Hidden Truths in Dead Software Paths

Michael Eichberg, Ben Hermann, Mira Mezini, Leonid Glanz
Technische Universität Darmstadt, Germany
{eichberg,hermann,mezini,glanz}@cs.tu-darmstadt.de

ABSTRACT

Approaches and techniques for statically finding a multitude of issues in source code have been developed in the past. A core property of these approaches is that they are usually targeted towards finding only a very specific kind of issue and that the effort to develop such an analysis is significant. This strictly limits the number of kinds of issues that can be detected.

In this paper, we discuss a generic approach based on the detection of infeasible paths in code that can discover a wide range of code smells ranging from useless code that hinders comprehension to real bugs. Code issues are identified by calculating the difference between the control-flow graph that contains all technically possible edges and the corresponding graph recorded while performing a more precise analysis using abstract interpretation.

We have evaluated the approach using the Java Development Kit as well as the Qualitas Corpus (a curated collection of over 100 Java Applications) and were able to find thousands of issues across a wide range of categories.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Java, Finding Bugs, Scalable Analysis, Abstract Interpretation, Infeasible Paths

1. INTRODUCTION

Since the 1970s many approaches have been developed that use static analyses to identify a multitude of different types of issues in source code [9, 3, 15, 11]. These approaches use very different techniques that range from pattern matching [9] to using formal methods [11] and have very different properties w.r.t. their precision and scalability. But, they all have in common that they are each targeting a very specific kind of issues. Those tools (e.g., FindBugs [9]) that can identify issues across a wide(r) range of issues are typically suits of relatively independent analyses. This limits the number of issues that can be found to those that are identified as relevant by the respective researchers and developers.

In this paper, we present a generic approach that can detect a whole set of control- and data-flow dependent issues in Java bytecode without actually targeting any specific kind of issues per se. The approach applies abstract interpretation to analyze the code as precisely as possible and while doing so records the paths that are taken. Afterwards, the analysis compares the recorded paths with the set of all paths that could be taken according to a naïve control-flow analysis that does not consider any data-flows. The paths computed by the latter analysis but not taken by the former analysis are then reported along with a justification why they were not taken.

The rationale underlying this approach is as follows. Many issues such as null dereferences, array index out of bounds accesses or failing class casts result in exceptions that leave infeasible paths behind. Hence, the hypothesis underlying the approach is threefold. First, in well-written code every path between an instruction and all it's direct successors is eventually taken, and, second, a path that will never be taken indicates an issue. Third, a large class of relevant issues manifests itself sooner or later in infeasible paths.

To validate the hypotheses we conducted a large-scale case study analyzing the Java Development Kit (JDK 1.8.0_25). Additionally, we did a brief evaluation of the approach using the Qualitas Corpus[22] to validate the conclusion drawn from the case study. The issues that we found range from seemingly benign issues to serious bugs that will lead to exceptions at runtime or to dead features. However, even at first sight benign issues, such as unnecessary checks that test what is already guaranteed, can have a significant impact. Manual code reviews are a common practice and the biggest issue in code reviews is comprehending the code [5]. A condition that always evaluates to the same value typically violates a reviewer's expectation and hence impedes comprehension causing real costs.

The analysis was explicitly designed to avoid false positives. And, yet it identified a large number of dead paths ($\approx 50\%$ of all identified dead paths), which do not relate to issues in the source code. This is because the analysis operates on byte code and is as such sensitive to compilation techniques and a few other intricacies of the language. To make the tool useable [16, 7], we implemented and evaluated three heuristics to filter irrelevant issues.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy

http://dx.doi.org/10.1145/2786805.2786865

Though we opted for analyzing the code as precisely as possible, we deliberately limited the scope of the analysis by starting with each method of a project then performing a context-sensitive analysis with a very small maximum call chain size. This makes the analysis unsound – i.e. we may miss certain issues – but it enables us to use it for large industrial sized libraries and applications such as the JDK. As the evaluation shows, the approach is effective and can identify a wide range of different issues while suppressing over 99% of all irrelevant findings.[1]

To summarize, we make the following contributions:

- We put forward the idea that infeasible paths in software are a good indication for issues in code and show that a large class of relevant issues does manifest itself sooner or later in infeasible paths.
- We present a new static analysis technique that exploits abstract interpretation to detect infeasible paths. The analysis is parametrized over abstract domains and the depth of call chains to follow inter-procedurally. These two features enable us to make informed reasonable trade-offs between scalability and soundness. Furthermore, the analysis has an extremely low rate of false positives.
- We validate the claims about the proposal in a case study of industrial size software; the issues revealed during the case study constitute themselves a valuable contribution of the paper.

The paper is structured as follows. In Section 2 we discuss a selection of issues that we were able to find. After that, we discuss the approach in Section 3 along with its implementation. The evaluation is then presented in Section 4. The paper ends with a discussion of related work in Section 5 and a conclusion thereafter.

2. CLASSIFICATION OF ISSUES

In this section, we discuss the kind of issues that we found by applying our analysis to the JDK 1.8.0_25[2]. Given that the approach is not targeted towards finding any specific category of issues, it is very likely that further categories emerge from investigating the results of applying the analysis to other projects. Yet, the following list already demonstrates the breadth of the applicability of the proposed approach.

Obviously Useless Code.

In some cases, we were surprised to find code that is obviously useless in such a mature library as JDK.

For illustration consider the code in Listing 1. In the method `isInitValue`, the developer checks whether an int variable contains a value that is smaller/larger than the smallest/largest possible value, which is obviously `false` and does not need to be checked for. Such useless code has two problems. First, it wastes resources. More importantly, it negatively impacts reasoning from the perspective of code that uses `isInitValue`. A developer has to understand that a call to `isInitValue` always returns `true`. This is most likely not obvious in the context of the calling method, as `isInitValue` suggest something different.

```
252  boolean isInitValueValid(int v) {
253    if ((v < Integer.MIN_VALUE) || (v > Integer.MAX_VALUE)) {
254      return false;
255    }
256    return true;
257  }
```

Listing 1: Obviously useless code in `com.sun.jmx.snmp.SnmpInt.isInitValueValid`

Confused Conjunctions.

The binary conjunction operators in Java (`||` and `&&`) are a steady source of logic confusion not only for novices but also for experienced developers. Confusing them will either lead to an expression that is overly permissive or one that is overly restrictive compared to the original intent.

```
1842  if (maxBits > 4 || maxBits < 8) {
1843    maxBits = 8;
1844  }
1845  if (maxBits > 8)
1846    maxBits = 16;
1847  }
```

Listing 2: Confused || conjunction in `com.sun.imageio.plugins.png.PNGMetadata.mergeStandardTree`

In the example in Listing 2, a variable `maxBits` is checked if it is greater than 4 or (operator `||`) less than 8. This is always true as the partial expressions overlap. As a result, the `maxBits` variable is always set to 8 rendering the following check (Line 1845) useless. In this example, the developer probably wanted to use the `&&` operator instead.

```
337  if (ix<0 && ix>=glyphs.length/*length of an array >= 0*/) {
338    throw new IndexOutOfBoundsException("" + ix);
339  }
```

Listing 3: Confused && conjunction in `sun.font.StandardGlyphVector.getGlyphCharIndex`

In Listing 3 we have the dual situation. Here, the developer probably wanted to use || instead of &&. Currently, the condition – as a whole – will always evaluate to false; an integer variable (`ix`) can never contain a value that is at the same time less than zero and also larger or equal to zero. Currently the precondition check is useless and `ix` could exceed the array's length or even be negative. In general, such useless pre-condition/post-condition checks may lead to deferred bugs at runtime which are hard to track down. For example, imagine that `ix` is just stored in a field for later usage and at runtime some other method uses it and fails with an exception. At that point-in-time the method that caused `ix` to contain an invalid value may no longer be on the stack and is therefore typically hard to identify.

In general confused conjunctions can render pre- or post-condition checks ineffective or mask paths that are necessary for correct behavior.

Confused Language Semantics.

The semantics of Java's `instanceof` check is in borderline cases sometimes not well understood. For example, the `instanceof` operator will return `false`, if the value is `null`.

[1]The tool and the data set are available for download at `www.opal-project.de/tools/bugpicker`.

[2]Some of the code samples shown in the following are abbreviated to better present the underlying issue.

```
381  public boolean containsValue(Attribute attribute) {
382    return
383      attribute != null &&
384      attribute instanceof Attribute &&
385      attribute.equals(...(Attribute)attribute...));
386  }
```

Listing 4: `Instanceof` and `null` confusion in `javax.print.attribute.HashAttributeSet.containsValue`

In the example in Listing 4, the developer first checks the parameter `attribute` for being not `null`. After that she checks if it is an instance of `Attribute`. However, the `instanceof` operator also checks if the object is not `null`, so the first check is redundant. But, given that the declared type of the parameter is `Attribute`, the respective value will always be either an instance of `Attribute` or `null`, hence the not-null check would be sufficient.

A second issue in this category is shown in Listing 5. In this case the developer checks if a newly created object is indeed created successfully. This is, however, guaranteed by Java's language semantics. In this case the developer most likely had a background in programming using the C family of languages where it is necessary to check that the allocation of memory was successful.

```
288  doc = new CachedDocument(uri);
289  if (doc==null) return null; // better error handling needed
```

Listing 5: New objects are never `null` in `com.sun.org.apache.xalan.internal.xsltc.dom.DocumentCache`

Instances of this category usually lead to redundant code that is detrimental to comprehensibility of the program.

Dead Extensibility.

Programmers may deliberately introduce *temporary* infeasible paths into code when programming for features that are expected to be used in the future. However, in some cases the implementation of a feature is aborted but no clean-up is done; leaving behind dead code. For example, a method that was defined by an interface with a single implementation and where the implementation of that method returns a constant value is an instance of dead extensibility.

We classify issues as belonging to dead extensibility only if (a) the package is closed for extension (e.g, everything in the `com.sun.*` or `sun.*` packages), (b) the development of the respective code is known to have ended (e.g., `javax.swing.*`) or (c) we find an explicit hint in the code.

```
189  // For now we set owner to null. In the future, it may be
190  // passed as an argument.
191  Window owner = null;


198  if (owner instanceof Frame) { ... }
```

Listing 6: Infeasible paths from unused extensibility (taken from `javax.print.ServiceUI.printDialog`)

For example, in Listing 6 the variable `owner` is set to `null` but later on checked for being of type `Frame`. This will always fail as discussed previously. Here, the comment in the code however identifies this issue as a case of programming for the (now defunct) future.

Another instance of this issue can be found in `java.util.concurrent.ConcurrentHashMap.tryPresize`. In that case our analysis identified that a variable (called `sc`) always has the value 0 which leads to a decent amount of complex dead code. Given the complexity of the code we directly contacted one of the developers and he responded: "Right. This code block deserves a comment: It was not always unreachable, and may someday be enabled ...".

In general dead extensibility primarily hinders comprehension; sometimes to a highly significant level as in the last case. Additionally, it may lead to wasted efforts if the code is maintained even though it is dead [13].

Forgotten Constant.

In case the declaration of a local variable and its first use are very distant, developers might have already forgotten about its purpose and value and assume that its value can change even though it is (unexpectedly) constant.

For example, in `javax.swing.SwingUtilities` in method `layoutCompoundLabelImpl` a variable called `rub` is set to zero and never touched over more than 140 lines of code. Then the variable is used in an expression where it is checked for being larger than zero and, if so, is further accessed. But, that code is never executed.

Issues in this category are often a hint to methods that need to be refactored because they are very long and hard to comprehend. Overall these issues are also causing maintainability problems, because they are hindering comprehension and the resulting dead code still is/needs to be maintained.

Null Confusion.

The most prevailing category of issues that our analysis discovered in the JDK is related to checking reference values for being (non)-`null`. The issues in this category can further be classified into two sub-categories: (a) unnecessary checks, which do not cause any immediate harm, and (b) checks that reveal significant issues in the code. We don't consider this distinction in the following sections because both categories are very relevant to developers [4].

An example of an issue of the first sub-category is a non-`null` check of a value stored in a (`private` and/or `final`) field that is known to only contain non-`null` values. Another instance is shown in Listing 7 - the variable `atts` is checked for being non-`null` and an exception is thrown if the check fails. Later in the code (Line 227 - not shown here) the variable is, however, checked again for being non-`null`, resulting in an infeasible `else` path.

```
211  if (atts != null)
212    types.add(att);
213  else
214    throw new IOException(paramOutsideWarning);
```

Listing 7: Ensuring non-`null`ness in `javax.management.loading.MLetParser.parse`)

Issues of the second sub-category are those where a non-`null` check is done after the reference was already (tried to be) dereferenced. An example of the latter category is shown in Listing 8. In this case, either the check is redundant or the code may throw `NullPointerException`s at runtime.

```
372  int bands = bandOffsets.length;
373  ... // [bandOffsets is not used by the next 6 lines of code]
374  if (bandOffsets == null)
375    throw new ArrayIndexOutOfBoundsException("...");
```

Listing 8: Late null check (taken from `java.awt.image.Raster.createBandedRaster`)

Range Double Checks.

We classify issues as *range double checks* if the range of a value is checked multiple times in a row such that subsequent checks are either just useless or will always fail. As in case of *Null Confusion* issues, such issues are benign if they are related to code that is just checking what is already known. However in other cases (more) significant issues may be revealed.

An issue of the first category is shown in Listing 9. Here, the variable `extendableSpaces` is first checked for being zero or less (Line 1095) and if so the method is aborted. Later on the dual check (Line 1097) is unnecessarily repeated.

```
1095  if (extendableSpaces <= 0) return;
1096  int adjustment = (targetSpan − currentSpan);
1097  int spaceAddon = (extendableSpaces > 0) ?
1098                        adjustment / extendableSpaces : 0;
```

Listing 9: Useless range check in `javax.swing.text.ParagraphView$Row.layoutMajorAxis`

An issue of the second category is shown in Listing 10. Here, the value of the variable `jcVersion` is first checked for being equal or larger than the constant `JAVA_ENC_VERSION`, which has the value 1. After that, the variable is checked again for being larger than the constant `CDR_ENC_VERSION`, which is 0. Hence, both checks are equivalent, but the code that is executed in both case clearly differs which makes it likely that it is incorrect.

```
331  byte jcVersion = jc.javaSerializationVersion();
332  if (jcVersion >= Message.JAVA_ENC_VERSION) {
333    return Message.JAVA_ENC_VERSION;
334  } else if (jcVersion > Message.CDR_ENC_VERSION) {
335    return jc.javaSerializationVersion();
336  } ...
```

Listing 10: Contradicting range checks in `com.sun.corba.se.impl.orbutil.ORBUtility`

Type Confusion.

Names of methods often explicitly or implicitly suggest that the handled or returned values have a specific type and that specific type casts are therefore safe. This, however, can be misleading and lead to runtime exceptions.

For example, the method `createTransformedShape` shown in Listing 11 suggests that a transformed version of the `Shape` object is returned that has been passed to it. However, the method always returns an instance of `Path2D.Double`. Now the method `getBlackBoxBounds` which is defined in `java.awt.font.TextLayout` calls `createTranformedShape` passing an instance of `GeneralPath` and casts the result to the very same type. This will always result in an exception as `GeneralPath` is not a subtype of `Path2D.Double`.

```
3825  public Shape createTransformedShape(Shape pSrc) {
3826     if (pSrc == null) { return null; }
3827     return new Path2D.Double(pSrc, this);
3828  }
```

Listing 11: Confusing implicit contract leading to a type confusion in `java.awt.geom.AffineTransform.createTransformedShape`

Unsupported Operation Usage.

If a method is called that always just throws an exception and that exception causes the calling method to also behave abnormally then we considered the related issues as *Unsupported Operation Usage*. Consider for example the code shown in Listing 12. Here, the `extract` method calls the `read` method, which always throws an exception. This results in two issues: First, both methods cannot be called without triggering an exception that disrupts the control flow. Second, the `extract` method calls the `create_input_stream` method which is afterwards not further used and – in particular – not closed.

```
29  static CodecFactory extract (org.omg.CORBA.Any a) {
30     return read (a.create_input_stream ());
31  }
32  static CodecFactory read (InputStream istream){
33     throw new org.omg.CORBA.MARSHAL ();
34  }
```

Listing 12: A method always throwing an exception in `org.omg.IOP.CodecFactoryHelper`

In such cases the code often seems to be in a state of flux where it is unclear what has happened or will happen.

Unexpected Return Value.

We found some cases where the name of a method suggests a specific range of return values – in particular if the name starts with `is`. In this case, Java developers generally expect that the set of return values is {`true`, `false`} and write corresponding code. This will, however, directly lead to dead code if the called method always returns the same value. A concrete example is shown in Listing 13. Here, the developer calls a function `isUnnecessaryTransform` and, if `true` sets the used transformer to `null`. However, this code is dead because the called method, which is shown in Listing 14, always returns `false`.

```
746  if (isUnnecessaryTransform(...)) {
747    conn.getDestination().setTransform(null);
748  }
```

Listing 13: Dead code due to a bug in the called method; found in `com.sun.media.sound.SoftPerformer`

```
761  static boolean isUnnecessaryTransform(... transform){
762    if(transform == null) return false;
763    ... // [the next four tests also just return false]
764    return false;
765  }
```

Listing 14: The result is always `false` in `com.sun.media.sound.SoftPerformer`

As demonstrated by the code in Listing 13, these issues are typically related to significant problems in the respective code. The example indicates that an inter-procedural analysis is required to discover such issues.

3. THE APPROACH

As described in the introduction, we are able to identify the issues presented in the previous section using a generic approach that relies on the detection of dead paths. The approach's three main building blocks: the generic detection of dead paths, the underlying analysis and the post-processing of issues to filter irrelevant ones is presented in the following.

3.1 Identifying Infeasible Paths

To identify infeasible paths in code, we first construct a control-flow graph, denoted CFG, for each method. The

CFG contains all possible control flow edges that may occur during the execution *if we do not consider any data flows.* In a second step, we perform an abstract interpretation (AI) of the code, during which a second flow graph is implicitly generated and which consists of the edges taken during the AI. We denote this flow graph AIFG in the following. The AIFG is potentially more precise than the CFG because the underlying analysis is context- and data-flow sensitive. Hence, if the AIFG contains fewer edges than the CFG then some paths in the CFG are not possible. Consider for illustration a conditional branch expression (e.g., `if`, `switch`). The CFG of such an expression contains an edge to the first instruction of each branch. On the contrary, the AIFG will not contain edges from the condition instruction to those branches, whose guarding condition cannot be the result of evaluating the condition, according to the AI.

Given a method's AIFG and CFG, we remove the edges of the former from the latter. The source instruction of the first edge of every remaining path in the CFG is reported to the developer, as it is directly responsible for an infeasible path. This instruction is the last executed one and is called the *guard instruction.* More precisely, the algorithm to detect infeasible paths is shown in Listing 15. Given a `method` (Line 1) and the `result` of its AI (Line 2), the algorithm iterates over each instruction, `instr`, of `method` (Line 4) testing whether it was evaluated (Line 5). If `instr` was executed, we iterate over all possible successors and check whether each expected path was taken (Line 7); if not, `instr` is a guard instruction and a report is generated which describes which path is never taken and why. If `instr` was not executed, it cannot be a guard instruction and the iteration continues with the next instruction.

```
1  val method : Method { val instructions : Array[Instruction] }
2  val result : AIResult { val evaluated : Set[Instruction] }
3  for {
4      instr <− method.instructions
5      if result.wasEvaluated(instr)
6      staticSuccInstr <− instr.successors // static CFG
7      if !result.successorsOf(instr).contains(staticSuccInstr)
8  } yield { /* rank and create error report w.r.t. instr */ }
```

Listing 15: Detecting Infeasible Edges

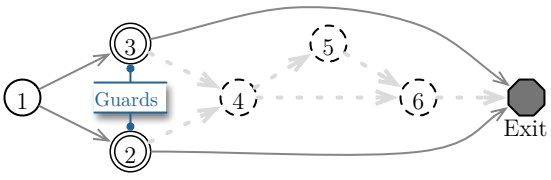

Figure 1: A CFG vs. an AIFG.

Consider for further illustration the graph shown in Figure 1. Each node in the graph represents an instruction in some piece of code, its edges represent the set of control-flows in the CFG. Now, let us assume that – by performing an abstract interpretation of the respective code – we can determine that the instructions 4, 5 and 6 are never executed and, hence, that the control-flow paths [2→**4**→6], [2→**4**→5→6], [3→**4**→6] and [3→**4**→5→6] are infeasible. In this case, the analysis will create one report for the guard instruction 2 and one for guard instruction 3.

To illustrate how the technique reveals issues presented in the previous section, we apply it to a concrete example.

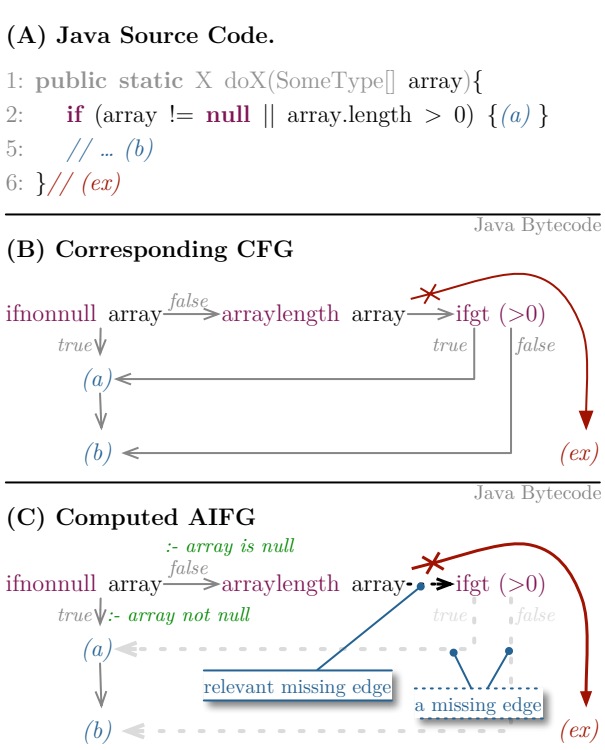

Figure 2: The Approach Exemplified.

The issue under consideration that we found multiple times in the JDK is the use of the wrong logical operator (e.g., `||`) in combination with a `null` check. The code in Figure 2(A) (Line 2) exemplifies this issue. The method `doX` will always fail with a `NullPointerException`, if the given `array` is `null`, because `length` is not defined on `null` values. Given the check of `array` against `null` that precedes the field access `array.length`, we assume that the developer actually intended to use the *logical and operator* (`&&`) instead of `||` to avoid the runtime exception. This issue is identified by our analysis because the AI determines the nullness property of reference values. When the analysis of the `null` check is performed, a constraint is created (cf. Figure 2(C)) that states that `array` is `null` if the test `ifnonnull` fails and vice versa. Now, the AI will – when it evaluates the `arraylength` instruction – always only follow the exception path and will never evaluate the `ifgt` instruction and the instructions dominated by it. The generated report will state that the predecessor instruction of `ifgt`, the `arraylength` instruction, is the root cause of the issue, as it always throws a `NullPointerException`.

3.2 The Abstract Interpretation Analysis

The analysis that performs the abstract interpretation is built on top of OPAL [14] - an abstract interpretation framework for Java Bytecode implemented in Scala.

Central to our idea is that the abstract interpretation does not pursue any specific goal – it simply tries to collect as much information as possible; the identification of infeasible paths is a side effect of doing so. Furthermore, unlike most abstract interpretation based analyses approaches, our analysis is not a whole program analysis. Instead, the analysis treats each method as a potential entry point and makes

no special assumptions about the overall state or input parameters. For example, when analyzing a method `m(Object o,int i)`, no assumptions are made about the parameters `o` and `i` (`o` could, e.g., be `null`, an alias for `this`, or reference an independent object). For each method, we perform an inter-procedural, path-, flow-, object- and context-sensitive analysis, however, only up to a pre-configured – typically very small – call chain length.

The rationale for the above design decisions is twofold: (a) to make the approach useable for analyzing libraries, and (b) to make it scalable to industrial sized applications. Libraries do not have a designated `main` method - any public method can be an entry point to the library. Hence, our decision to handle each method in isolation. Our understanding of a very large codebase is, e.g., the entire JDK 8, which consists of more than 190,000 concrete methods with more than 9,000,000 byte code instructions. An approach that does not restrict the length of call chains to be followed may find more issues, but does not scale for such codebases as the effort roughly scales exponentially. So, we deliberately trade off the number of issues that we find for scalability. This is a common design decision taken by static analysis tools [9, 7].

The analysis is designed as a product line, currently customizable with respect to: (a) abstract domains used to represent and perform computations on values, and (b) the maximum call chain length that the analysis follows.

Customizing the Abstract Domains.

Concerning (a), the current implementation is configured with the following abstract domains.

(i) For computations on integer values we use a standard interval domain based on the description found in [19]. Our domain implements all artihmetical operations supported by the JVM and facilitates path-sensitive analyses. It is further parameterized over the maximum size of the intervals for which we try to compute the result before the respective variable is assumed to take any value (the top value in the abstract domain's lattice). The parameterization over abstract domains enables us to adapt the precision of the analysis and to make it scalable to large code bases. For example, for the method in Listing 16 the analysis would compute that the return value is in the range $[200, 1000]$ if the configured maximum cardinality is larger than 800. If the maximum is, e.g., 32 the result would be *AnyInt* (the top value).

```
1  int m1(boolean test){
2    int i = 100;
3    if (test) i *= 10;
4    else i *= 2;
5    return i; // i is a value in the range [200,1000] or "Any Int"
6  }
```

Listing 16: Integer Ranges

We could as well use a domain that represents integer values as sets. In the example above, such a domain could precisely capture the information that the return value is either 200 or 1000. However, in a pre-study, we found that a domain based on ranges is superior in terms of its efficiency and hence better suited for our setting.

(ii) For the other primitive data types supported by the Java Virtual Machine (`float`, `long`, `double`), the current implementation performs all computations at the type level; hence, currently we cannot identify bugs related to these values. This limitation, however, is only a matter of putting more engineering effort, which is currently ongoing.

(iii) The domain for reference values is object-sensitive and distinguishes objects based on their allocation site. Hence, for objects created within an analyzed method, we can derive precise type information. The domain also supports alias and path-sensitive analyses. Currently, this domain cannot be configured any further.

Customizing the Maximum Call Chain Length.

Let us now consider the second customization dimension (b); adapting the maximum call chain length. At each call site, the analysis uses the configured maximal call chain length to determine whether to make the call. If the length of the current call chain is smaller than the configured maximum, the analysis invokes the called method using the current context - however, only if the information about the receiver type is precise, i.e., the call site is monomorphic. If we only have an upper type bound for the receiver object it could happen that a developer overwrites the respective method later on and would render the analysis unsound.

To illustrate the effect of configuring the call chain length, consider the code in Listing 17 and assume that the maximum call chain length is 2 and the analysis tries to find issues in method `m1`. In this case, the method `m2` will also be analyzed with the current context (`i <= 0`) to find out that `m2` will always throw an exception. However, since the max call chain length is 2 the constructor call in `m2` (Line 7) will not be analyzed. Hence, the analysis draws no conclusion about the precise exception that is thrown[3].

```
1  void m1(int i){
2      if(i <= 0 && m2(i) > 1) System.out.println("success");
3      else System.out.println("failure");
4  }
5  int m2(int i){
6      if(i > 0) sqrt(i);
7      else throw new IllegalArgumentException();
8  }
```

Listing 17: Example for the effects of call chain length

We complement the abstract interpretation analysis described so far with two simple pre-analyses. Their goal is to determine precise types for values stored in private variables respectively of values returned by private methods. In Java (and other object-oriented) libraries it is common to use interface types for fields and method return values, even if these values are actually instances of specific classes. The pre-analyses that we perform are able to determine the concrete types of private fields and of return values of private methods in 40% of the inspected cases[4].

We limit the application of these analyzes to private methods and fields only to make them useable for libraries. Extending the scope of the analyses beyond private methods/fields may lead to wrong results. A developer, using the library, may create a new subtype which stores other values of different kinds in the field or overrides the method returning a different kind of object. This cannot happen with private fields/methods, both are not accessible in subclasses and, hence, cannot be changed.

[3]The constructor of `llegalArgumentException` may throw some other exception.

[4]When analyzing the JDK 1.8.0 Update 25.

Both pre-analyses are very efficient as they only analyze one class at a time and perform abstract interpretations of a class methods using domains that perform all computations at the type level. However, they still contribute significantly to the overall power of the analysis w.r.t. identifying issues in particular if the maximum call chain length is small. If the call chain length is 1 then 10% of the issues are found based on the pre-analysis.

3.3 Post-processing Analysis Results

Obviously, it is desirable to minimize the number of false reports produced by the analysis. However, completely avoiding them is due to the complexities associated with statics analyses generally not possible. In our setting, we distinguish between two kinds of false reports. First, there are infeasible paths that are correctly identified as such by the analysis, but which do not hint at issues in the software. They are rather due to (I) code generated by the compiler, (II) intricacies of Java and (III) common best practices. We call reports related to such issues *irrelevant reports* to indicate that they are not real false reports in the sense of wrong. Nevertheless, they would be perceived as such by developers as they would not help in fixing any source code level issues. Second, there are paths that are wrongly identified as infeasible. Such paths are due to the imprecision of the analysis w.r.t. code that uses reflection or reflection-like mechanisms. In the following, we discuss each source of false reports and the heuristics used to identify and suppress them.

Next, we discuss these sources and how we handle them as part of a post-processing step.

Compiler Generated Dead Code.

The primary case that we identified, where Java compilers implicitly generate dead code, is due to the compilation strategy for `finally` blocks. The latter are typically included twice in the byte code, for both cases when an exception occurred respectively did not occur. This strategy often results in code that is provably dead, but where there is nothing to fix at the source code level.

```
1  void conditionInFinally(java.io.File f) {
2    boolean completed = false;
3    try {
4      f.canExecute();
5      completed = true;
6    } finally {
7        if (completed) doSomething();
8  } }
```

Listing 18: Implicit dead code in finally

For illustration, consider the code in Listing 18. The `if` statement (Line 7) gets included twice in the compiled code. If an exception is thrown by the `canExecute` call (Line 4), `completed` will always be `false`. Therefore, the call to `doSomething` would be dead code. However, if no exception is thrown, `completed` will be `true` and, hence, the branch, where completed is `false` is never taken. Now, to identify that there are no issues at the source code level, it is necessary to correlate both byte code segments to determine that both branches are taken. In other words, we have to recreate a view that resembles the original source code to determine that there is nothing to fix at the source code level.

To suppress such false warnings, we use the following simple heuristics. We search in the byte code for a second `if` instruction that accesses the same local variable and which is in no predecessor/successor relation with the currently considered guard instruction, i.e., both instructions are on independent paths. After that, we check that one of the two `if` instructions strictly belongs to a finally block, i.e., we check that the `if` instruction is dominated by the first instruction of a finally handler, which the other one is not.

The Intricacies of Java.

In Java, every method must end each of its paths with either a return instruction or by throwing an exception. Now, if a method is called, whose sole purpose is to create and throw an exception (e.g., `doFail` in Listing 19), thus intentionally aborts the execution of the calling method (`compute` in Listing 19), the calling method still needs to have a `return` or `throw` statement. These statements will obviously never be executed and, hence, without special treatment would be reported.

```
1  Throwable doFail() { throw new Exception(/* message*/); }
2
3  Object compute(Object o) {
4      if(o == null) [ return doFail(); OR throw doFail(); ]
5      else return o;
6  }
```

Listing 19: `doFail()` Always Throws an Exception

Such infeasible paths, however, are not related to a fixable issue in code and should hence be suppressed. We do not generate a report if (a) the first instruction of an infeasible path is a `return` or `throw` instruction and (b) the guard instruction is a call to a method that always throws an exception.

Established Idioms.

A source of several irrelevant reports in our study of the JDK is the common best practice in Java programs to throw an exception if an unexpected value is encountered in case of a `switch` statement. The implementations in these cases always contain a default branch that throws an error or exception stating that the respective value is unexpected. A prototypical example is shown in Listing 20.

```
1  switch (i) {
2    case 1 : break;
3    // complete enumeration of all cases that should ever occur
4    default : throw new UnknownError();// should not happen
5  }
```

Listing 20: Infeasible Default Branch

In the JDK we found multiple instances of such code, which vary significantly. In particular, the exceptions that are thrown vary widely ranging from `UnknownError` over `Exception` and `RuntimeException`, to custom exceptions. Furthermore, in several cases the code is even more complex. In these cases a more meaningful message, which captures the object's state, is first created by calling a helper method that creates it. In some cases that helper method even immediately throws the exception. To handle all cases in a uniform way, we perform a local data- and control-flow analysis

that starts with the first instruction of the default branch to determine whether the default branch will always end by throwing the same exception.

Assertions.

Another source of irrelevant reports related to correctly identified infeasible edges is code related to assertions. We found several cases, where we were able to prove that an `assert` statement will always hold or where an `AssertionError` was explicitly thrown on an infeasible path. In the latter case, the Java code was following the pattern `if( /* condition was proven to be false */ ) throw new AssertionError( /*optional message*/ )`) and is directly comparable to the code that is generated by a Java compiler for `assert` statements. As in the previous case, reports related to such edges are perceived as irrelevant by developers. We suppress related reports by checking whether an infeasible path immediately starts with the creation of an `AssertionError`.

Reflection and Reflection-like Mechanisms .

Though we tried to reduce the number of false positives to zero, we found some instances of false positives for which the effort of programmatically identifying them would by far outweigh the benefits. In these cases, the respective false positives are due to the usage of Java Reflection and/or the usage of `sun.misc.Unsafe`. Using both approaches it is possible to indirectly set a field's value such that the analysis is not aware of it. For example `java.lang.Thread` uses `Unsafe` to indirectly set the value of the field `runner` using a memory address stored in a long variable. Our analysis in this case cannot identify that the respective field is actually set to a value that is different from `null` and hence, creates an issue related to every test of the respective variable against `null`. As the evaluation will show, the absolute and relative numbers of false positives are so low that the heuristics can still be considered effective.

4. EVALUATION

We evaluated our analysis by using the approach on the JDK to (I) get a good understanding of the issue categories that our analysis can identify and the effectiveness of the techniques for suppressing false warnings, and (II) to derive a good understanding of how the maximum call chain length and the maximum cardinality of integer ranges effects the identification of issues as well as the runtime. After that, we applied the approach to the Qualitas Corpus [22] to test its applicability to a diverse set of applications and to validate our findings.

4.1 JDK 1.8.0 Update 25

4.1.1 Issue Categories

The issue categories that we identified were discussed in Section 2. The distribution of the issues across different categories is shown in Table 1. It was determined by two of the authors of this paper as well as an advanced student. They manually evaluated all reported issues in the packages that constitute the JDK's public API: `java*`, `org.omg*`, `org.w3c*` and `org.xml*`. To achieve a consistent rating of the issues across all persons, we randomly selected 10% of all issues and rated them individually. Afterwards the ratings were compared and discussed to get consistent results when classifying the remaining issues. The analysis was executed using a maximum call chain length of 3 and setting the maximum cardinality of integer ranges to 32; no upper bound was specified for the analysis time per method.

Overall 556 reports were related to the public API; including the reports that were automatically identified as irrelevant because they are expected to belong to compiler generated dead code, to assertions or to common programming idioms. All 556 reports were manually inspected to check for false positives, to classify the reports and to assess the filtering mechanism. In the results of the analysis we found 19 reports that were related to code for which we did not find the source code and they were dropped from the study.

Table 1: Issues per Category

Category	Percentage
Obviously Useless Code	1%
Confused Conjunctions	2%
Confused Language Semantics	3%
Dead Extensibility	9%
Forgotten Constant	4%
Null Confusion	54%
Range Double Checks	11%
Type Confusion	3%
Unexpected Return Value	5%
Unsupported Operation Usage	7%
False Positives	1%

From the 537 remaining reports 279 ($\approx$ 52%) were automatically classified as irrelevant. A further analysis revealed that 81% of the 279 irrelevant reports are related to compiler generated/forced dead code - a finding that clearly demonstrates the need to suppress warnings for compiler generated dead code. Another 12% are due to assertions or common programming idioms. The remaining 7% were false negatives, i.e. the filtering was too aggressive and suppressed warnings for true issues. All these cases were related to a method `read`[5] that just throws an exception and which was called at the end of the calling methods. Hence, they were automatically classified in the *Intricacies of Java* category. Given that the filter otherwise proved very helpful, we decided to accept these false negatives.

The vast majority of the reported issues – i.e., the issues that were deemed relevant by the analysis – is related to `null` values. A closer investigation revealed that developers are often literally flooding the code with `null` checks that only check what is already guaranteed. However, as already discussed in the introduction we also found numerous cases where field and method accesses were done on `null` values. The second top most category of issues is related to checking that an integer value is in a specific range. As in the previous case this is sometimes useless, but in many cases it identifies situations where it is obvious that the code does not behave as intended.

For each of the other categories we found so few issues that the development of a specially targeted analysis would probably not be worthwhile for other approaches. However, taken together, these categories make up 34% of all issues

[5]Presented in Listing 12.

and given that we identify them for free, these findings are significant.

Finally, we found two false positives (< 1% of all issues) as discussed in the previous section.

4.1.2 Varying the Analysis Parameters

To determine the sensitivity of the analysis on changed analysis parameters, we ran several instances of the analysis with maximum call chain lengths of 1, 2, 3, 4 and 5 and with a maximum cardinality settings for integer ranges of 2, 4, 8, 16 and 32. If the analysis time of a single method exceeded 10 seconds, the respective analysis was aborted. This time limit was chosen because it enables the vast majority of methods to complete within the timeframe, but it still avoids that the entire analysis is completely dominated by a few methods with extraordinary complexity[6]. The analysis was executed on a 8-Core Mac Pro with 32GB of main memory. Overall, we ran the analysis 25 times.

Table 2: Evaluation of the Analysis Parameters Sensitivity

Max Call Chain Length	Max Integer Ranges Cardinality	Issues Relevant	Issues Filtered	Issues Total	Time [s]	Aborted Methods
1	2	690	1157	1847	10	0
1	4	699	1172	1871	11	0
1	8	701	1180	1881	13	0
1	16	702	1182	1884	21	0
1	32	702	1182	1884	27	0
2	2	1078	1248	2326	25	0
2	4	1090	1269	2359	31	0
2	8	1093	1277	2370	43	0
2	16	1094	1279	2373	73	0
2	32	1094	1279	2373	149	1
3	2	1189	1252	2441	60	0
3	4	1201	1273	2474	78	0
3	8	1204	1281	2485	139	0
3	16	1205	1283	2488	289	1
3	32	1205	1283	2488	894	10
4	2	1224	1252	2476	156	0
4	4	1236	1273	2509	205	1
4	8	1237	1281	2518	438	7
4	16	1237	1283	2520	1259	27
4	32	1233	1283	2516	6117	63
5	2	1225	1252	2477	457	4
5	4	1235	1273	2508	990	7
5	8	1239	1281	2520	1566	20
5	16	1234	1283	2517	5482	80
5	32	1233	1283	2516	39274	143

As expected, increasing the max call chain length or the maximum cardinality also increases the number of identified issues. However, the effectiveness of the analysis in terms of *number of issues* per *unit of time* decreases sharply. As shown in Table 2, the number of additional relevant issues that are found if the call chain length increases from 1 to 2 is remarkable (≈ 50%). However, the number of additional issues that are found if the maximum call chain length is increased from 4 to 5 is much less impressive (depending on the maximum cardinality of integer ranges between 1 and 2 additional issues).

Hence, increasing the maximum cardinality of ranges of integer values is initially much less effective than increasing the maximum call chain length. If the maximum call chain length is 1 then increasing the cardinality from 2 to 32 increases the necessary effort more significantly than increasing the call chain length by one, though the latter will lead to the detection of more issues. Nevertheless, certain issues can only be found if we increase the maximum cardinality and at some point increasing the cardinality is the only feasible way to detect further issues. For example, increasing the call chain length from 4 to 5 does not reveal significantly new issues. However, the analysis still revealed new issues when we increased the range's cardinality. Furthermore, if we specify an upper bound of 10 seconds for the analysis of a single method – which includes the time needed to analyze called methods – the number of aborted methods rises significantly and we therefore even miss some issues.

In summary, from an efficiency point-of-view, a maximum call chain length of 2 or 3 and a maximum cardinality of 4 or 8 seems to be the sweet spot for the analysis of the JDK.

We also examined some of the issues (using random sampling) that only show up if we increase the call chain length from one to two to get an initial understanding of such issues. This preliminary analysis reveals that most additional issues were related to a defensive coding style, which can also be seen in the *Null confusion*, *Range Double Checks* or *Confused Language Semantics* categories.

A prototypical example is shown in Listing 21. The check of `tmpFile` against `null` (Line 2) is useless as the method `createTempFile` will never return `null`.

```
1  tmpFile = File.createTempFile("tmp","jmx");
2  if (tmpFile == null) return null; (* return null is dead *)
3
4  File createTempFile(String pre, String suf) throws ... {
5      return createTempFile(pre, suf, null); }
6  File createTempFile(String pre, String suf, File dir) throws ... {
7      ... File f; ...
8      if (!fs.createFileExclusively(f.getPath())) throw ...
9      return f; (* The only return statement. *) }
```

Listing 21: Defensive code in `javax.management.loading.MLet.getTmpDir`

4.2 Qualitas Corpus

We ran the analysis twice on all 109 projects of the Qualitas Corpus [22][7]. The corpus is well suited to evaluate general purpose tools such as the proposed one as it is a curated collection of projects across all major usage areas of Java (Desktop-/Server-side Applications and Libraries). The evaluation is done once using a maximum call chain length of 1 and once using a maximum length of 2. Overall, this study supports our previous findings. As in case of the JDK study, increasing the maximum call chain length from 1 to 2 led to the identification of a significantly higher number of issues. Sometimes even more than twice as many issues are found (e.g., Eclipse, JRuby or Findbugs), which further

[6]A particularly complex method can be found in `jdk.internal.org.objectweb.asm.ClassReader`. The method `readCode` is more than 500 lines long and contains several loops that each call multiple methods.

[7]Version 20130901.

stresses the importance of context-sensitivity for bug finding tools. Overall, we found more than 11 000 issues across all projects. Interestingly, we did not find any issues in the projects jUnit, jFin, jGraphT, Trove, sablecc and fit. A close investigation of these projects revealed that they are small, maintained by a small number of developers, and possess good test suites.

5. RELATED WORK

Besides the term *dead code*, which we are using, other closely related terms are also frequently used in related work. For example, Chou et al. [8] use the term *unreachable code*. In their case a fragment of code is unreachable if there is no flow of control into the fragment and is thus never on any path of execution of the program. Their focus is, however, on generating good explanations that facilitate reasoning why the code is dead.

Kasikci et al. [17] use the term *cold code* to refer to code that is rarely or never executed at runtime. They propose to use a dynamic analysis based on sampling the execution frequencies of instructions to detect such code. As in our case, code that is identified as dead is seen as a threat to software maintenance. In a similar context Eder et al. [13] use the term *unused code* and they also point out that unused code constitutes a major threat to the maintainability of the software. As usually, approaches using dynamic analyses as proposed by Kasikci et al. and our approach which uses static analysis complement each other.

Approaches based on formal methods that rely on SMT solvers were also used to identify variants of dead code. The approach described in [1, 2, 6] for example also determines code that is never executed. They use the term *infeasible code* to describe code that is contained in an infeasible control-flow path. Schäf et al. [21] take an even wider look at the topic by looking for *Inconsistent Code*. I.e., code that always fails and code that makes conflicting assumptions about the program state. Compared to our approach these approaches are trying to prove that the software is free of respective issues. However, they are interested in dead paths of a specific kind while we are interested in dead paths as such and find dead paths related to a variety of issues.

Abstract interpretation [10] was already used in the past to detect dead code [11, 20]. But, compared to our approach the respective tools try to prove that the code is free of such issues by performing whole-program analyses. The result is that those tools favor precision over scalability and are often not targeting the analysis of libraries. Payet et al. [20] for example use the Julia static analyzer to identify – among others – dead code in Android programs.

Though detecting and eliminating code that is dead has been the subject of a lot of research related to compiler techniques [12, 18], our case study has shown that it is still a prevalent issue in compiled code. Nevertheless, standard techniques that rely on live variable analysis or other data-flow analyses are now an integral part of many modern integrated development environments such as Eclipse or IntelliJ. They help developers to detect some related issues. Compared to these approaches we perform an inter-procedural, context-sensitive analysis based on abstract interpretation which is well beyond the scope of compilers and – as demonstrated – is able to find many more issues.

Other approaches which also try to identify a broader range of issues, such as FindBugs [9] or JLint [3], implement one issue detector for each kind of issue. This limits these approaches to only detect those issues the developer explicitly had in mind which is not the case in our approach. Compared to the presented analysis they can also find issues that are not related to the control-/data-flow.

6. CONCLUSION

In this paper, we have presented a generic approach for the detection of a range of different issues that relies on abstract interpretation. The approach makes it possible to adapt the executed analysis to a library's or application's needs and can easily be extended in a generic fashion. As the evaluation has shown, the presented approach is able to find a large number of issues across many categories in mature large Java applications. We have furthermore motivated and discussed the filtering of false positive that are technical artifacts as well as those related to common programming idioms. The presented techniques reduce the number of (perceived) irrelevant reports to nearly none. Overall, the approach is very promising and will be the foundation for future work on self-adaptive static analyses.

The tool is available for download at:
`www.opal-project.de/tools/bugpicker`

7. ACKNOWLEDGMENTS

This work was supported by the International Research Experience Program, the BMBF within EC SPRIDE, by the BMBF within the Software Campus initiative (01IS12054) and by the LOEWE excellence initiative within CASED.

The authors would like to thank Tobias Becker for his support in classifying the JDK's issues.

8. REFERENCES

[1] S. Arlt, Z. Liu, and M. Schäf. Reconstructing Paths for Reachable Code. 8144:431–446, 2013.

[2] S. Arlt and M. Schäf. Joogie: Infeasible Code Detection for Java. In *CAV'12: Proceedings of the 24th international conference on Computer Aided Verification*. Springer-Verlag, July 2012.

[3] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In *Proceedings of the 13th Australian Conference on Software Engineering*, ASWEC '01, pages 68–, Washington, DC, USA, 2001. IEEE Computer Society.

[4] N. Ayewah and W. Pugh. The Google FindBugs Fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–252, New York, NY, USA, 2010. ACM.

[5] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the International Conference on Software Engineering*. IEEE, May 2013.

[6] C. Bertolini, M. Schäf, and P. Schweitzer. Infeasible Code Detection. *VSTTE*, 7152(Chapter 24):310–325, 2012.

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and

D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[8] H.-Z. Chou, K.-H. Chang, and S.-Y. Kuo. Facilitating Unreachable Code Diagnosis and Debugging. *2011 16th Asia and South Pacific Design Automation Conference ASP-DAC 2011*, pages 1–6, Feb. 2011.

[9] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving Your Software Using Static Analysis to Find Bugs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 673–674, New York, NY, USA, 2006. ACM.

[10] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why Does Astrée Scale Up? *Form. Methods Syst. Des.*, 35(3):229–264, Dec. 2009.

[12] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, Mar. 2000.

[13] S. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K.-H. Prommer. How Much Does Unused Code Matter for Maintenance? In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1102–1111, Piscataway, NJ, USA, 2012. IEEE Press.

[14] M. Eichberg and B. Hermann. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[15] E. Geay, E. Yahav, and S. Fink. Continuous Code-quality Assurance with SAFE. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 145–149, New York, NY, USA, 2006. ACM.

[16] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.

[17] B. Kasikci, T. Ball, G. Candea, J. Erickson, and M. Musuvathi. Efficient Tracing of Cold Code via Bias-Free Sampling. *USENIX Annual Technical Conference*, pages 243–254, 2014.

[18] J. Knoop, O. Rüthing, and B. Steffen. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 147–158, New York, NY, USA, 1994. ACM.

[19] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[20] E. Payet and F. Spoto. Static Analysis of Android Programs. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, pages 439–445, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] M. Schäf, D. Schwartz-Narbonne, and T. Wies. Explaining Inconsistent Code. *the 2013 9th Joint Meeting*, pages 1–11, Aug. 2013.

[22] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering*