each discovered 44 and 39 issues, respectively. The result indicates that CompCheck is more effective than both Sensor and CIA+SBST.

Figure 10 shows an example client method where CompCheck outperforms Sensor in generating incompatibility-revealing tests. The code snippet is from the XStream [89] project. In this example, the incompatible API is "*DateTimeFormatter.parseDateTime*" from the Joda-Time [45] library (Line 5), which accepts two input variables: "*formatter*" and "*str*". The incompatibility issue is that when Joda-Time was upgraded to the new version, the parsing logic of certain date values was changed, thus can cause exceptions in its clients [20]. To trigger the incompatible behavior, the "str" variable must be of specific values (the string representations of certain dates). Thus, randomly-generated strings can hardly trigger it. In this case, CompCheck was able to generate incompat ibility-revealing tests because it directly reused the string values mined from the test execution of Amazon's AWS Java SDK [8] in its knowledge mining phase—"*292278994-08-17T07:12:55.807Z*", which was also confirmed by the AWS developers in the release notes [9]. However, Sensor was not able to generate incompatibility-revealing tests, as its *class instance pool* [84] only contains the objects and primitives from the project under test (XStream in this case). However, no relevant string value, which triggers this incompatibility issue, appears in XStream's code base. As a result,

```
1 public Object fromString(final String str) {
2     for (int i = 0; i < formattersUTC.length ; ++i) {
3         final DateTimeFormatter formatter = formattersUTC[i];
4         try {
5             final DateTime dt = formatter.parseDateTime(str);
6             final Calendar calendar = dt.toGregorianCalendar();
7             calendar.setTimeZone(TimeZone.getDefault());
8             return calendar;
9         } catch (final IllegalArgumentException e) {
10            ...
11        }
12    }
13    ...
14 }
```

Fig. 10. An example where CompCheck outperforms Sensor in generating incompatibility-revealing tests.

Sensor failed to find useful seeds for its test generation, thus could not reveal this incompatibility issue. This example demonstrates the benefit of CompCheck's knowledge-based approach. With the help of past knowledge mined from other projects, it can discover new incompatibility issues in unseen clients.

Out of the 76 incompatibility issues revealed by CompCheck, 44 were manifested as direct incompatibility, 13 were manifested as transitive incompatibility, and 19 were manifested as co-evolution incompatibility. This indicates that CompCheck is able to reveal incompatibility issues of all the manifestation patterns. In total, 297 test methods were generated by CompCheck. Among these tests, 103 are incompatibility-revealing tests and 194 are non-incompatibility-revealing tests. CompCheck utilizes EvoSuite as its underlying test generator. In general, EvoSuite generates one test class per call site, where each test class can contain multiple test methods. The number of test methods in a test class varies. Each test method corresponds to one client calling context of the target API.

There are 52 call sites where CompCheck matched but was not able to generate incompatibility-revealing tests. We inspected them, found that a common reason is that they have arguments that are too complex to generate within the time bound. For example, Fig. 11 shows a client method in the project Heritrix [5]. The incompatible API is *Base64.decodeBase64* (Line 12). The client (caller) method *PersistProcessor.populatePersistEnvFromLog* accepts as input a *BufferedReader* object. This *BufferedReader* object is supposed to be initialized from a text file with each line separated by a space. The method then extracts certain contents from the file, processes them, and passes them to the target API. It is almost impossible to construct such a specific *BufferedReader* by searching. Furthermore, CompCheck's caller slicing cannot be applied to the *BufferedReader* argument here, as the target API call site has data dependency on it. On these call sites, not only CompCheck, but neither Sensor nor CIA+SBST was able to generate incompatibility-revealing tests.

Answer to RQ3: CompCheck is effective in discovering incompatibility issues. Among the 104 manually labeled revealable call sites, CompCheck successfully revealed incompatibility issues on 76 (73.1%). Every incompatible call site revealed by CompCheck is true positive and is validated by a generated incompatibility-revealing test.

```
1 public static int populatePersistEnvFromLog(BufferedReader persistLogReader,
      StoredSortedMap<String,Map> historyMap) throws
      UnsupportedEncodingException, DatabaseException {
2       int count = 0;
3       Iterator<String> iter = new LineReadingIterator(persistLogReader);
4       while (iter.hasNext()) {
5           String line = iter.next();
6           if (line.length() == 0) {
7               continue;
8           }
9           String[] splits = line.split(" ");
10          ...
11          alist = (Map) SerializationUtils.
12              deserialize(Base64.decodeBase64(splits[1].getBytes("UTF-8")));
13          ...
14      }
15      ...
16 }
```

Fig. 11. An example that CompCheck failed to generate incompatibility-revealing test.

Answer to RQ4: CompCheck is more effective in discovering incompatibility issues than existing techniques. It revealed incompatibility issues on 76 call sites, 72.7% more than Sensor (revealed 44) and 94.9% more than CIA+SBST (revealed 39), respectively.

6.3 Effectiveness Contributions of Technical Components

For RQ5, to measure the improvement brought by object reusing, we compared the effectiveness of CompCheck with its variant that has this feature disabled; to measure the improvement brought by the optimizations (i.e., type conversion table and caller slicing), we compared the effectiveness of CompCheck with its variant that disables the optimizations. Same as Section 6.2, in these experiments, we used the default strategy $S^{*}_{\text{poly,prim}}$ for CompCheck's knowledge matching. We give details of each experiment setup later in this section. The experiments were performed in the same environment as Section 6.2.

To evaluate the benefit of object reusing in generating incompatibility-revealing tests, we compared CompCheck with its variant that only disables object reusing (named CompCheck^{-}). From an implementation perspective, CompCheck^{-} is equivalent to running EvoSuite on the knowledge-matched target call sites. The experiment setup is similar to the previous one. As shown in Fig. 9b, out of the 128 matched call sites, CompCheck^{-} managed to generate incompatibility-revealing tests on 52 while CompCheck succeeded on 76. This indicates that object reusing can significantly help improve the effectiveness of test generation (24 more issues revealed). As an example, the incompatibility issue of k2 can be exposed only when it takes as input an object with final fields annotated with the "*@Option*" annotation (an annotation class in Args4j). For plain search-based test generation (used by CompCheck^{-}), such an object is extremely hard to create. However, with object reusing, CompCheck can reuse a stored object from the knowledge base, cutting the search space of the object and achieving significant saving. Note that here we compare CompCheck with CompCheck^{-} on 128 call sites because improvement is only possible when the knowledge is

matched. For other call sites, CompCheck behaves exactly the same as CompCheck^{-}, making the comparison meaningless.

To measure the benefit of CompCheck's optimizations in incompatibility discovery, we compared CompCheck with its two variants, CompCheck^{-table} and CompCheck^{-slice}, each disabling one optimization—type conversion table and caller slicing, respectively (both optimizations are enabled by default in CompCheck). As shown in Fig. 9b, both variants perform less effective than CompCheck, which confirms that both optimizations improve the effectiveness of incompatibility discovery. Type conversion table and caller slicing each helped solve 5 and 20 cases, respectively.

Answer to RQ5: CompCheck's object reusing and optimizations significantly contribute to its overall effectiveness. 24 of the 76 revealed incompatibility issues were discovered with the help of object reusing. The two optimizations, type conversion table and caller slicing, helped discover 5 and 20 issues, respectively.

6.4 Threats to Validity

Our evaluation is subject to the following threats to validity.

External. Projects that we used for the experiments may not be representative of all software projects. To mitigate this threat, we chose popular open-source projects from GitHub that use Maven—a widely used project management tool for Java—as their build system. Our current implementation of CompCheck supports only Java, but our methodology and workflow can apply to any programming language. However, for weakly typed or untyped languages, such as Python and JavaScript, the matching strategy should be redesigned as we are not able to use type information to control the matching sensitivity.

The knowledge base used may not be representative. We limited our experiments with a knowledge base built at the latest versions of the libraries at the time of the study. For the library pairs, the old ones are usually more than a year old and the new ones are the most recent release (median age is 5.5 months) at the time of experimenting. The age of the clients does not matter for knowledge mining, since we upgrade their libraries automatically. We focused on high-starred projects on GitHub, as they generally have high test coverage and representative usages of libraries. By considering the latest version of the libraries, we simulate the scenario where knowledge is mined right after a library is released. In a realistic setting, e.g., deploying CompCheck on a continuous integration environment, one would monitor the releases of the libraries of interest and perform knowledge mining after every new release, resulting in a much richer knowledge base over time.

Internal. When manually labeling the call sites of incompatible APIs for the matching strategy experiment, the call sites labeled as revealable are guaranteed to be revealable, as we could create tests for them; but for those call sites labeled as unrevealable, we cannot guarantee it is unrevealable under all possible inputs, because we could not enumerate all input combinations. Thus, the mislabeling of some revealable call sites as unrevealable is a possible threat. As a result, when calculating the precision and recall values, our computation may be an under-approximation for the precision and an over-approximation for the recall. In our experiment, on all the call sites that are manually labeled as unrevealable, none of the experimented tools was able to generate an incompatibility-revealing test. Therefore, we did not observe mislabeled call sites in practice.

Test flakiness may affect the soundness of CompCheck. In knowledge mining, if a client test fails intermittently, CompCheck may treat it as an incompatibility failure by mistake. To mitigate this threat, in the module-level regression testing phase, for each client, the initial clean test run without upgrading any library was executed three times, and CompCheck accepts the project only if the test suite passes in all three runs. In this way, we can alleviate the harm of "shallow" flaky

tests that are easy to expose. For incompatibility discovery, CompCheck uses EvoSuite as the test generator, which avoids generating flaky tests by design [26]. Besides, we manually inspected all the incompatibility-revealing tests generated by CompCheck and confirmed all the failures were indeed caused by incompatibility instead of flakiness.

CompCheck implementation or any scripts we wrote to run experiments may contain bugs. To mitigate this threat, we reviewed code thoroughly and wrote unit tests.

7 RELATED WORK

We populate the related work along the following axes.

API Incompatibility. Mostafa et al. [62] studied 126 real-world bug reports related to behavioral incompatibilities of Java libraries. The authors reported that, unhandled exceptions and GUI changes are the most common incompatible behaviors, and most client software bugs caused by backward incompatibility of libraries are fixed with simple changes to the code around the backward incompatible API invocation. Table 4 summarizes and compares the key techniques from the API incompatibility detection literature.

Wang et al. [84] studied *dependency conflicts*, i.e., the library APIs referenced by a project have identical method signature but inconsistent semantics across the loaded and the shadowed versions of the libraries. The definition of dependency conflict is similar to that of ours on upgrade incompatibility. But the reasons behind the semantic inconsistencies are different—the former is caused by dependency shadowing and the latter is because of library version upgrades. Their empirical findings also indicate that it is rather challenging to expose the semantic inconsistencies by randomly generated input values to the clients, especially when class instances are required. Therefore, they proposed to construct a *class instance pool* with pregenerated class instances instantiated with parameters found statically from source code at all invocation contexts, which will then be used as seeds for EvoSuite in test generation. We faced the same issue in test generation, and our solution is to rely on runtime input values recorded in the knowledge base, which are known to cause incompatibilities, and avoid the need to perform random searching.

Foo et al. [30] suggested a light-weight static analysis to determine if an upgrade to library packages may introduce an API incompatibility. This is done by checking if any deleted or changed library method is still used by the client. Various optimizations, such as bytecode hashing, have been implemented to ensure the queries are finished quickly so that the compatibility checking can be integrated into a CI/CD pipeline. Yet, there might be imprecision in such an analysis, because using a changed library method does not imply that the client's behavior will be affected. Chen et al. [15] proposed DeBBI, which uses a large number of test suites of open-source client projects to test the incompatibility of library upgrades. To reduce the testing cost of clients, DeBBI prioritizes client test suites to expose issues earlier via information retrieval and test selection. Different from these techniques, CompCheck is client-specific. It is not designed for library developers who have no knowledge of client implementations, but is supposed to be used by users of the library for checking the compatibility of their specific clients.

Apart from Java, similar API incompatibility issues have also been studied for other ecosystems, such as JavaScript [58, 59, 63], Python [30, 93], Ruby [30], Android [52], and REST APIs [36]. Mezzetti et al. [58] presented NoRegrets, a type regression testing technique to detect type-related breaking changes in Node.js libraries. Moller et al. [59] proposed NoRegrets+, a model-based variant of NoRegrets, improving the scalability. Given a single execution of the client tests, NoRegrets+ generates an API model to summarize multiple subsequent updates to the library and utilizes the model to dynamically explore the library and check the type consistency of the values passing between the client and library. Godefroid et al. [36] proposed differential regression

Table 4. A comparative summary of the API incompatibility detection literature ("C.S." stands for "Client-Specific").

Techniques	Incompatibility Definitions	Studied Ecosystems	C.S.?
CompCheck	Inconsistent library method semantics captured by client tests	Java	Yes
Wang et al. [84]	Inconsistent library method semantics captured by client tests	Java	Yes
Chen et al. [15]	Inconsistent library method semantics captured by client tests	Java	No
Foo et al. [30]	Missing or changed library methods	Java, Python, Ruby	No
Mostafa et al. [62]	Behavioral incompatibilities reported as bugs	Java	No
Li et al. [52]	API Methods accessed without proper Android API-level protections	Android	Yes
Mezzetti et al. [58]	Changed type signatures of public API interfaces	JavaScript	No
Godefroid et al. [36]	Previous valid requests are no longer accepted or return wrong results	REST API	Yes

testing for REST APIs, which discovers REST API behavioral differences by feeding generated HTTP requests to cloud services and comparing the output HTTP responses. Mujahid et al. [63] studied the upgrade incompatibility issues in the npm ecosystem. They proposed to rely on test cases from multiple dependent projects collectively, to detect breakage-inducing library versions, a.k.a. upgrade incompatibilities. They have shown that using crowd-sourced tests could increase the overall test coverage, and thus improve the capacity of incompatibility detection. Their definition of incompatibility is similar to ours, i.e., both using crowd-sourced tests as an indicator, but they do not consider the client-specificity of library usages.

API Usage Patterns and Misuses. There is usually insufficient documentation on how to correctly use library APIs. Library *API usage patterns* can be useful resources for client developers when facing such a challenge. Many studies have been done to mine API-related usage patterns [64, 79–81, 94]. For example, Saied et al. [79] proposed a multi-level API usage mining technique which detects groups of API methods that are highly cohesive in terms of usage by client programs. The detected patterns are constructed in a hierarchical manner and are useful to enrich the API's documentation. Later, Saied et al. [80] introduced an algorithm, GenLTL, to recover API usage patterns as Linear Temporal Logic (LTL) expressions. The LTL expressions can then be translated into natural language documentation to assist developers in safely using the multiple APIs necessary for their development tasks.

A closely related topic is to recognize *API misuses* [85, 86], where implicit usage constraints (patterns) are violated. For instance, Zhang et al. [92] performed a study on the API misuses in practice. They verified the reliability of code examples found on the Stack Overflow Q&A forum by examining whether the code samples provided in the most voted answers match the context-enriched call sequence patterns mined from a wide range of usage examples on GitHub. Amann et al. constructed a standard benchmark, MUBench, for real-world API misuses, which contains 73 misuses from software projects and 17 from the survey on developers. They also systematically evaluated various API-misuse detectors on it [2].

The upgrade incompatibility issues that we study in this paper, are not necessarily caused by API misuses. Indeed, some misuses may be induced by library upgrades, when a client usage no

longer complies with the new API-usage constraints. Yet, other incompatibilities are manifested as behavior differences at the client side, even when the usages are correct, e.g., see *Client1* in Fig. 1. CompCheck is able to capture both types of incompatibility issues, as long as they have been previously recorded in the knowledge base.

Program Equivalence Checking. Equivalence verification [11, 29, 37, 51, 61, 67] and compatibility checking [15, 58, 59] are two major approaches for checking behavioral incompatibility of library upgrade. For equivalence verification, regression verification [29, 37] aims to formally prove that two programs are functionally equivalent. An example of such techniques is RVT [37], which proves the partial equivalence of two related programs, i.e., producing the same outputs for all inputs given that they both terminate, based on a set of proof rules. On the other hand, differential symbolic execution (DSE) [67, 68] performs standard symbolic execution on both program versions before and after the change, either reports the two versions equivalent or characterizes the behavioral differences by identifying the sets of inputs causing different effects. Different from RVT and DSE, which defines equivalence over the entire program without separating clients and libraries, Clever [61] performs client-specific equivalence checking, which considers the effect of client context on library changes. The equivalence verification problem, in its most general form, is shown to be intractable [14]—there is no known complete solution to it which can tackle real software packages, especially ones of large sizes and with complex language constructs, often witnessed in real-world development scenarios.

Program Analysis Guided Test Generation. Automated test generation [16, 33] is developed to automatically generate test cases to discover program defects. Program analysis techniques are actively employed for guiding test generation. Jaygarl et al. [44] proposed an object capture-based automated testing technique, which uses dynamically captured object instances from program executions to improve code coverage of existing test generation tools. Zhang et al. [91] used dynamic analysis to extract information for guiding legal sequence generation and use static analysis to improve the diversity of the generated test cases. Babić et al. [10] used dynamic analysis to revolve around a visibly pushdown automaton (VPA) for representing the program's global control flow structure and used static analysis to assist symbolic execution to explore vulnerabilities based on the shortest paths and loop pattern heuristics. Ma et al. [54] propose GRT, a guided random testing technique which uses static analysis to extract information on program types, data, and dependencies to guide run-time test generation. Kechagia et al. [47] proposed Catcher, a tool that utilizes static exception propagation analysis to guide search-based test case generation, in order to effectively pinpoint crash-prone API misuses.

To the best of our knowledge, CompCheck is the first that combines dynamic analysis of previously mined clients, static analysis of new target clients, and search-based test generation, focusing on revealing client-specific library upgrade incompatibility issues.

Code-to-Code Search. Given a simple code snippet as a query, code-to-code search techniques aim to search for code similar to the query snippet. Bajracharya et al. [12] proposed Sourcerer, which takes a complete compilation unit as input and extracts structural code information for fine-grained code search. McMillan et al. [56] propose Portfolio to extract function invocation chain to help developers search and understand code usage. Luan et al. [53] propose Aroma, a structural code search tool, which takes a code snippet as input, filters out similar function bodies from corpus and clusters and intersects them to provide developers representative code snippets. Kim et al. [48] propose FaCoY, a code-to-code search engine which takes a code snippet as query, searches in a database to find relevant natural language descriptions and recommends related posts and code. These approaches are similar to the context matching of CompCheck in terms of the goal.

However, instead of using code snippet or a query, CompCheck leverages consolidated execution information, e.g., argument traces, to model and match the API usage context.

8 CONCLUSION

Modern complex software systems heavily rely on libraries developed by different teams and organizations and the systems suffer from higher external vulnerability due to the incompatibility issues caused by library upgrade. In this paper, we studied the impact of library upgrade on the dependent software and proposed CompCheck, an automated compatibility checking technique for specific clients. CompCheck constructs an offline knowledge base summarizing previous recognized incompatibility issues on a large scale program corpus. During the online incompatibility discovery, given a new client, CompCheck matches it with context summary in the knowledge base with respect to the usage of target API, and uses the stored execution data to guide the generation of incompatibility-revealing tests for it. We evaluated our technique on 202 call sites from 37 open-source projects, the results show that CompCheck successfully generate incompatibility-revealing tests for 76 call sites, 72.7% and 94.9% more than two existing techniques. The evaluation results show that CompCheck is effective in generating incompatibility-revealing tests, which can be used to assist developers in a smooth upgrade of library dependencies.

REFERENCES

[1] Alibaba. 2022. Fastjson. https://github.com/alibaba/fastjson
[2] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.
[3] Apache. 2022. Apache HttpComponents. https://hc.apache.org
[4] Apache Software Foundation. 2020. Apache Maven Project. https://maven.apache.org
[5] Internet Archive. 2020. Heritrix. https://github.com/internetarchive/heritrix3
[6] Artemis-odb. 2020. Artemis-odb. https://github.com/junkdog/artemis-odb.
[7] ASM Developers. 2020. ASM. https://asm.ow2.io
[8] AWS. 2022. AWS SDK for Java. https://github.com/aws/aws-sdk-java
[9] AWS. 2022. Release: AWS SDK for Java 1.8.0. https://aws.amazon.com/releasenotes/release-aws-sdk-for-java-1-8-0
[10] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-Directed Dynamic Automated Test Generation. In *International Symposium on Software Testing and Analysis*. 12–22.
[11] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: Scaling Program Equivalence Checking via Iterative Abstraction and Refinement of Common Code. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 13–24.
[12] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search. In *Companion to the ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. 681–682.
[13] Leyla Bilge and Tudor Dumitraş. 2012. Before We Knew It: An Empirical Study of Zero-Day Attacks In The Real World. In *The ACM Conference on Computer and Communications Security*. 833–844.
[14] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. 2012. Regression Verification for Multi-Threaded Programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. 119–135.
[15] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis. In *International Conference on Software Engineering*. 112–124.
[16] Yanping Chen, Robert L Probert, and Hasan Ural. 2007. Model-Based Regression Test Suite Generation Using Dependence Analysis. In *International Workshop on Advances in Model Based Testing*. 54–62.
[17] Jamie Cleare and Claudia Iacob. 2018. GemChecker: Reporting on the Status of Gems in Ruby on Rails Projects. In *International Conference on Software Maintenance and Evolution*. 700–704.
[18] CompCheck. 2020. CompCheck. https://sites.google.com/view/compcheck
[19] Dependabot. 2020. Dependabot. https://dependabot.com.
[20] AWS SDK Developers. 2022. New JodaTime-based DateUtils can't deserialize some dates saved in DynamoDB by an older version of the SDK. https://github.com/aws/aws-sdk-java/issues/233
[21] Kryo Developers. 2015. Line #125 of Kryo.java at version 3.0.3. https://github.com/EsotericSoftware/kryo/blob/9422c847db584dcddfa614303cd41d57eb76220f/src/com/esotericsoftware/kryo/Kryo.java#L125.

[22] Kryo Developers. 2019. Line #141 of Kryo.java at version 5.0.0-RC4. https://github.com/EsotericSoftware/kryo/blob/d729517080d5a6037e6fd4bb953936106228ab43/src/com/esotericsoftware/kryo/Kryo.java#L141.
[23] Dromara. 2020. Myth. https://github.com/dromara/myth.
[24] EsotericSoftware. 2020. Kryo. https://github.com/EsotericSoftware/kryo.
[25] EvoSuite. 2020. Differential / Regression Test Generation. https://www.evosuite.org/evosuiter/.
[26] EvoSuite. 2020. *EvoSuite Documentation*. https://www.evosuite.org/documentation/tutorial-part-1.
[27] FasterXML, LLC. 2020. jackson-core. https://github.com/FasterXML/jackson-core
[28] FasterXML, LLC. 2020. jackson-databind. https://github.com/FasterXML/jackson-databind
[29] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *International Conference on Automated Software Engineering*. 349–360.
[30] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient Static Checking of Library Updates. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 791–796.
[31] Apache Software Foundation. 2019. Apache Commons Codec. https://commons.apache.org/proper/commons-codec
[32] Apache Software Foundation. 2020. Apache HttpClient. https://github.com/apache/httpcomponents-client
[33] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 416–419.
[34] GitHub. 2020. GitHub. https://github.com.
[35] GitHub. 2020. Jackson-databind version X should depend on jackson-annotations version X. https://github.com/FasterXML/jackson-databind/issues/135.
[36] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential Regression Testing for REST APIs. In *International Symposium on Software Testing and Analysis*. 312–323.
[37] Benny Godlin and Ofer Strichman. 2013. Regression Verification: Proving the Equivalence of Similar Programs. *Software Testing, Verification and Reliability* 23, 3 (2013), 241–258.
[38] Google. 2020. Guice. https://github.com/google/guice
[39] Sable Research Group. 2022. Soot. https://github.com/Sable/soot
[40] Kafka Upgrade Guide. 2020. Kafka Upgrade Guide. https://kafka.apache.org/20/documentation/streams/upgrade-guide.
[41] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. In *International Symposium on Software Reliability Engineering*. 112–122.
[42] HtmlCleaner. 2022. HtmlCleaner. http://htmlcleaner.sourceforge.net
[43] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450.
[44] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. 2010. OCAT: Object Capture-Based Automated Testing. In *International Symposium on Software Testing and Analysis*. 159–170.
[45] Joda-Time. 2022. Joda-Time. https://www.joda.org/joda-time
[46] Kohsuke Kawaguchi. 2020. Args4j. https://github.com/kohsuke/args4j
[47] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and Efficient API Misuse Detection via Exception Propagation and Search-Based Testing. In *International Symposium on Software Testing and Analysis*. 192–203.
[48] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: A Code-to-Code Search Engine. In *International Conference on Software Engineering*. 946–957.
[49] Kryo. 2019. Migration to v5. https://github.com/EsotericSoftware/kryo/wiki/Migration-to-v5
[50] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do Developers Update Their Library Dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.
[51] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential Assertion Checking. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 345–355.
[52] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-Related Compatibility Issues in Android Apps. In *International Symposium on Software Testing and Analysis*. 153–163.
[53] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code Recommendation via Structural Code Search. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 1–28.
[54] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-Analysis-Guided Random Testing. In *International Conference on Automated Software Engineering*. 212–223.
[55] Maven Central Repository 2020. Maven Central Repository. https://mvnrepository.com/repos/central.

[56] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding Relevant Functions and Their Usage. In *International Conference on Software Engineering*. 111–120.
[57] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. 2012. A History-Based Matching Approach to Identification of Framework Evolution. In *International Conference on Software Engineering*. 353–363.
[58] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *European Conference on Object-Oriented Programming*. 7:1–7:24.
[59] Anders Møller and Martin Toldam Torp. 2019. Model-based Testing of Breaking Changes in Node.js Libraries. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 409–419.
[60] MongoDB. 2022. MongoDB Java Drivers. https://mongodb.github.io/mongo-java-driver
[61] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-Specific Equivalence Checking. In *International Conference on Automated Software Engineering*. 441–451.
[62] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. A Study on Behavioral Backward Incompatibility Bugs in Java Software Libraries. In *International Conference on Software Engineering Companion*. 127–129.
[63] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2020. Using Others' Tests to Identify Breaking Updates. In *International Conference on Mining Software Repositories*. 466–476.
[64] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-Based Mining of In-the-Wild, Fine-Grained, Semantic Code Change Patterns. In *International Conference on Software Engineering*. 819–830.
[65] José Oncina and Pedro Garcia. 1992. Inferring Regular Languages in Polynomial Updated Time. In *Pattern Recognition and Image Analysis: Selected Papers from the IVth Spanish Symposium*. 49–61.
[66] Jan Ouwens. 2022. EqualsVerifier. https://jqno.nl/equalsverifier
[67] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential Symbolic Execution. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 226–237.
[68] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed Incremental Symbolic Execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 504–515.
[69] Versions Maven Plugin. 2022. Versions Maven Plugin. https://www.mojohaus.org/versions-maven-plugin.
[70] JVM Profiler. 2020. Uber JVM Profiler. https://github.com/uber-common/jvm-profiler
[71] Protostuff. 2020. protostuff-runtime. https://github.com/protostuff/protostuff/tree/master/protostuff-runtime
[72] Querydsl. 2022. Unified Queries for Java. http://querydsl.com
[73] Reflections. 2020. Reflections. https://github.com/ronmamo/reflections
[74] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 432–448.
[75] Restlet. 2022. Restlet Framework. https://restlet.talend.com
[76] Rtree. 2020. Rtree. https://github.com/davidmoten/rtree.
[77] Julia Rubin and Martin Rinard. 2016. The Challenges of Staying Together While Moving Fast: An Exploratory Study. In *International Conference on Software Engineering*. 982–993.
[78] Rxjava-extras. 2020. Rxjava-extras. https://github.com/davidmoten/rxjava-extras.
[79] Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. 2015. Mining Multi-level API Usage Patterns. In *International Conference on Software Analysis, Evolution and Reengineering*. 23–32.
[80] Mohamed Aymen Saied, Erick Raelijohn, Edouard Batot, Michalis Famelis, and Houari Sahraoui. 2020. Towards Assisting Developers in API Usage by Automated Recovery of Complex Temporal Patterns. *Journal of Information and Software Technology* 119 (2020), 106213.
[81] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. 2015. An Observational Study on API Usage Constraints and Their Documentation. In *International Conference on Software Analysis, Evolution and Reengineering*. 33–42.
[82] SLF4J. 2022. Simple Logging Facade for Java (SLF4J). https://www.slf4j.org
[83] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *International Conference on Software Maintenance and Evolution*. 35–45.
[84] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhi-liang Zhu. 2021. Will Dependency Conflicts Affect My Program's Semantics? *IEEE Transactions on Software Engineering* 48, 7 (2021), 2295–2316.
[85] Andrzej Wasylkowski and Andreas Zeller. 2011. Mining Temporal Specifications from Object Usage. *Automated Software Engineering* 18, 3 (2011), 263–292.

[86] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 35–44.
[87] WebMagic. 2022. WebMagic. https://webmagic.io/en
[88] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. AURA: A Hybrid Approach to Identify Framework Evolution. In *International Conference on Software Engineering*. 325–334.
[89] XStream Developers. 2020. XStream. https://x-stream.github.io
[90] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.
[91] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *International Symposium on Software Testing and Analysis*. 353–363.
[92] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are Code Examples on an Online Q&A Forum Reliable?: A Study of API Misuse on Stack Overflow. In *International Conference on Software Engineering*. 886–896.
[93] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How Do Python Framework APIs Evolve? An Exploratory Study. In *International Conference on Software Analysis, Evolution and Reengineering*. 81–92.
[94] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *European Conference on Object-Oriented Programming*. 318–343.