Bixie: Finding and Understanding Inconsistent Code

Tim McCarthy*, and Philipp Rümmer†, and Martin Schäf*
*SRI International
†Uppsala University

***Abstract*—We present Bixie, a tool to detect inconsistencies in Java code. Bixie detects inconsistent code at a higher precision than previous tools and provides novel fault localization techniques to explain why code is inconsistent. We demonstrate the usefulness of Bixie on over one million lines of code, show that it can detect inconsistencies at a low false alarm rate, and fix a number of inconsistencies in popular open-source projects. Watch our Demo at http://youtu.be/QpsoUBJMxhk.**

I. Introduction

We present `Bixie`, a static analysis tool that automatically detects inconsistent code in Java programs. Inconsistent code, as defined for example in [1], or [2], occurs when two or more statements in a program make inconsistent assumptions. In other words, inconsistent code is either unreachable, or any execution through it must lead to an error. Common examples of inconsistent code are nullness-checks of pointers that have already been dereferenced, or nullness-checks followed by a dereference of the same pointer. Such inconsistencies are not necessarily bugs, but they are a code smell because they can never be executed safely.

Inconsistent code is an interesting target for static analysis because it can be detected locally and without knowing the intended purpose of the code. That is, such an analysis can be implemented in a fully automated fashion and, in theory, without false positives: if we can prove that a piece of code in isolation has no normal terminating execution (either because it is unreachable or because any execution must crash), this proof will also hold in any larger context. Hence, inconsistent code can be detected in code snippets or even while typing. No user-provided specification is required.

In our prior work, we have demonstrated efficient algorithms and prototype tools to detect inconsistent code [3], [4] and given the theoretical underpinning on how to explain inconsistent code [1]. With `Bixie`, we present the first complete tool to detect and report inconsistent code in Java programs. `Bixie` is based on our previous prototype Joogie [4] and extends this in numerous ways: `Bixie` uses Boogie [5] as an intermediate language and provides a reusable translation from Java bytecode into Boogie. `Bixie` implements several filters to suppress false alarms that are rooted in inconsistent code introduced by the compiler when translating Java source code into bytecode (see Section III). `Bixie` implements the interpolation-based inconsistent code explanation algorithm presented in [1]. This makes `Bixie` the only tool that reports the pair of statements that are inconsistent, compared to other tools that only report one statement that has no execution [2], [4].

To demonstrate the usefulness of `Bixie`, we conducted a series of experiments on several open source Java programs. The goal of our experiments was to demonstrate that inconsistent code exists even in well maintained code, that inconsistent code can be detected efficiently and with relatively few false alarms by `Bixie`, and that developers care about the reported inconsistencies.

Contributions. We make the following key contributions:

1) we present a robust tool to detect inconsistent code on real-world Java programs,
2) we present the first implementation of an interpolation-based algorithm to explain inconsistent code that works on real-world code,
3) we find and report inconsistent code in several open-source projects.

The `Bixie` tool, an on-line version of the tool, a demonstration video, and the experimental data from this paper are available on the `Bixie` website: http://csl.sri.com/projects/bixie/. Links to GitHub repositories including all source code scripts to reproduce the experiments from this paper are also given there.

II. Example

We illustrate how `Bixie` finds and reports inconsistent code along the example in Figure 1. The figure shows a piece of code in the Apache Tomcat web server. The conditional choice in line 5 assumes that the variable `size` is greater than the length of the array `resolvers`. Then, in line 6 the array is accessed at that position. This is inconsistent with the implicit runtime assertion that arrays can only be accessed within their legal bounds. That is, every time line 6 is reached, a runtime exception is thrown.

```
1  @Override
2  public synchronized void add(
3      ELResolver elResolver) {
4    super.add(elResolver);
5    if (resolvers.length < size) {
6      resolvers[size] = elResolver;
7    } else {
8    ...
```

Fig. 1. Inconsistent Code found by `Bixie` in Tomcat. The conditional in line 5 assumes that the variable `size` is larger than the size of the array which is inconsistent with the array access in line 6.

To detect this inconsistency, `Bixie` takes the Tomcat source code as input, either as Java source files or as class files, and uses the Soot bytecode analysis framework [6] to preprocess it into the simpler Jimple format. The Jimple format is then translated into the Boogie intermediate verification language. In this step, all implicit runtime assertions, such as array bounds checks, are turned into assertion statements. On the Boogie program, Bixie performs several abstractions to make the set of control-flow paths finite [7]. The abstraction over-approximates loops, eliminates procedure calls, and eliminates other language features that are hard to analyze, such as reflection and multithreading. The details and limitations of this abstraction are discussed in more detail in Section III.

The resulting abstract Boogie program is now analyzed one procedure at a time. Bixie does not perform inter-procedural analysis. Each procedure is turned into a logic formula of first order [8]. This formula has the property that each model satisfying this formula can be mapped to a feasible complete control-flow path in the original program (we discuss the soundness of this step in the next section).

This formula is sent to the Princess theorem prover [9]. If the formula is satisfiable, we retrieve the corresponding path from the model and add a clause to the formula that disallows all models that refer to the same path. We repeat this process until the formula becomes unsatisfiable. Once this process terminates, we have collected a set of paths that cover all statements that can occur on feasible complete paths. All statements that are not covered in that process are inconsistent with other parts of the code.

For our example in Figure 1, Bixie can cover all statements but the assignment in line 6. However, reporting only line 6 might not be enough – an inconsistency requires at least two statements. Therefore, Bixie implements the approach to explain inconsistent code from [1]. For each inconsistent line, Bixie extracts the sub-program that consists of all control-flow paths through this line. The first-order logic formula for this sub-program looks similar to the following:

$$\begin{aligned}
&\mathtt{super.add(elResolver)}\\
\wedge\, &\mathtt{resolver} \neq \mathtt{null}\\
\wedge\, &\mathtt{resolvers.length} < \mathtt{size}\\
\wedge\, &\mathtt{resolver} \neq \mathtt{null}\\
\wedge\, &0 \leq \mathtt{size} < \mathtt{resolvers.length}\\
\wedge\, &\mathtt{resolver[size]} = \mathtt{elResolver}\\
\wedge\, &\ldots
\end{aligned}$$

We know, that this formula must be unsatisfiable (because the line provably does not occur on a feasible complete path). We use Princess to generate one Craig interpolant after each conjunct from the proof of unsatisfiability. Given an unsatisfiable logic formula $A_0 \wedge \ldots \wedge A_n$, the i-th Craig interpolant I_i in this sequence is a formula such that $A_0 \wedge \ldots \wedge A_{i_1} \implies I_i$, and $I_i \wedge A_i \wedge \ldots \wedge A_n \models false$. That is, I_i is an abstraction of all conjunctions before A_i that is sufficient to prove that the formula is unsatisfiable. The first interpolant in that sequence is by definition $true$ and the last interpolant is $false$. Bixie now checks, for each statement, if the same formula can be used as interpolant before and after a conjunct in the original formula (or, for that matter before and after a statement in the program). If so, we know that this conjunct did not change the reason why the formula is unsatisfiable and we do not need to report the corresponding statement in the original program. This approach is similar to, but a bit more flexible than unsatisfiable-core-based approaches such as [10]. For a detailed description of our algorithm we refer to [1].

Propagating the interpolants for our example reveals that only for the two conjuncts, $\mathtt{resolvers.length} < \mathtt{size}$ and $0 \leq \mathtt{size} < \mathtt{resolvers.length}$, there is no interpolant that holds before and after them. The interpolant before the first conjunct is $true$ which means that none of the conjuncts before are relevant for the proof, and the conjunct after the second is $false$ which means that none of the conjuncts after are necessary for the proof. In between, the interpolant $\mathtt{resolvers.length} < \mathtt{size}$ which means that none of the conjuncts between these two are relevant. Hence, we can report the lines 5 and 6, from which the two conjuncts were generated, to the user and further add the information that it is line 6 that does not occur on any feasible complete path. Bixie currently does not report the actual interpolants, even though they may contain valuable information about the relevant variables and the relationship between them. This is because the interpolants generated from Boogie (or Jimple) programs contain information and variables that do not exist in the source code which makes them hard to read and sometimes confusing.

We reported the inconsistency between line 5 and 6 in our example to the Tomcat developers together with a pull request which was merged within two days. In the following, we discuss the implementation of Bixie, the engineering challenges we faced on the way, and the compromises we had to make to build a usable tool.

III. Implementation Notes

Figure 2 gives a high-level overview of the architecture of Bixie. In the following we discuss the engineering challenges and limitations of the different components in Bixie.

Bixie uses Soot as a front-end to parse and pre-process the input Java programs. Soot has the advantage that it already provides a mature infrastructure to parse and analyze both source and bytecode programs. Bixie currently uses only the translation into Jimple format. We discuss the challenges that arise from using Soot later in this section.

The first component of Bixie is called Jar2Bpl which translates Jimple code into Boogie. During the translation, we have to trade precision for scalability on several occasions. The biggest challenge is the proper translation of exception handling. Loosely speaking, in Java bytecode, any statement can throw any exception, so building a conservative (and therefore sound) control-flow graph is impractical. We take a moderate approach and model common runtime exceptions such as NullPointerException or IndexOutOfBounds-Exception, while ignoring others such as Concurrent-ModificationException. Further, we use the strong

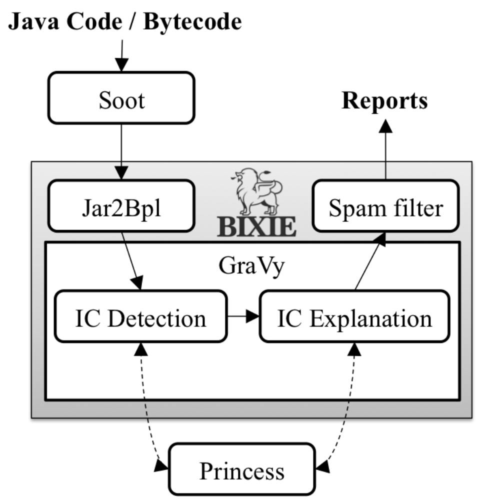

Fig. 2. Architecture diagram of Bixie. Bixie uses Soot as a front-end to parse and pre-process Java (byte)code and Princess as a theorem prover. Bixie is composed of three components: Jar2Bpl, which translates the Jimple code generated by Soot into Boogie, GraVy, which detects and reports inconsistent code, and a spam filter to suppress false alarms.

assumption that no exception should leave a procedure unless stated in the throws clause. We further do not model language features related to reflection. For multithreading we take a middle course. We do not model interleaving, but at the border of synchronized blocks and in blocks that catch InterruptedException or similar exceptions, we add non-deterministic assignments to all variables that may be modified by another thread. All these choices are unsound and may lead to false alarms. However, a sound modeling of Java is not practical. One might argue that there are better choices than ours, but for our experiments this middle course turned out to be very practical.

For the detection of inconsistent code (IC Detection), Bixie uses a component called GraVy. GraVy is a general purpose tool to compute feasible paths in Boogie programs which we also use for gradual verification [11]. It detects inconsistent code as described in Section II for one Boogie procedure at a time. That is, Bixie does not perform any inter-procedural analysis. GraVy uses a sound over-approximation of loops and procedure calls shown in [7].

One of the main novelties in Bixie is the algorithm to explain inconsistent code (IC Explanation). This algorithm implements the approach from [1] for real-world code. Therefore, we implemented an interpolation procedure that is robust on logic formulae over complex theories including arrays and quantifiers.

A key component to the usability of Bixie is the Spam filter that suppresses false alarms. These false alarms are, in most cases, not rooted in the unsound decisions we made when translating Java into Boogie, but in our decision to use bytecode instead of source code. We illustrate this problem using two examples. The first example is the representation of finally-blocks in bytecode. The Java compiler duplicates finally blocks and adds them to each location where they can be reached. That is, if the finally contains code that is only reachable if a particular object is null, this might be reachable in one duplicate of this block but not in another, which is then falsely reported as inconsistent. Unfortunately, these are not exact clones, they use slightly different instructions and variables, so detecting these duplications is not always simple. The Spam filter implements several mechanisms to suppress such false alarms. Another source of false alarm is inconsistent code that is introduced by the compiler. The Java 7 compiler, for example, is known to introduce unreachable code when translating try-blocks that use resources.[1] The Spam filter implements several checks to suppress such false alarms.

Many sources of false alarms are rooted in our decision to use bytecode and Soot instead of working directly on the source code. In retrospective, using bytecode was not the best decision. In the future, we plan to refactor Bixie to directly use Java source code. This will reduce the number of false alarms and make Bixie more practical.

IV. Evaluation

We applied Bixie to over one million lines of Java code. The scripts we used to run the experiments are available online.[2] The benchmarks are picked arbitrarily from GitHub and represent the subset of the analyzed code that could be evaluated before publication of this article. More benchmarks can be found on our website. Note that Bixie does not perform inter-procedural analysis, so scaling to millions of lines of Java code only reduces the selection bias but does not demonstrate scalability. Bixie uses at most 40 seconds per procedure and then times out. The fault localization is allowed to use another 30 seconds if inconsistent code is found.

Table I lists our benchmark programs and the reported inconsistencies. Columns 4 and 5 count the occurrences of inconsistent code that are worth reporting. That is, code that either must lead to an exception, or that is unreachable. Column 6 counts reports where implicit else-branches were unreachable. We found that developers often prefer to use an explicit else-if instead of a simple else to make code more readable, which renders the implicit else-branch unreachable. While this code is inconsistent according to the definition, we still treat it as a false alarm and try to suppress it with our Spam filter.

The remaining columns count false alarms rooted in our abstraction that could not be suppressed. Overall, we only got one false alarm from ignoring reflection. The next column counts false alarms that are rooted in the duplication of finally blocks in bytecode. The second to last column counts false alarms caused by our abstraction of threads. In most cases, these are loops that can only be left if a flag is set by another thread. The last column counts all other sources of unsoundness. About half of these cases refer to our unsound handling of integers: we use mathematical integers, but in rare

[1]http://stackoverflow.com/questions/25615417/try-with-resources-introduce-unreachable-bytecode

[2]https://github.com/martinschaef/bixie/releases/tag/0.9.

Benchmark	kLoC	Time (minutes)	possible bug	unreachable	missing else	reflections	finally	threads	other
Ant	271	385	0	5	2	1	3	1	0
Bouncy Castle	461	133	3	13	1	0	1	1	3
Hadoop	507	100	2	23	2	0	4	1	4
Hive	692	35	17	23	33	0	3	3	1
jMeter	114	121	2	2	2	0	0	2	1
Joda Time	84	2	0	0	0	0	0	0	4
Log4j	65	49	0	3	0	0	3	2	1
Maven	43	46	1	1	11	0	1	0	0

TABLE I

RESULTS OF APPLYING BIXIE ON SEVERAL OPEN-SOURCE PROGRAMS. COLUMNS REPRESENT THE CATEGORIES OF INCONSISTENT CODE THAT WE FOUND. THE COLUMNS FOUR AND FIVE REPRESENT ACTUAL PROBLEMS WHERE CODE MUST THROW AN EXCEPTION OR IS UNREACHABLE. THE REMAINING COLUMNS COUNT LESS INTERESTING OCCURRENCES OF INCONSISTENT CODE AND FALSE ALARMS

cases where programmers check round-trip arithmetics, this leads to false alarms. The remaining cases can be attributed to bugs in Bixie that we could not fix before the deadline.

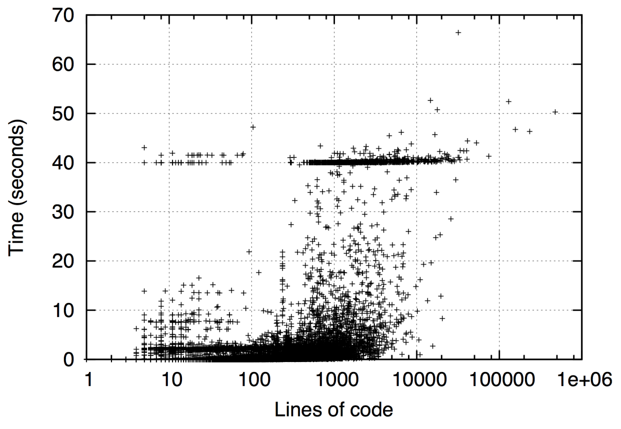

Fig. 3. Computation time per procedure for all benchmark programs. The lines of code on the horizontal axis refers to bytecode instructions.

Figure 3 shows the computation time per procedure in our experiments. In the vast majority of cases Bixie analyzes procedures in seconds. Timeouts typically occur for constructors that initialize a large amount of objects (e.g. for GUIs), and other complex procedures like parsers.

We reported our findings from Table I to developers on GitHub to evaluate if inconsistent code actually does matter. Before we reported inconsistent code, we made sure that the code is not inconsistent because of debug constants (e.g., code being disabled with an if-false), and we made sure that we can provide a patch. For the curious reader, we keep a list of all pull requests on our website. So far, we have successfully merged pull requests with fixes of inconsistent code for Bouncy Castle, jMeter, Maven, Tomcat, and Soot.

V. RELATED WORK

Approaches to detect inconsistent code have been presented by [2] and [4]. The former two use deductive verification-based techniques similar to the one in Bixie. Unfortunately, this tool is not available for comparison.

One of the main novelties in Bixie is its ability to explain inconsistent code using the approach from [1]. This approach uses a proof-based fault abstraction to eliminate statements that are not needed to understand the inconsistency. Similar proof-based approaches have been presented in [10] and [12]. Our implementation is the first interpolation-based approach that can be applied to arbitrary Java programs.

VI. CONCLUSION

With Bixie, we have created a useful and usable tool to detect inconsistent code. We have demonstrated that Bixie finds interesting issues in real-world code and that developers do accept inconsistent code as bugs that they are willing to fix. In the future, we will further improve the precision of Bixie by performing the analysis directly on source code instead of bytecode which will eliminate a large percentage of false alarms. Eventually, we will integrate Bixie into an IDE and perform inconsistent code detection on-the-fly while the programmer is typing.

REFERENCES

[1] M. Schäf, D. Schwartz-Narbonne, and T. Wies, "Explaining inconsistent code," in *FSE*. New York, NY, USA: ACM, 2013, pp. 521–531. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491448

[2] A. Tomb and C. Flanagan, "Detecting inconsistencies via universal reachability analysis," in *ISSTA*, 2012, pp. 287–297.

[3] S. Arlt, P. Rümmer, and M. Schäf, "A theory for control-flow graph exploration," in *ATVA*, 2013, pp. 506–515.

[4] S. Arlt and M. Schäf, "Joogie: Infeasible code detection for java," in *CAV*, 2012.

[5] K. R. M. Leino and P. Rümmer, "A polymorphic intermediate verification language: design and logical encoding," in *TACAS*, 2010.

[6] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot - a Java Optimization Framework," in *CASCON 1999*, 1999, pp. 125–135. [Online]. Available: www.sable.mcgill.ca/publications

[7] J. Hoenicke, K. R. Leino, A. Podelski, M. Schäf, and T. Wies, "Doomed program points," *Formal Methods in System Design*, 2010.

[8] S. Arlt, P. Rümmer, and M. Schäf, "Joogie: From java through jimple to boogie," in *SOAP*. ACM, 2013.

[9] P. Rümmer, "A constraint sequent calculus for first-order logic with linear integer arithmetic," in *LPAR*, 2008.

[10] M. Jose and R. Majumdar, "Bug-Assist: Assisting fault localization in ANSI-C programs," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 504–509.

[11] S. Arlt, C. Rubio-González, P. Rümmer, M. Schäf, and N. Shankar, "The gradual verifier," in *NASA Formal Methods*. Springer, 2014, pp. 313–327.

[12] V. Murali, N. Sinha, E. Torlak, and S. Chandra, "What gives? a hybrid algorithm for error trace explanation," in *VSTTE*. Springer, 2014, pp. 270–286.