Severity Levels of Inconsistent Code

Martin Schäf and Ashish Tiwari

SRI International
Menlo Park CA 94025

Abstract. Inconsistent code detection is a variant of static analysis that detects statements that never occur on feasible executions. This includes code whose execution ultimately must lead to an error, faulty error handling code, and unreachable code. Inconsistent code can be detected locally, fully automatically, and with a very low false positive rate. However, not all instances of inconsistent code are worth reporting. For example, debug code might be rendered unreachable on purpose and reporting it will be perceived as false positive.
To distinguish relevant from potentially irrelevant inconsistencies, we present an algorithm to categorize inconsistent code into a) code that must lead to an error and may be reachable, b) code that is unreachable because it must be preceded by an error, and c) code that is unreachable for other reasons. We apply our algorithm to several open-source project to demonstrate that inconsistencies of the first category are highly relevant and often lead to bug fixes, while inconsistencies in the last category can largely be ignored.

1 Introduction

In this paper, we present a severity ranking for *inconsistent code*. Inconsistent code refers to a statement that is never executed on a normal terminating execution. That is, this statement is either unreachable, or any execution containing this statement leads to an error[1]. The concept of inconsistent code is appealing because it lends itself to be detected using static analysis – one simply has to prove that none of the paths containing the statement of interest is feasible. Hence, by using a sound over-approximation of the feasible paths of a program, one can build a tool to detect inconsistent code that never raises false alarms (at least in theory). Over the past years, several static analysis tools have been developed that detect, among other things, inconsistent code (e.g., [2,12,18,20]). We have seen interesting bugs rooted in inconsistent code being detected, e.g., in the Linux kernel [7], in Eclipse [11], or in Tomcat [16]. However, not all inconsistent code is worth reporting. For example, unreachable code, which is a special case of inconsistent code, is often used deliberately or is unavoidable. Reporting harmless instances of unreachable code would be perceived as false positives.

[1] E.g., the violation of an assertion or a (user-provided) safety property. The concrete definition of error depends on the tool.

Hence, it is vital to distinguish different reasons why code is inconsistent and prioritize warnings based on this.

In this paper, we introduce three tiers of inconsistent code – `doomed`, `demood` (doomed spelt backward), and `unr` code. Inconsistent code is categorized as `doomed` if it is possibly reachable (i.e., we cannot prove that it is unreachable), but any execution passing through it must lead to an error (violation of a safety property). Inconsistent code is categorized as `demood` if it is unreachable because any execution that would reach it must necessarily trigger an error earlier. Inconsistent code is categorized as `unr` if it is unreachable and not `demood`. We present an inconsistent code detection algorithm that can categorize inconsistent code as `doomed`, `demood`, or `unr`.

We show on a set of open-source benchmarks that this categorization can be made with very small computational overhead, and that the proposed severity levels help to identify critical inconsistencies easily. In most cases, code categorized as `doomed` indicates a patchable bug in the program while code categorized as `unr` tends to be less interesting and in the vast majority of cases not worth patching. Our experiments further indicate that inconsistent code of category `doomed` and `demood` is rare compared to unreachable code of category `unr`. Another observation is that false alarms, caused by imprecise handling of advanced language features, such as multi-threading and reflection, are always categorized as `unr`. Hence, by only reporting warnings of type `doomed` and `demood`, we obtain a highly usable inconsistent code detection tool.

2 Overview

We motivate our severity levels for inconsistent code using the illustrative examples in Figure 1. Each of the four procedures, `f1` to `f4`, has inconsistent code in the then-block of the conditional choice. The reason why this code is inconsistent, however, is different for each procedure.

In procedure `f1`, line 3 is inconsistent because on any execution passing through line 3, `o` is guaranteed to be `null` which violates the (implicit) run-time assertion in line 5 that `o` must be properly allocated before it can be de-referenced. We categorize this type of inconsistent code `doomed` because it may be (forward) reachable and inevitably leads to an error. This category comprises what we want to report with the highest severity. For this category of inconsistent code, the developer has to be notified because the only way to prevent an error is to make this code unreachable. Later in this section we will show some real-world examples of this case.

Procedure `f2` contains inconsistent code in line 4. To reach this line, `o` has to be `null`. This, however, would violate the implicit run-time assertion that `o` must not be `null` in line 2. We categorize this case, where code is rendered unreachable by an (implicit) safety property, as `demood`. Code in this category often indicates that error handling is in the wrong place (e.g., a null-check of a pointer that has already been de-referenced). While this is not necessarily a bug, it certainly indicates confusion about the necessary error handling, which often

```
1 void f1(Object o) {
2   if (o == null) {
3     // inconsistent
4   }
5   o.toString();
6 }
```

```
1 void f2(Object o) {
2   o.toString();
3   if (o == null) {
4     // inconsistent
5   }
6 }
```

```
1 void f3() {
2   Object o =
3     new Integer(123);
4   if (o == null) {
5     // inconsistent
6   }
7 }
```

```
1 void f4(Object o) {
2   int i=0;
3   if (o == null) {
4     i++ // inconsistent
5   }
6   assert (i==0);
7 }
```

Fig. 1. Four examples of inconsistent code. In each procedure, the `then`-block is inconsistent. The procedures `f1`, `f2`, and `f3` represent the shortest possible examples for inconsistent code of category `doomed`, `demood`, and `unr` respectively. We added the procedure `f4` to clarify that inconsistent code is more than just forward or backward reachability.

is an indicator for bit rot or unclear specifications. Code categorized as `demood` will be reported with the second highest severity. While technically being plain unreachable code, it still indicates there may be a potential risk of an assertion violation.

In procedure `f3`, we have an example of unreachable code. Line 5 is unreachable because, in Java, `new` cannot return `null`. In this case, no run-time assertion is involved in making line 5 unreachable and we categorize it as `unr`. We report unreachable code of this category with the lowest severity (or even hide it completely). There are many reasons why code in this category should not be reported: in languages without pre-processor, such as Java, code is often rendered unreachable on purpose (e.g., debug code). Furthermore, translating high-level languages into simpler three-address code formats often introduces unreachable code, e.g., through translation of conjunctions into nested conditional choices, or inlining of `finally`-blocks in exception handling (see [1]). Also, unsound abstractions, such as ignoring possible interleaving in multi-threaded code, may introduce false positives which manifest as unreachable code.

Procedure `f4` has inconsistent code in line 4. This procedure illustrates the difference between unreachability and inconsistency. Unlike the previous examples, where the inconsistent code was either forward- or backward-unreachable, the inconsistent code in this example is both forward- and backward-reachable. Since line 4 is inconsistent and forward reachable, our algorithm will categorize it as `doomed` and report it with a high priority.

Motivating examples. Figure 2 shows two occurrences of inconsistent code categorized as `doomed` by our approach. Both cases have been reported to the de-

```
1 //@org.apache.jasper.el.JasperELResolver
2 public synchronized void add(ELResolver elResolver) {
3   super.add(elResolver);
4   if (resolvers.length < size) {
5     resolvers[size] = elResolver;
6   //...
```

```
1 //@org.apache.maven.repository.MetadataResolutionResult
2 public MetadataResolutionResult addError(Exception e) {
3   if (exceptions==null)
4     initList( exceptions );
5   exceptions.add( e );
6   return this;
7 }
```

Fig. 2. Two examples of inconsistent code in the wild. The first example was found and fixed in Tomcat. Line 4 guarantees that line 5 access the array out of bounds and line 4 is forward reachable. The second example was found and fixed in Maven. Line 4 uses the list initializing incorrectly, thus, line 5 must throw an exception. In both cases, our algorithm categorizes the inconsistent code as `doomed` because it is reachable and must lead to an exception.

velopers and our fixes have been accepted. In the first example taken from the application server Tomcat, the operator in line 4 is flipped resulting in an inevitable out-of-bounds exception being thrown in line 5. By inspecting the code, it was easy to see that this was just a typo and the operands merely had to be flipped. The second example is taken from Maven. Here, the procedure `initList` is used in the wrong way. The author of this code assumed that `initList` has a side effect on the field `exceptions` which is not the case. Even though our analysis is not really inter-procedural, it detects that `exceptions` cannot be modified by this call and hence detects that executing line 4 must lead to a `NullPointerException` in line 5.

These are just two examples of the type of problems that are categorized and reported as `doomed` by our algorithm. These are bugs that seem trivial but do occur in practice. In fact, they even occur on the main branches of well tested long standing open-source projects. In our evaluation, we will discuss in more detail which projects we analyzed, and how the severity levels helped us to stay focused on genuine bugs and ignore false positives.

3 Inconsistent Code

In this section, we formally define the notion of inconsistent code and present our static analysis approach for detecting inconsistent code. In subsequent sections, we will define the three categories of inconsistent code and then we will present

$$\begin{array}{rcl} \mathit{Program} & ::= & \mathit{Block}^* \\ \mathit{Block} & ::= & \mathit{label}:\ \mathit{Stmt};^*\ \textbf{goto}\ \mathit{label}^*\textbf{;} \\ \mathit{Stmt} & ::= & \mathit{VarId} := \mathit{Expr}\textbf{;}\ \mid\ \textbf{assert}\ \mathit{Expr}\textbf{;}\ \mid\ \textbf{assume}\ \mathit{Expr}\textbf{;} \end{array}$$

Fig. 3. The syntax of our simple (unstructured) Language

an inconsistent code detection procedure that also outputs the category with each instance of inconsistent code.

We present our approach using the simple unstructured language shown in Figure 3. The language is a simplified version of Boogie [4] and is sufficient for demonstration purposes. Even though it is simple, it is expressive enough to encode a large class of programs in high-level languages such as Java [1].

A program in this language is a set of *Blocks*, with one unique entry block, b_e, where execution of the program starts, and a unique sink block, b_x, where execution terminates. Each block is connected to possibly multiple other blocks using (non-deterministic) `goto`s. A block is a piece of sequential code containing assignments, assertions, and assumptions. Assertions have no effect if the asserted condition evaluates to true and abort the execution with an error, otherwise. Assume statements behave similar to assertions except that execution *blocks* if the assumed condition evaluates to false. Assume statements are used to reduce non-determinism introduced by gotos and model common control-flow constructs such as conditional choices or loops. Assignments update the value of program variables. We do not explicitly present the syntax for expressions. We do allow the assignment of non-deterministic values to variables (e.g., for abstraction).

A (complete) path in a program is a sequence of blocks $b_e b_1 \ldots b_x$ such that each block in the sequence is connected (via *goto*) to the next block in the sequence. Throughout this paper, the term *path* always refers to a complete path, starting in b_e and ending in b_x. For our purposes here, the semantics of a path in a program is just a Boolean value indicating if the path is *feasible*; that is, if the sequence of assignment statements on this path can be executed without violating any assumption or assertion. Formally, we define the semantics of a path, feasibility of a path, and inconsistent code as follows.

Definition 1. *The function* $[\cdot]$ *mapping a sequence of statements* $s_1; \cdots ; s_m$ *to a first-order formula is defined recursively as follows:*

$$[s_1; s_2] = [s_1] \wedge [s_2] \qquad [v := e] = v = e \qquad [\textbf{\textit{assume}}\ e] = e \qquad [\textbf{\textit{assert}}\ e] = e$$

The <u>*semantics*</u> $[b_e b_1 \ldots b_x]$ *of a path* $b_e b_1 \ldots b_x$ *is the formula* $[s_{b_e}; s_{b_1}; \ldots ; s_{b_x}]$, *where* $s_{b_e}; s_{b_1}; \ldots ; s_{b_x}$ *is a static single assignment form for the straight-line program* $\mathit{Stmt}_{b_e}; \mathit{Stmt}_{b_1}; \ldots ; \mathit{Stmt}_{b_x}$ *obtained using the statements* Stmt_{b_i} *in the definition of block* b_i.

A path π *is* <u>*feasible*</u> *if the formula* $[\pi]$ *is satisfiable.*

A block (and each statement in that block) is <u>*inconsistent*</u> *in a program, if there is no feasible path inside the program containing this block.*

Feasibility of a path π can be checked using SMT solvers to check satisfiability of $[\pi]$; see also [13].

Finding Inconsistent Code

Since the number of paths in the program can be unbounded (due to loops), most existing algorithms for detecting inconsistent code perform *program abstraction* to remove loops from the program, and then use a satisfiability checker on the abstract program; see for example [10, 12, 18]. We discuss these steps next.

Program Abstraction. The goal of the program abstraction step is to eliminate loops and procedure calls from the program. This is usually done by replacing the corresponding code by non-deterministic assignments to (an over-approximation of) all variables modified in the original code surrounded by a (possibly trivial) pair of pre- and postcondition. The property guaranteed by the abstraction step is that it only adds executions to the program but never removes one (see [10,18]). We use $\texttt{abs}(P)$ to denote an abstraction of program P.

Static Single Assignment. In the second step, we apply static single assignment transformation [5] to $\texttt{abs}(P)$ to get a program $\texttt{ssa}(\texttt{abs}(P))$. The program $\texttt{ssa}(\texttt{abs}(P))$ is a loop-free program in which each variable is only written once. We refer to such a program as *passive program.*

Inconsistent Code Detection. Finally, the passive program $\texttt{ssa}(\texttt{abs}(P))$ is encoded into a first-order logic formula $[\texttt{ssa}(\texttt{abs}(P))]$ such that each model of this formula maps to some feasible paths in the passive program. Formally, given a passive program in our language from Figure 3, its encoding into first-order logic is done as follows:

Statement	First-order representation
$[\mathit{Program} ::= \mathit{Block}_0 \mathit{Block}_1 \ldots \mathit{Block}_n]$	$\bigwedge_{0\leq i\leq n}([\mathit{Block}_i])$
$[\mathit{Block} ::= \ \mathit{label}: \ \mathit{Stmt};^* \ \textbf{goto}\, \mathit{label}_1 \ldots \mathit{label}_n\textbf{;}]$	$\mathit{label} = ([\mathit{Stmt};^*] \wedge \bigvee_{0\leq i\leq n} \mathit{label}_i)$
$[\mathit{Stmt}_0; \ldots \mathit{Stmt}_n;]$	$\bigwedge_{0\leq i\leq n}([\mathit{Stmt}_i])$
$[\textbf{assume } e], [\textbf{assert } e], [v := e]$	As in Definition 1

The most important step is the translation of *Block*. Each block comes with a *label* which becomes a Boolean variable in the first-order representation. This variable is true (in any model) if and only if there exists a feasible suffix from that block to a terminal block of the program. Since the program is already in single assignment form, each program variable can be translated into a variable in the first-order formula of appropriate (SMT) type. Going from program types to SMT types may require inserting additional assertions (or be a source

		Statement s is possibly reachable.	
		Yes	No
Executions containing s lead to an error.	Yes	s is `doomed`	s is `demood`
	No	-	s is `unr`

Fig. 4. Categories of inconsistent code.

of unsoundness); for example, when fixed size integers are encoded as natural numbers. We do not discuss this last point in this paper.

Given program P, the procedure for detecting inconsistent code in P first computes the formula $\phi := [\texttt{ssa}(\texttt{abs}(P))]$, and then it checks satisfiability of $\phi \wedge \mathit{label}_e$, where label_e is the label for the initial block block_e. If the formula is satisfiable and M is a model, then we can extract *at least one* complete path with labels, say $\mathit{label}_e, \mathit{label}_1, \ldots, \mathit{label}_n, \mathit{label}_x$, where each label in the path is mapped to *True* in the model M. Each block in this path is marked "consistent". We iterate this process after adding an additional constraint to ϕ that eliminates M from the set of models of the new formula. The new constraint could either be a blocking clause that excludes this path (i.e., $\neg \bigwedge \mathit{label}_i$), or it could just set the label of one of the unmarked blocks to *True*. Iterating this process excludes at least one feasible path in each iteration. Since the number of feasible paths in a passive program is finite (because we removed loops and procedure calls), eventually the new formula becomes unsatisfiable and the process terminates. At termination, all unmarked blocks are output as "inconsistent code". Since the abstraction of loops and procedure calls only adds executions, we have the guarantee that all inconsistent code found by the above procedure on $\texttt{abs}(P)$ is also inconsistent in the original program P.

The above procedure describes the basic steps necessary to build a tool that detects inconsistent code as described in [2,16,18]. In the following, we show how this basic inconsistent code detection can be extended to distinguish different categories of inconsistency.

Note that, for inconsistent code detection, assumptions and assertions are treated in the same way because, for the proof of inconsistency, it is not relevant why paths through a block are not feasible.

4 Severity Levels of Inconsistency

The key contribution of this paper is the introduction of the concept of severity levels for inconsistent code. This categorization is intended to reflect the confidence of the static analysis tool in its claim about the presence of a bug in the software and its severity.

We categorize inconsistent code in two dimensions. First, we distinguish between inconsistent code that is possibly reachable versus inconsistent code that is provably unreachable. Second, we also distinguish inconsistent code based on whether executions along paths containing that code results in assertion violation. Figure 4 shows the categories resulting from the distinction along these two dimensions: `doomed`, for inconsistent code that is possibly reachable and leads to an assertion violation error; `demood`, for inconsistent code that is provably unreachable because it must be preceded by an exception; and `unr`, for code that is provably unreachable, but executions along paths containing it do not cause assertion violations (that is, they all block due to assume violations).

Definition 2 (Severity Levels). *Given a program P, let P' denote the program obtained from P by removing all **assert** statements.*

An inconsistent block in a program is <u>`unr`</u> *if for every complete path π in P' containing this block, the formula $[\pi]$ is unsatisfiable.*

An inconsistent block in a program is <u>`doomed`</u> *if for some complete path π in P' containing this block, the formula $[\pi]$ is satisfiable and for some path π in P from an initial block to this block, the formula $[\pi]$ is satisfiable.*

An inconsistent block in a program is <u>`demood`</u> *if for some complete path π in P' containing this block, the formula $[\pi]$ is satisfiable and for every path π in P from an initial block to this block, the formula $[\pi]$ is unsatisfiable.*

Intuitively, if all paths through some block cause an assertion violation, then that block is either `doomed` or `demood`. It is `doomed` if that block is reachable, and it is `demood` if it that block is not reachable. All other instances of inconsistent code are categorized as `unr`.

We remark here that **assert** statements in our program can arise either from explicit assert statements in the source, or from implicit assert statements that arise, for example, when dereferencing a pointer or dividing by a number.

Since our inconsistent code detection and categorization procedure will necessarily run on an *abstraction* of some concrete program, we relate a categorization on an abstraction to a categorization on the concrete in Lemma 3. For our purposes, abstractions can just add behaviors: formally, program Q is an abstraction of P, if Q and P share the same blocks, and for every path π in P, there is a corresponding path σ in Q (containing the same blocks) such that $[\pi] \Rightarrow [\sigma]$ is valid in both cases – when we retain the **assert** blocks in P and Q and also in the case when we remove the **assert** blocks from P and Q.

Lemma 3. *Let Q be an abstraction of P. Then, if a block is* `unr` *in Q, then it is* `unr` *in P. If a block is* `demood` *in Q, then it is either* `demood` *or* `unr` *in P. If a block is* `doomed` *in Q, then it is either* `doomed` *or* `demood` *or* `unr` *in P.*

Proof. Suppose a block b is `unr` in Q. Consider any complete path π through that block in P. We know there is a corresponding path σ in Q such that $[\pi] \Rightarrow [\sigma]$. Since block b is `unr` in Q, the formula $[\sigma]$ (with all asserts removed) is unsatisfiable; and hence $[\pi]$ (with all asserts removed) is also unsatisfiable.

Statement	First-order representation
[**assume** e]	e
[**assert** e]	$e \vee$ **ignore**
$[v := e]$	$v = e$

Fig. 5. To detect which code is inconsistent because

Next, suppose block b is `demood` in Q. Consider any partial path π in P from an initial block to block b. Since b is `demood` in Q, for the corresponding partial path σ in Q, $[\sigma]$ is unsatisfiable. Since $[\pi] \Rightarrow [\sigma]$, this implies that $[\pi]$ is unsatisfiable. This shows that b can not be `doomed`. Hence, b is either `demood` or `unr`. □

Since abstractions add behaviors and inconsistent code is about the absence of feasible behaviors, it is clear (also from Lemma 3) that if we use sound abstractions to compute inconsistent code, we will never get a "false positive" instance of inconsistent code. In reality, however, a static analyzer in general and our tool in particular, has to use all kinds of abstractions – both over (sound) and under (unsound) – to enable scalable analysis [15]. Under-approximations can cause false positives. An even more interesting feature of inconsistent code is that false positives for inconsistent code can also arise from perfect semantic-preserving transformations (i.e. no abstractions) because introducing inconsistent code does not change semantics of programs. Preprocessors and transformers used in compilers and analysis tools can often introduce inconsistent code, which is irrelevant for the developer. Our categorization is significant also because all such false-positive inconsistencies get categorized as `unr` in our approach. Pragmatically, irrespective of whether an inconsistency is a true one or a false positive, it will be counted as a false positive in practise if it is not fixed by a developer, and this aspect will guide our evaluation in Section 6.

In the following, we discuss how the inconsistent code detection algorithm outlined in Section 3 can be extended to distinguish the three severity levels.

5 Algorithm for Categorizing Inconsistent Code

We refine the inconsistent code detection algorithm from Section 3 to also report the categories `doomed`, `demood`, and `unr`, of inconsistent code.

Recall that to appropriately categorize inconsistent code, we need to be able to 1) detect if a statement is inconsistent because the executions containing it lead to an error, and 2) check if a statement is forward reachable.

The first check can be easily integrated in the existing algorithm. After the algorithm has covered all feasible paths, we simply remove all assertions from the program and check if there are statements that can be covered now but couldn't be covered before. These are the statements that are inconsistent because of assertion errors (i.e., `doomed` or `demood`). Everything that still cannot be covered after removing the assertions falls into the category `unr`. To implement this step efficiently, we modify the way we encode programs into first-order logic as shown in Figure 5. The only change is that we translate assertions of the form **assert**(e) into $e \vee$ *ignore* instead of just e. Here, *ignore* is an uninitialized Boolean variable that we introduce. That is, if *ignore* is true, the whole expression becomes true and thus the assertion e is ignored.

Our refined algorithm works as follows: first, we compute the first-order formula ϕ_{new} using the new encoding with the alternative treatment of assertions. Then, in Phase 1, we run the algorithm from Section 3, but in place of the old ϕ, we use ϕ_{new} and, before we start searching for models, we push the axiom $ignore = false$ on the theorem prover stack (for our implementation we use Princess [17], but other SMT solvers can be used equally well). That is, in Phase 1, we do inconsistent code detection exactly as before. Once we have marked all feasible blocks, we pop $ignore = false$ from the prover stack (thus allowing the solver to pick $ignore = true$ which removes all asserts), and continue our search for (more) feasible paths in Phase 2. Every block that is marked in this second phase (and was left unmarked in the first phase) is inconsistent solely due to assert violations; that is, it either falls into `doomed`, or `demood`. Everything that is unmarked even after the second phase is categorized as `unr`.

With this small extension, we are already able to distinguish `unr` from the other two categories. What is left is to distinguish `doomed` from `demood`. To that end, we need to check if the inconsistent code is reachable. This step can be implemented by a single theorem prover query (e.g., [12]) by encoding the subprogram containing all traces from the program's source to this statement using the simple encoding from Section 3, and checking the satisfiability of the resulting formula. If the formula is SAT, then the statement is reachable, and the inconsistent code is categorized as `doomed`. If it is unsatisfiable, the statement is unreachable and it is categorized as `demood`.

Theorem 4. *The procedure outlined above correctly categorizes code as* `doomed`*,* `demood`*, or* `unr`*.*

We recall that, in practice, the categorization is performed on some abstraction, and even though Lemma 3 informs us that our categorization as `doomed` and `demood` (computed on the abstract) may not hold for the concrete, we still report the severity level computed on the abstract as the severity level for the concrete. Now, we have to show experimentally that these categories are useful and that our intuition is correct that `doomed` is the most important, followed by `demood`, and that `unr` can mostly be ignored.

6 Evaluation

We show experimentally that the three severity levels introduced above can be effectively used to differentiate inconsistencies that are deemed relevant by developers from the ones that are considered irrelevant. Specifically, we answer the following research questions:

1. Does our categorization improve usability of inconsistent code detection? Are reports in the `doomed` category rare and highly relevant to developers?

Benchmark	Inconsistencies	`doomed`	`demood`	`unr`	Total time	Overhead
Apache Cassandra	127	0	0	127	415.142s	5.636s
Apache Flume	12	5(1)	0	7	1475.955s	31.412s
Apache Hive	534	11	7	516	12623.264s	179.478s
Apache jMeter	9	0	0	9	1389.117s	29.394s
Apache Maven	15	1	0	14	777.079s	10.78s
Apache Tomcat	62	7(2)	0	56	5141.671s	170.065
Bouncy Castle	23	2	0	21	2067.994s	22.358s
WildFly (JBoss)	31	7	0	24	3130.415s	41.268s

Table 1. Results of applying our approach on several open-source programs. The table shows per benchmark the number of detected inconsistencies, the categories they fall into, the time for analyzing the benchmark, and the overhead introduced by our approach. For Flume and Tomcat, which use custom run-time assertions libraries, we needed to specify their assertion procedure to avoid false alarms.

Are reports in `demood` rare and interesting (but less relevant than the previous category)? Does the bulk of reports fall into the `unr` category, and are the reports non-critical so that we can hide them from the user unless she deliberately asks to see them?

2. Can we categorize inconsistent code at a reasonable cost?

Experimental Setup. We implemented the proposed extensions from Section 5 on top of our Bixie tool [16]. Bixie is based on Soot [19] and performs intra-procedural inconsistent code detection on Java bytecode. The tool and all scripts necessary to repeat the presented results are available on the tool website.

We evaluate our extensions to Bixie on a set of popular open-source Java projects with a total of over a million lines of code. We picked the projects without having a particular pattern in mind. Mostly, we picked projects because we could build them easily. The projects include popular projects from the Apache foundation, such as Cassandra, Flume, Hive, jMeter, Maven, and Tomcat. We also applied our analysis to the crypto library Bouncy Castle and the application-server WildFly (formally known as jBoss).

The upside of picking real open-source projects is that we will find real bugs that we can report and we avoid the confirmation bias that would arise from handcrafted examples. The downside is that popular open-source projects tend to be of good quality and rather well-tested meaning that inconsistent code will be rare. However, as we will see, it still exists.

For each project, Bixie scans one procedure at a time for inconsistent code with a timeout of 40 seconds. If Bixie is not conclusive within the given time limit, it reports nothing for that procedure. We analyze all projects on a standard desktop PC. Table 1 summarizes our results, showing the number of inconsistent code snippets detected in total, and by category, as well as the total computation time per benchmark and the extra time spent by our extensions to put inconsistent code into the different categories.

Discussion. With our experimental results, we try to answer our first research question, whether the proposed categories make sense and help to identify rele-

Benchmark	`doomed`	`demood`	`unr`	Total	fixed
Apache Cassandra	0	0	1	1	1
Apache Flume	1	0	0	1	1
Apache Hive	3	1	0	4	11
Apache jMeter	0	0	2	2	2
Apache Maven	1	0	0	1	1
Apache Tomcat	1	0	0	1	1
Bouncy Castle	1	0	4	5	5
WildFly (JBoss)	6	0	0	6	6

Table 2. Number of inconsistencies from Table 1 for which we proposed a patch per category and in total, and number of inconsistencies that have been fixed by the time of writing this paper.

vant inconsistent code quickly. Looking at the distribution of inconsistent code over the different categories in Table 1, we can see that, as expected, a large part of the detected code falls into the `unr` category. A welcomed side effect is that all false positive that arise from our unsound abstraction of multi-threading or reflection fall into this category. That is, none of the `doomed` inconsistencies qualify as false positive. To our surprise, `demood` inconsistencies are very rare. We assume that this can be largely attributed to the fact that the most common Java IDEs, Eclipse and IntelliJ, use constant propagation to warn the user about a subset of `demood` inconsistent code (but Eclipse, for example, often does not warn about `doomed` inconsistent code).

Table 2 shows for how many inconsistencies of each category we have proposed a patch, and how many of those got accepted by developers. Further down, we will discuss these findings for each benchmark individually. Out of the total 24 `doomed` inconsistencies (after removing the inconsistencies related to Guava calls as described in the caption of the table), we provided patches for 13, and by the time of writing this paper, 22 have been fixed (in the 9 cases where we did not provide patches, the developers fixed the bugs by themselves). That is, 91% of `doomed` inconsistent code has been fixed. We only reported 1 out of 7 `demood` inconsistencies (all of which have been found in Hive). We did not report the remaining 6 because the developers did not respond to our earlier pull requests for this project and we did not want to spam their Git. Three of the remaining six `demood` inconsistencies were unnecessary `null`-checks. The other three occurred in three instances of the same cloned code and resulted from a bug in Bixie which we were not able to fix by the time of submission. In total, we found 774 `unr` inconsistencies out of which we reported 7 that we found worth patching. The remaining inconsistencies were either deliberately unreachable code, unreachable code that occurred in bytecode but not in source code (see [1]), or false positives from unsound translation of multi-threading, reflection, or other programming bugs. Out of the 7 reported `unr` inconsistency, only one had an effect on the program behavior which we discuss below (in the findings for Cassandra), the others were mostly cosmetic.

In the following we discuss, benchmark by benchmark, what we found, what we patched and the developer feedback that we got.

For Cassandra, we reported one inconsistent statement from `unr`. The code computed a random Boolean by computing `rnd%1==0`. Since modulo 1 is constant, the Boolean expression is constant, which caused unreachable code. None of the other 127 statements in `unr` was worth reporting.

For Flume, we reported one of five inconsistent statements from `doomed` where a variable of reference type was checked for nullness and later, one of it's fields was accessed. The four remaining inconsistent statements from `doomed` were found because Flume uses the Guava library for run-time assertions. Without inter-procedural analysis, our analysis does not see that Guava throws an exception if an assertion fails. Thus the analysis assumes that, if the assertion fails, the code behind it is still reachable (and, in these cases, inconsistent). For code that uses custom assertion libraries, our analysis will produce these 'false alarms', unless we provide contracts (which can be done manually), or perform inter-procedural analysis.

For Hive, we reported several bugs but since there was no feedback from the developers so we decided not to submit patches for the remaining bugs. However, by the time of writing this paper, all `doomed` inconsistent code had been removed by the developers (without acknowledging our pull requests). We assume that the developers only mirror their git to GitHub and hence could not integrated our pull requests. The large number of `unr` inconsistent code in Hive is rooted in generated code that contains a lot of deliberately unreachable code.

For jMeter, we reported two occurrences of `unr`. Both were duplicated cases in case splits. While these cases were not changing the behavior of the code they were obvious mistakes that had a straight forward patch, so we decided to report them anyway.

For Maven, we reported the one bug found by our tool which is shown in the second listing of Figure 2. Our patch got accepted.

For Bouncy Castle, we report 5 occurrences of inconsistent code and all fixes got accepted. We only report one of the two `doomed` inconsistencies. We did not report second inconsistency which was a pointer de-reference that inevitably failed if the loop, which iterated over a list, was not entered. However, we (and our algorithm) could not immediately decide if this list may be empty. While adding an additional check may help to harden the code we decided not to write a patch. For the other `doomed` inconsistency, which was a possible array out-of-bounds read after loop, we submitted a patch. We further submitted patches for four `unr` inconsistencies which were obviously unreachable because of duplication. None of these were actual bugs but since they were easy to fix, we decided to patch them anyway.

For Tomcat, we reported one out of seven instances of `doomed` inconsistent code. Similar to Flume, five of the seven entries in `doomed` are inconsistent because Tomcat uses its own run-time assertion library. Once we provide a specification for these procedures, these five reports disappear. One reported `doomed` inconsistent code was rooted in an implicit `else`-case that would lead to an inevitable run-time exception. However, since we believe this case cannot occur, we decided not to propose a patch.

For WildFly, we submitted patches for 6 out of 7 occurrences of `doomed`. All got accepted (but have not been merged at the time of writing the paper). We did not report one occurrence, which was an implicit `else`-block in a switch whose execution must cause an exception. However, it looked like the developers decided to use an `else if`-block instead of the `else` to make the code more readable so we decided not to propose a patch.

To summarize, we can answer our first research question with **yes**. Our categories significantly improve the usability of our inconsistent code detection tool: distinguishing `doomed` inconsistencies from which over 90% are relevant and worth fixing, from `unr` inconsistencies where less than 1% was fixed and which may contain false positives, dramatically improves the user experience of the tool. To our surprise, `demood` inconsistencies did not play a role at all. As discussed above, we believe that modern IDE support is sufficient to detect and eliminate `demood` inconsistencies before they find their way into the repository. To answer our second research question, we look at the computational overhead in last column of Table 1 to see if our extensions are prohibitively expensive. This is not the case. For our benchmarks the time overhead for putting inconsistent code into categories is in the single digit percentage. Even for projects like Cassandra and Hive where many inconsistencies are found the overhead is small. This is because a large percentage of the detected inconsistent code falls into the `unr` category for the (potentially expensive) reachability check in our algorithm does not need to be performed. Hence, we can give a positive answer to our second research question: categorizing inconsistent code can be done at a reasonable computational overhead. In particular, given the number of non-interesting reports it can suppress, the overhead time clearly pays off for the user.

Threats to validity. We identify the following three threats to validity in our experimental setup. First, we only picked the master-branches of each project. Most of these projects use some form of continuous integration (CI), so the master-branch is usually well tested. This may distort how inconsistencies are distributed across the categories during development. The CI system may have commit hooks that reject contributions according to various rules. WildFly, for example, enforces coding conventions for each commit, which potentially affects the number of `demood` inconsistencies. However, in general, we assume that, without CI, we would just find more relevant inconsistencies and our finding that the proposed categorization improves usability still holds.

Second, the selection of benchmark projects may be biased because we selected only projects which we could compile (and hence analyze) easily. However, since all projects are major open-source projects, we believe that it is still possible to generalize from the results.

Finally, the selection of tools, theorem prover, etc. can be seen as a threat to validity. However, since there is no competing tool to detect inconsistent code we can only assume that our implementation is not biased towards or against the finding in this paper. For transparency, we make the code and experimental setup available online [16].

7 Related Work

Algorithm that detect, among other things, inconsistent code have been published in several previous papers (e.g., [2, 6, 18]). The term inconsistent code is used by [2, 10, 18], in other papers, the same phenomenon is called doomed program points [10], or deviant behavior [7].

A similar technique to detect unreachable code in the programs with annotations has been presented in [12]. Like our approach, their technique is based on deductive verification, but only detects a subset of `demood` and `unr`, but would not detect `doomed` inconsistent code.

Our approach of using Boolean flags to disable assertions is inspired by the work of [14] which uses a similar encoding to obtain error traces and the assertions that cause them to fail from SMT solver counterexamples.

Other papers have discussed the value of post-processing reports from static analysis tools. In [8], warnings produced by Coverity Static Analysis and FindBugs are clustered by similarity. The authors show that the grouping significantly reduces the reports that have to be considered. In [9], a set of benchmarks is proposed to evaluate categorization of static analysis reports. Unfortunately, these benchmarks target light-weight analysis and are not comparable to our approach. FindBugs uses a set of detectors that detect a particular category of potential bugs. In a case study in [3] the authors discuss the value of categorizing warnings and its impact in an industrial cases study. While all these papers share our motivation that categorizing static analysis warnings increases usability of tools, their work is based on light-weight analysis tools that notoriously produce many false alarms. We operate on inconsistent code (which is not detected as such by their tools).

8 Conclusion

We have presented an efficient way to categorize inconsistencies in source code. With a small extension to existing inconsistent code detection algorithms, we are able to distinguish three categories `doomed`, `demood`, and `unr` of inconsistent code. Our experiments show that the `doomed` category contains only few, but highly relevant warnings, while the `unr` category contains hardly any critical warning but collects all false alarms. Hence, the proposed approach dramatically increases the usability of inconsistent code detection.

Our experiments indicate that `demood` inconsistent code is rare which would suggest that making a distinction between `demood` and `doomed` inconsistent code is not necessary as this step is relatively costly. However, this observation may be biased by using only master-branches and further experiments on code in active development will be necessary.

For our future work, we have identified that custom assertion libraries like Guava can still introduce false alarms which can easily be avoided by adding very simple contracts. These contracts could be generated automatically, or we could extend our tool to inline procedures up to a certain size.

If nothing else, we have fixed several bugs in open-source projects that run on many web servers that we talk to every day, and thus, we can claim that we have made the world a bit safer.

Acknowledgement This work was supported in part by the National Science Foundation under grant contracts CCF 1423296 and CNS 1423298, and DARPA under agreement number FA8750-12-C-0225.

References

1. S. Arlt, P. Rümmer, and M. Schäf. Joogie: From java through jimple to boogie. In *SOAP*, 2013.
2. S. Arlt and M. Schäf. Joogie: Infeasible code detection for java. In *CAV*, 2012.
3. N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *PASTE*, 2007.
4. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.
5. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 1991.
6. I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *PLDI*, 2007.
7. D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.
8. Z. P. Fry and W. Weimer. Clustering static analysis defect reports to reduce maintenance costs. In *WCRE*, 2013.
9. S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM*, 2008.
10. J. Hoenicke, K. R. Leino, A. Podelski, M. Schäf, and T. Wies. Doomed program points. *FMSD*, 2010.
11. D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE*, 2007.
12. M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *SAVCBS*, 2007.
13. K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 2005.
14. K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 2005.
15. B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 56(1), Feb. 2015.
16. T. McCarthy, P. Rümmer, and M. Schäf. Bixie: Finding and understanding inconsistent code. http://www.csl.sri.com/bixie-ws/, 2015.
17. P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, 2008.
18. A. Tomb and C. Flanagan. Detecting inconsistencies via universal reachability analysis. In *ISSTA*, 2012.
19. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON*, 1999.
20. X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *SOSP*, 2013.