The Gradual Verifier

Stephan Arlt[1⋆], Cindy Rubio-González[2], Philipp Rümmer[3⋆⋆], Martin Schäf[4], and Natarajan Shankar[4⋆⋆⋆]

[1] Université du Luxembourg
[2] University of California, Berkeley
[3] Uppsala University
[4] SRI International

Abstract. Static verification traditionally produces yes/no answers. It either provides a proof that a piece of code meets a property, or a counterexample showing that the property can be violated. Hence, the progress of static verification is hard to measure. Unlike in testing, where coverage metrics can be used to track progress, static verification does not provide any intermediate result until the proof of correctness can be computed. This is in particular problematic because of the inevitable incompleteness of static verifiers.

To overcome this, we propose a *gradual verification* approach, GraVy. For a given piece of Java code, GraVy partitions the statements into those that are unreachable, or from which exceptional termination is impossible, inevitable, or possible. Further analysis can then focus on the latter case. That is, even though some statements still may terminate exceptionally, GraVy still computes a partial result. This allows us to measure the progress of static verification. We present an implementation of GraVy and evaluate it on several open source projects.

1 Introduction

Static verification is a powerful technique to increase our confidence in the quality of software. If a static verifier, such as VCC [6] provides us a proof that a piece of code is correct, we can be sure beyond doubt that this code will not fail for any specified input. If the static verifier fails to compute a proof, we end up with one counterexample. This counterexample may reveal a bug or may be spurious in which case we have to provide annotations to help the verifier. This process is repeated until no new counterexample can be found.

⋆ Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03).
⋆⋆ Supported by the Swedish Research Council.
⋆⋆⋆ This work was supported by NSF Grant CNS-0917375, NASA Cooperative Agreement NNA10DE73C, and by United States Air Force and the Defense Advanced Research Projects Agency under Contract No. FA8750-12-C-0225. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, NASA, US Air Force, DARPA, or the U.S. Government.

Unfortunately, the process of eliminating counterexamples one by one is not suitable for assessing software quality. Certainly, eliminating one bug improves the quality of software, but the static verification does not provide us with any information on how much we have verified already or how many bugs might still be in there.

Ultimately, static verification is incomplete and thus the proof we are looking for might not exist. In this case we end up with nothing. Static verification does only provide yes/no answers, but to obtain partial results manual effort is needed.

Testing, on the other hand, only delivers such partial results in the form of coverage data. Each test case increases the confidence in the application under test. Progress can be measured using different kinds of coverage metrics. That is, from an economic point of view, testing is more predictable. The more time we invest in testing the more we can observe the coverage (and thus our confidence) increase.

In this paper we present a *gradual verification* approach, *GraVy*. Gradual verification is an extension to existing static verification techniques such as VCC [6] or Smack [16] that helps us quantify the progress of the verification. Instead of computing just one counterexample, gradual verification computes an over-approximation of all counterexamples to identify the subset of statements that provably cannot terminate exceptionally anymore. That is, beyond the counterexample indicating that the program is not yet verified, gradual verification gives a percentage of statements that are already guaranteed to be safe.

Gradual verification results can be integrated into existing testing workflows. Each time gradual verification is executed on an application under test, it returns the subset of statements for which it can already prove that exceptional termination is impossible. This provides a metric of progress of the verification process and allows the verification engineer to focus on the remaining statements.

Gradual verification is *not* a new static verification technique. It is an extension that can be applied to any existing static verification techniques to provide additional information to the verification engineer. Thus, issues, such as handling of loops or aliasing are not addressed in this paper. These are problems related to sound verification, but gradual verification is about how to make the use of such verification more traceable and quantifiable.

In gradual verification, we consider programs as graphs, where nodes correspond to a *control location* in the program and edges represent *transition relations* between these control locations. Further, we assume that sink nodes in this graph either are *exceptional sink nodes* where the execution of the program ends exceptionally, or *normal sink nodes* where executions terminate normally.

A statement in a programming language such as Java may be represented by more than one edge if, for example, the statement throws an exception when executed on certain inputs. In order to verify that a statement never terminates exceptionally, we need to show that none of the edges representing this statement goes into an exceptional sink. That is, either the statement is not represented by edge going into an exceptional sink node, or this edge has no feasible execution in the program.

On this graph, we perform a two-phase algorithm: in phase one, we identify all edges that may occur on a feasible execution terminating in a normal sink. For the remaining edges we have a guarantee that they are either unreachable or only occur on feasible executions into exception sinks.

In the second phase we check which of these remaining edges occur on *any* feasible execution. That is, we identify edges that are unreachable and edges that must flow into an exceptional sink. This allows us to categorize program statements depending on the edges that they are represented by: a statement is *unreachable* if it is represented by no feasible edge, *safe* if it is represented only by feasible edges terminating in normal sinks (and reachable), *strictly unsafe* if it is represented only by feasible edges terminating in exceptional sinks (and not unreachable), and *possibly unsafe* otherwise.

That is, a program is safe, if all its statements are safe. However, if we cannot show that all statements are safe, our algorithm still can provide a subset of statements that are guaranteed to be safe, helping the programmer to focus on those parts of the program that still need work. gradual verification can be applied to full programs as well as to isolated procedures. It can be applied in a modular way and also incorporate assertions generated by other tools.

We evaluate an implementation of our gradual verification technique, GraVy, on several large open source projects. Our experimental results show that even using a coarse abstraction of the input program, GraVy can still prove that a large percentage of statements can never throw exceptions.

Related Work. GraVy is based on modular static verification as known from VCC [6], Smack [16], or ESC/Java [12]. These tools translate an application under test "procedure by procedure" into SMT formulas that are valid if this procedure is safe w.r.t. a desired property. A counterexample to this formula can be mapped back to an execution of the (abstract) procedure that violates the property. The problem of these approaches is that they only produce one counterexample at a time which makes it hard to estimate the progress of the verification. To overcome this, GraVy uses techniques that detect contradictions in programs [5,10,19] to identify the subset of statements that never (or always) occur on a counterexample.

Gradual verification can also be compared with symbolic execution techniques as found in program analysis tools like Frama-C [9], Java Pathfinder [14], or Pex [18]. These techniques compute an over-approximation of the set of states from which a program statement can be executed. A program statement is considered safe if this set of states does not contain any state from which the execution of the statement is not defined. Gradual verification can be seen as a lightweight alternative to these approaches: like static verification, it can be applied locally even on isolated code fragments, but it still can identify individual statements that will never terminate exceptionally. Hence, gradual verification does not provide the precision of other symbolic execution techniques, but it is still sufficient to visualize the progress of verification to its user.

Recently, approaches have been presented that generate information during the verification process that go beyond simple yes/no answers. Clousot [8,11],

for example, infers a precondition for each procedure that is sufficient to guarantee the safe execution of this procedure. Compositional may-must analysis, such as [13], can be used to distinguish between possibly and strictly unsafe statements. GraVy can be seen as a lightweight mix of both approaches. It detects a subset of statements that cannot throw exceptions (but does not provide preconditions), and categorizes statements that may, or must throw exceptions (but does not provide the precision of a may-must analysis.)

2 Example

We illustrate our approach with the toy example shown in Figure 1. The Java procedure `toyexample` takes a variable `x` of type `Obj` as input and first sets `x.a` to `1` and then sets `x.b` to `2`. An execution of the first statement `x.a = 1` terminates with a `NullPointerException` if `toyexample` is called with `x==null`. Otherwise it terminates normally. Note that the second statement `x.b = 2` can never throw a `NullPointerException`, because the first statement already ensures that `x` cannot be `null` at this point.

```
1 void toyexample(Obj x) {
2   x.a = 1;
3   x.b = 2;
4 }
```

Fig. 1. Java source code of a toy example.

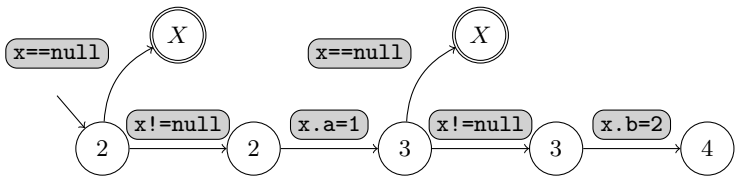

Fig. 2. Program graph of the procedure `toyexample`. Edges are labeled with transition relations, and nodes are labeled with line numbers, where the label X refers to the point in the program that is reached when an exception is thrown.

Suppose that we are interested in verifying that, for any input, the procedure does not terminate with an exception. First, we create a graph representation of our program as shown in Figure 2. In this graph, nodes are labeled with line numbers, where the label X refers to the point in the program that is reached when an exception is thrown. This labeling is simplified for demonstration. Two

different nodes might still share the same line number. Each statement of our original program from Figure 1 is associated with one or more edges in this graph, starting in the nodes labeled with the respective line number. For example, the statement `x.a` in Figure 1 is represented by the three edges in Figure 2 starting in nodes labeled with 2: $(2, \texttt{x==null}, X)$, stating that, if `x` is `null`, the execution terminates exceptionally; $(2, \texttt{x!=null}, 2)$, stating that execution moves on if `x` is initialized; and $(2, \texttt{x.a=1}, 3)$ which is the actual assignment if `x` is initialized.

Now it is time to check if our procedure does not terminate exceptionally. Existing techniques would easily come up with a counterexample to this property that shows that for the input `x==null` the procedure will throw a `NullPointerException`. However, this is a very pessimistic answer, and, given that we do not know in which context `toyexample` will be called, it may even be a useless answer if there is no calling context such that `x==null`. Hence, we propose to give a different, optimistic, answer when checking if our procedure does not terminate exceptionally:

`x.b = 2` never throws an exception.

There are a few things to notice: First, our answer gives proofs instead of a simple counterexample. Second, our answer holds in any context (but might be too weak), whereas the counterexample may turn out to be infeasible. Third, in our answer, our verifier verifies; existing techniques just complain.

To get to this answer, we start a two-phase algorithm. In phase one we try to cover all edges that occur on any feasible path of the program that terminate normally. That is, in our example, we try to find feasible complete paths ending in the sink labeled with 4. One such path exists:

$$(2, \texttt{x!=null}, 2)(2, \texttt{x.a=1}, 3)(3, \texttt{x!=null}, 3)(3, \texttt{x.b=2}, 4)$$

That is, the only two edges that cannot be covered during that process are $(2, \texttt{x==null}, X)$ and $(3, \texttt{x==null}, X)$. For these edges we know that either they are unreachable, or their execution leads to an exceptional termination. Note that in this example, both edges happen to be immediately connected to the error location X. However, in general, there might be other edges, not directly connected to a sink, that can only be executed if normal termination is not possible.

For the uncovered edges we start the second phase of our algorithm, where we try to find *any* feasible complete path. From the first phase, we know that no path through the remaining edges exists that terminates in 4, hence we are only interested in paths terminating in X. For our example, this reveals one more feasible path:

$$(2, \texttt{x==null}, X)$$

That is, the other edge, $(3, \texttt{x==null}, X)$, provably does not have any feasible execution. Now, we have a proof that `x.b=2` in line 3 never throws a `NullPointerException` in `toyexample`. We can further report that `x.a=1` in line 2 may throw a `NullPointerException` if `x` is `null`.

From here, the verification engineer knows that she has to focus on `x.a=1`, and either guard the code with a conditional choice or strengthen the precondition under which `toyexample` can be called. Then, gradual verification can be re-run for the modified code. This is repeated until all statements are safe or a desired percentage of statements is safe.

3 Statement Safety and Gradual Verification

In this section we give a precise definition of our gradual verification methodology. We assume a piece P of sequential program code (in case of Java, the body of a method), containing the set *Stmt* of statements. The control-flow of P can be represented as a finite directed graph $CFG_P = (\mathcal{L}, \ell_0, \mathcal{L}_{exit}, \mathcal{L}_{exc}, \delta, stmt)$, where $\mathcal{L}$ represents control locations, $\ell_0 \in \mathcal{L}$ is the unique entry point, $\mathcal{L}_{exit} \subseteq \mathcal{L}$ is a set of exit locations representing regular termination, and $\mathcal{L}_{exc} \subseteq \mathcal{L}$ is a set of error locations representing termination due to a runtime exception. Further, we assume that $\mathcal{L}_{exit} \cap \mathcal{L}_{exc}$ is empty. An edge $(\ell, Tr, \ell') \in \delta$ is labeled with a transition formula $Tr(\bar{v}, \bar{v}')$ over unprimed and primed variables describing program state.

A statement in our sequential program P is represented by possibly multiple transitions, some of which may lead into error locations $\mathcal{L}_{exc}$. The latter case models runtime exceptions. For instance, a Java statement `a.x = 1` could be translated into two edges: one that assumes that `a` is allocated and `a.x` is assigned to 1, and one where `a` is not allocated and control passes to an appropriate error location. Throughout the paper, we use the partial function $stmt : \delta \rightharpoonup Stmt$ mapping edges to statements in the program code P. Conditional choice of the form `if (c) A else B` is represented by at least two transitions, one assuming c and one assuming $\neg c$, and all other transitions that are necessary to represent c. The transitions representing the blocks `A` and `B` are not considered as part of the conditional choice.

A *complete path* in a program is a finite sequence of control locations and transition formulas $\pi = \ell_0 Tr_0 \ell_1 Tr_1 \ell_2 Tr_2 \dots Tr_{n-1} \ell_n$, where ℓ_0 is the entry point, $\ell_n \in \mathcal{L}_{exit} \cup \mathcal{L}_{exc}$ is an exit location, and for each $i \in \{0, \dots, n-1\}$ it is the case that $(\ell_i, Tr_i, \ell_{i+1}) \in \delta$. A complete path π is called a *regular path* if $\ell_n \in \mathcal{L}_{exit}$, and an *error path* if $\ell_n \in \mathcal{L}_{exc}$. A path is *feasible* if the composition $Tr_0 \circ Tr_1 \circ \dots \circ Tr_{n-1}$ is satisfiable. An edge $(\ell, Tr, \ell') \in \delta$ is called *feasible* if it occurs on a complete feasible path.

We use $\delta_{reg} \subseteq \delta$ to denote the subset of edges that occur on regular paths (i.e., on paths that end in a location in $\mathcal{L}_{exit}$). Further we use $\delta_{bad} = \delta \setminus \delta_{reg}$ to denote all edges that inevitably lead into an error location. With the help of δ_{reg} and δ_{bad}, Figure 3 defines safety categories for a statement $s \in Stmt$ in P considered in gradual verification, which correspond to the four possible combi-

$\exists$ feasible $e \in \delta_{bad}$ with $s = stmt(e)$	$\exists$ feasible $e \in \delta_{reg}$ with $s = stmt(e)$: Yes	$\exists$ feasible $e \in \delta_{reg}$ with $s = stmt(e)$: No
Yes	s is *possibly unsafe*	s is *strictly unsafe*
No	s is *safe*	s is *unreachable*

Fig. 3. Safety categories of statements

nations of regular or error transitions being feasible. For instance, a statement s is considered *safe* if all transitions representing s are in δ_{reg}.

4 The Analysis Procedure

To check whether a statement is *safe*, *strictly unsafe*, *unreachable*, or *possibly unsafe*, we determine for each edge in the control-flow graph whether it can be part of a feasible regular path, and if a statement is represented by a feasible transition into an error location. We introduce Algorithm 1 to this end. The algorithm takes a control-flow graph $CFG_P = (\mathcal{L}, \ell_0, \mathcal{L}_{exit}, \mathcal{L}_{exc}, \delta, stmt)$ as input and returns two sets δ_{bad} and δ_{inf}. δ_{bad} contains all edges of the control-flow graph that do not occur on any feasible regular path. $\delta_{inf} \subseteq \delta_{bad}$ is the set of edges that do not occur on any feasible path (regular, or error paths).

The algorithm uses a local variable S to track the edges in δ that have not been covered yet. In a first loop, Algorithm 1 covers edges that occur on feasible regular paths. That is, all edges that remain in S after the loop terminates can either only be executed on error paths or are unreachable. This set is stored in δ_{bad}. In the second loop, our algorithms checks which of the remaining edges can be covered with feasible error paths and removes them from S. That is, any edge covered in the second loop has a feasible path into an error location. All uncovered edges are stored in δ_{inf} because they have no feasible execution at all.

With the resulting sets δ_{bad} and δ_{inf}, we can check the above properties as follows: given a statement st and the set of edges $\delta_{st} = \{(\ell, Tr, \ell') | (\ell, Tr, \ell') \in \delta \wedge stmt((\ell, Tr, \ell')) = st\}$. The statement st is unreachable if $\delta_{st} \setminus \delta_{inf}$ is empty and δ_{inf} is not empty. It is safe if $\delta_{st} \cap (\delta_{bad} \setminus \delta_{inf})$ is empty and $\delta_{st} \setminus \delta_{inf}$ is not empty. In other words, st is safe if it is not represented by any feasible edge into an error location and has at least one feasible edge. We say, st is strictly unsafe if $\delta_{st} \setminus \delta_{bad}$ is empty and $\delta_{bad} \setminus \delta_{inf}$ is not empty. In any other case, we say st is possibly unsafe.

Algorithm 1 terminates only if the control-flow graph CFG_P has a finite number of paths (i.e., is loop-free). For programs with looping control-flow, abstrac-

Algorithm 1: Gradual verification algorithm.

Input: $CFG_P = (\mathcal{L}, \ell_0, \mathcal{L}_{exit}, \mathcal{L}_{exc}, \delta, stmt)$: control-flow graph
Output: δ_{bad}: set of edges that never occur on feasible regular paths;
δ_{inf}: set of edges that do not occur on any feasible path

```
begin
    S ← δ ;
    for regular path π in CFG_P do
        if isFeasible(π) then
            for (ℓ, Tr, ℓ') in π do
                S ← S \ {(ℓ, Tr, ℓ')} ;
            end for
        end if
    end for
    δ_bad ← S ;
    for error path π in CFG_P do
        if isFeasible(π) then
            for (ℓ, Tr, ℓ') in π do
                S ← S \ {(ℓ, Tr, ℓ')} ;
            end for
        end if
    end for
    δ_inf ← S ;
end
```

tion is necessary. We will discuss one possible abstraction in Section 5 together with other implementation details.

We say an abstraction of a control-flow graph CFG_P is sound, if for any feasible (regular and error) path CFG_P, there exists a corresponding path in the abstraction. That is, an abstraction is sound if it over-approximates the set of feasible control-flow paths.

Given a program CFG_P and an abstraction of it, and, given a statement *st* that exists in the program and its abstraction (but may be represented by a different set of edges in the control-flow graph), the following properties hold if the abstraction is sound:

- If *st* is *safe* in the abstraction then it is *safe* or *unreachable* in CFG_P.
- If *st* is *strictly unsafe* in the abstraction then it is *strictly unsafe* or *unreachable* in CFG_P.
- If *st* is *unreachable* in the abstraction then it is *unreachable* in CFG_P.
- If *st* is *possibly unsafe* in the abstraction then it is *safe*, *strictly unsafe*, *unreachable*, or *possibly unsafe* in CFG_P.

That is, for any sound abstraction, our algorithm guarantees that any statement that may transition into an error location will be declared as either *possibly unsafe* or *strictly unsafe*. Hence, if all statements in our program are either *safe* or *unreachable*, we have a proof that the program will never terminate exceptionally.

To be useful in practice, an implementation of our algorithm has to make sure that it does not report overly many *possibly unsafe* in the abstraction, as we cannot say much about them in the original program. Further, it would be useful if *unreachable* statements in the original program are not reported as *strictly unsafe* in the abstraction. Even though we are of the opinion that unreachable code should be avoided at all cost, a user may be alienated if unreachable code is reported as error. In the following we evaluate our approach.

5 Implementation

We have implemented our technique in a static verifier for Java bytecode called GraVy. Our analysis automatically checks for the following types of exceptions: `NullPointerException`, `ClassCastException`, `IndexOutOfBoundsException`, and `ArithmeticException`. Other exceptions and arbitrary safety properties can be encoded using `RunTimeExceptions`.

An error location in GraVy is an exceptional return of a procedure with one of the above exceptions, unless this exception is explicitly mentioned in the `throws`-clause of this procedure. Hence, if GraVy proves a statement to be safe, it only means that none of the above exceptions may be thrown.

GraVy analyzes programs using the bytecode analysis toolkit Soot [20]. It translates the bytecode into the intermediate verification language Boogie [4] as described in [2]. In this step we add guards for possible runtime exceptions: for each statement that may throw a runtime exception, we add a conditional choice with an explicit throw statement before the actual statement. Further, we add a local helper variable `ex_return` to each procedure which is `false` initially. For any of the exceptions that we are looking for which is not in the `throws` clause and not caught, we add a statement that assigns this variable to `true`. This variable is used later on by the prover to distinguish between normal and exceptional termination of a procedure.

Abstraction. Our algorithm from Section 4 requires a loop-free program as input. Hence, we first need to compute loop-free abstractions of programs. To this end we use a simple loop elimination as discussed in [1]: for each loop, we compute a conservative approximation of the variables that may be modified within the loop body. Then, we add statements that assign non-deterministic values to these variables at the beginning and at the end of the loop body. Finally, we redirect all looping control-flow edges of the loop body to the loop exit.

This way, we simulate an arbitrary number of loop iterations: the non-deterministic assignments allow the loop body to be executed from any possible initial state and allow the variables modified within the loop to have any possible value after the loop. This is a very coarse abstraction which also loses all information about possible non-termination. However, if a statement can be proved safe in this approximation, it will be safe in the original program, as the abstraction over-approximates the program's executions.

GraVy does not perform any inter-procedural analysis. Like in the case of loops, we first compute an over-approximation of the set of variables that may

be modified by the called procedure and then replace the call statement by a non-deterministic assignment to these variables. In our translation into Boogie, exceptions are treated as return values of a procedure and are thus included in this abstraction. Again, this is an over-approximation of the program's executions, and thus, any statement that can be proved safe in this abstraction will be safe in the original program.

All these abstractions can be refined to increase the precision of GraVy.

Gradual Verification. On the loop-free program without procedure calls, we can apply our algorithm from Section 4 to each procedure in a straightforward manner (e.g., [1]) by translating the loop-free program into a SMT formula that is satisfiable only by models that can be mapped to a feasible path in the program. In the first pass of our analysis, GraVy adds an assertion to the SMT formula such that the helper variable `ex_return` is `false`, in order to only allow paths that do not terminate with unwanted exceptions. We use the theorem prover Princess [17] to check for the satisfiability of this formula. For each model returned by Princess, we extract an enabling clause to ensure that another path must be picked in the next query. This process is repeated until the formula becomes unsatisfiable. Then, GraVy pops the assertion that `ex_return` must be `false` and continues until the formula becomes unsatisfiable again.

Using the information obtained during this process, GraVy prints a report for each procedure that pigeonholes its bytecode instructions into the categories unreachable, safe, strictly unsafe, and possibly unsafe as described in Section 3.

Soundness. GraVy is **neither sound nor complete**. Here, *soundness* means that if a statement is reported to be safe, it is always safe. *Completeness* means that any statement that is safe will be reported to be safe. GraVy has several sources of unsoundness: e.g., Java integers are modeled as natural numbers (i.e., over- and under-flows are ignored). Furthermore, we ignore the use of reflection (i.e., `InvokeDynamic`), and we do not consider parallel executions.

However, note that the unsoundness is specific to our prototype implementation. Gradual verification is always as sound as its underlying static verification algorithm. Thus, there is much room for improvement by combining GraVy with more advanced static verifiers.

6 Evaluation

The motivation of gradual verification is to make static verification predictable by providing a progress metric. That is, to be of practical use, GraVy must identify a reasonable percentage of statements to be safe, so that the verification engineer can focus on the remaining code. Further, it must be fast enough to be applicable in an incremental verification process. That is, it must not be significantly slower than existing static verifiers such as VCC. This leads us to the following two research questions:

Q1 Is GraVy precise enough to show that a reasonable percentage of statements are safe in well-tested applications?
Q2 Is GraVy fast enough to be applied to real-world software?

Experimental Setup. To answer these questions we evaluate GraVy on several open source programs. For each application under test (AUT), we analyzed the JAR files of the latest stable (and thus hopefully tested) release from the official websites. All experiments were carried out on a standard notebook with an i7 CPU and 8 GB RAM (the Java VM was started with initially 4 GB). GraVy tried to analyze each procedure of the AUTs for at most 10 seconds. If no result is reached after 10 seconds, the procedure is skipped and a timeout is reported. We ran the analysis two times for each AUT: once with gradual verification, and once with a weakest-precondition-based static verifier [15]. For the weakest-precondition-based static verifier we implemented a simple verifier inside GraVy that reused large parts of the GraVy infrastructure. Instead of repeatedly querying the theorem prover, the static verifier only sends one query. The result to this query is either a proof that no statement in the procedure may throw an exception of the previously mentioned types, or a counterexample that represents an execution of the abstract procedure that leads to exceptional termination.

In addition to the results returned by GraVy about which statements are unreachable, safe, strictly unsafe, or possibly unsafe, we collected the following information: the total time for analyzing a procedure including the time for printing the report, and the total number of procedures for which GraVy returns a timeout.

To compare the gradual static verification with the weakest-precondition-based static verification, we also stopped the time that both approaches spent inside the SMT solver. We compared the time inside the prover rather than actual computation time, because the overhead for both approaches is the same, and thus, the time spent in the prover is the only relevant time difference.

AUT	*# stmts*	*safe*	*# throwing stmts*	*possibly unsafe*	*strictly unsafe*	*unreachable*
Args4j	**2,322**	**2,011**	820	311	0	0
GraVy	**20,372**	**16,522**	15,516	3,844	0	6
Hadoop	**209,683**	**177,373**	109,758	32,249	7	54
Log4j	**25,128**	**22,381**	11,007	2,746	0	1

Table 1. Results of applying GraVy to several AUTs. *# stmts* is the number of analyzed statements per AUT. *# throwing stmts* is the number of statements that is represented by at least one edge into an error location.

Discussion. Table 1 shows the report computed by GraVy for each AUT. By comparing the columns *# stmts* and *safe*, GraVy is able to prove more than

80% of the analyzed statements to be safe for all AUTs. For *Args*4*j* and *Log*4*j*, GraVy can even prove over 86% percent of the statements to be safe. If we only consider the statements that are represented by edges into an error location (i.e., by comparing the columns *# throwing stmts* and *possibly unsafe*), GraVy proves 62% of the statements to be safe in *Args*4*j*, and 75% in *Log*4*j*.

For *Hadoop*, which is also widely used and well-tested, we only achieve 84% to be safe (and 70% of statements represented by edges into error locations). This is because Hadoop makes heavy use of multithreading which is not handled by GraVy. The use of multithreading is also the cause of the reported unreachable and strictly unsafe statements. None of these statements is actually unreachable or strictly unsafe, they rather exhibit situations where a thread is waiting for another thread to initialize an object.

GraVy applied to itself can only prove 81% to be safe (and 75% of the statements represented by edges into error locations). This supports the idea that the percentage of safe statements relates to the maturity of the code: GraVy is currently under development and represents a rather prototypical implementation.

Hence, we can give a positive answer to our research question **Q1**. GraVy can, even on a coarse abstraction, prove a large percentage of statements safe. Further, experiments indicate that the percentage of safe statements may correlate with code quality.

What remains open is what useful thresholds for the percentage of safe statements are. Many statements in Java bytecode can never throw any of the considered exceptions and thus are always safe. Therefore it is hard to define a lower bound for the percentage of safe statements. Our experiments cannot say anything about an upper bound either, because we did not try to improve the AUTs and rerun GraVy as this would exceed the scope of this paper. For the future of GraVy we plan a case study on how to apply gradual static verification, e.g., by using specification languages such as JML [7].

AUT	*# procedures*	*time (s)*	*time per procedure (s)*	*# timeouts*	*removed exceptions*
Args4j	361	57	**0.16**	2	6.7%
GraVy	2,044	668	**0.33**	33	8.3%
Hadoop	18,728	6,459	**0.34**	391	8.4%
Log4j	3,172	704	**0.22**	40	13.0%

Table 2. Performance and number of timeouts of GraVy on the different AUTs. The column # timeouts states the number of procedures that could not be analyzed within the time limit. The last column states the percentage of exceptions that could be removed using constant propagation.

Table 2 shows the performance results of our experiments. For all AUTs the average time needed per method is significantly below one second ($< 0.4s$). The number of procedures that reach a timeout is below 2.1% for all AUTs. Experimenting with timeouts larger than 10 seconds did not significantly improve this

number. Most procedures that timeout contain large amounts of initialization code (e.g., constructors), or GUI related code.

Before running gradual verification we run a constant propagation to eliminate all possible exceptions that can be ruled out trivially. We are able to eliminate between 6.7% and 13.0% of the exceptions. That is, a significant percentage of the statements proved safe by GraVy need non-trivial reasoning.

In summary, GraVy can produce meaningful results within a reasonable time (less than 0.4s per procedure) and with few timeouts.

	Args4j	*GraVy*	*Hadoop*	*Log4j*
GSV	45s	346s	4,425s	294s
SV	7s	33s	689s	41s
Safe	32.6%	40.8%	40.2%	40.3%

Table 3. Theorem proving time for gradual static verification (GSV) and weakest-precondition-based static verification (SV) for the AUTs. Only the time spent in the theorem prover is measured as the overhead for transformation, etc. is identical for both approaches. The last row states how many procedures can be proven safe by both approaches (i.e., only contain safe statements).

Table 3 compares the computation time of GraVy and a normal non-gradual verifier. For this purpose we built our own weakest-precondition based static verification following the idea from [15]. As both approaches require the same program transformation, we only compare the time spent by the theorem prover.

The first row shows the theorem proving time for gradual verification (GSV), the second row shows the theorem proving time for non-gradual verification (SV), and the last row shows the percentage of the procedures that can be proven safe by both approaches (i.e., procedures that only contain safe statements). For each AUT, the extra time needed for gradual static verification is less than a factor of 10. Most procedures still can be analyzed within few seconds. We believe that, by further improving the reuse of theorem prover results as suggested in [3], we can reduce these extra costs even further.

Non-gradual verification alone is able to verify between 30% and 40% of the procedures (excluding timeouts) for the desired property for all AUTs. Most of these procedures are generated by the Java compiler, such as default constructors. For all remaining procedures, non-gradual verification only returns a counterexample. Here, gradual static verification provides additional information by ruling out those statements that are already safe within the remaining procedures.

In conclusion, we can also give a positive answer to research question **Q2**: GraVy takes less than a second per procedure for real-world software. Although it is (naturally) slower than non-gradual verification, it is still fast enough to be usable in practice.

Threats to Validity. The main and by far most important threat to validity is our unsoundness. For example, in Hadoop, where we prove 177,373 statements

to be safe, it is not possible to manually inspect if they are subject to unsoundness or not. To get a sense of how our abstraction affects the precision of GraVy, we investigated roughly a hundred statements from different AUTs and different categories (i.e., safe, unreachable, etc.). We found several cases where code was reported to be unreachable or strictly unsafe in the abstraction but safe in the original program. We did not find statements that are reported safe in the abstraction but unsafe in the original program.

Another threat to validity is the subset of exceptions that we consider in our analysis. There are many more exceptions that can cause unexpected program behavior. However, from manual data-flow inspection we can see that `NullPointerException` is by far the most common exception that can be thrown. Thus, we believe that adding more classes of exceptions to GraVy will certainly increase the usefulness of our approach, but will only have a limited influence on the results presented in this paper.

Finally, the used tools are a threat to validity. Using Java bytecode as input allows us to use a much simpler memory model than, e.g., for C. It is not clear if our approach can be applied to C programs equally well. For example, we allow arbitrary aliasing between variables when analysing a procedure. This would, most likely be too coarse for analyzing C programs and further analysis would be required.

7 Conclusion

We have presented a technique for gradual static verification. Gradual verification extends existing static verification which provides yes/no answers (i.e., either a proof or a counterexample) by a notion of the verification progress. That is, even if a full correctness proof is impossible (e.g., because there are some cubic formulas in the code), we can still report how many statements can be "verified".

Gradual verification blends nicely with existing best practices in testing, where a test coverage metric is used to measure progress, and to decide when to stop testing. Therefore, we believe that gradual verification can make the use of formal methods in industrial software development more acceptable.

Our experiments show that GraVy is reasonably fast and that it can already prove a convincingly high percentage of statements to be safe, even using a coarse abstraction. Further, the experiments indicate that verification coverage may be a good indicator for the maturity of code.

We are convinced that gradual static verification is a useful addition to existing static verification tools and a nice and cheap alternative to verifiers based on symbolic execution such as Frama-C.

References

1. S. Arlt, Z. Liu, and M. Schäf. Reconstructing paths for reachable code. In *ICFEM*, pages 431–446, 2013.
2. S. Arlt, P. Rümmer, and M. Schäf. Joogie: From java through jimple to boogie. In *SOAP*. ACM, 2013.
3. S. Arlt, P. Rümmer, and M. Schäf. A theory for control-flow graph exploration. In *ATVA*, pages 506–515, 2013.
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*, pages 364–387, 2005.
5. C. Bertolini, M. Schäf, and P. Schweitzer. Infeasible code detection. In *VSTTE*, pages 310–325, 2012.
6. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, pages 23–42, 2009.
7. D. R. Cok. Openjml: Jml for java 7 by extending openjdk. In *NFM*, 2011.
8. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, pages 128–148, 2013.
9. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c - a software analysis perspective. In *SEFM*, pages 233–247, 2012.
10. 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.
11. M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, 2011.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. *SIGPLAN Not.*, pages 234–245, 2002.
13. P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, pages 43–56, 2010.
14. S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.
15. K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, pages 281–288, 2005.
16. Z. Rakamaric and A. J. Hu. A scalable memory model for low-level code. In *VMCAI*, pages 290–304, 2009.
17. P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, 2008.
18. N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/SIGSOFT FSE*, pages 253–262, 2005.
19. A. Tomb and C. Flanagan. Detecting inconsistencies via universal reachability analysis. In *ISSTA*, pages 287–297, 2012.
20. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON 1999*, pages 125–135, 1999.