Reconstructing Paths for Reachable Code

Stephan Arlt, Zhiming Liu, and Martin Schäf

United Nations University, IIST,
Macau S.A.R., China.
{arlt,lzm,schaef}@iist.unu.edu

Abstract. Infeasible code has proved to be an interesting target for static analysis. It allows modular and scalable analysis, and at the same time, can be implemented with a close-to-zero rate of false warnings. The challenge for an infeasible code detection algorithm is to find executions that cover all statements with feasible executions as fast as possible. The remaining statements are infeasible code. In this paper we propose a new encoding of programs into first-order logic formulas that allows us to query the non-existence of feasible executions of a program, and, to reconstruct a feasible path from counterexamples produced for this query. We use these paths to develop a path-cover algorithm based on blocking clauses. We evaluate our approach using several real-world applications and show that our new prover-friendly encoding yields a significant speed-up over existing approaches.

1 Introduction

Recently, static verification techniques are being used to prove the existence of *infeasible code* [**?**,**?**,**?**,**?**]. These techniques prove the existence of statements that do not occur on any feasible (complete) control-flow path in a program. Such statements could be unreachable code, a null-check of memory that has already been accessed, or a guaranteed violation of a run-time assertion. Infeasible code is an interesting target for static verification: proving the absence of feasible executions can be done on a code snippet in isolation without knowing its context, and the proof still holds if the context is extended, which allows scalable implementations at a close-to-zero false positives rate.

To detect infeasible code, one has to prove that any complete path containing a particular statement is infeasible. This is usually done by computing a first-order logic formula of (an over-approximation of) the weakest-liberal precondition of the set of paths containing this statement with respect to the empty post-state. Then a theorem prover is used to check if this formula is valid. If the proof succeeds, then there is no terminating execution along any of these paths.

To check if there is *any* infeasible code in a program, we have to repeat this check for every statement. Of course, this is very inefficient. Several optimizations are possible. E.g., the existence of a feasible complete control-flow path containing a particular statement immediately implies that all other statements on this path have a feasible execution as well. However, such optimizations require that counterexamples from the theorem prover can be mapped to feasible executions.

The problem we are addressing in this paper is the following: If we query the theorem prover with the weakest-liberal precondition formula of a code snippet, the counterexample we receive only states the existence of *at least one* feasible path through this snippet, it, in general, does not allow us to reconstruct any particular path. This is, because the query we sent to the theorem prover allows the prover to find a satisfying valuation that represents several program executions, but, as the program we encoded into the formula is deterministic, we cannot match this valuation to one execution in the input program. Hence, the formula has to be augmented to force the prover to find a valuation that can be mapped to exactly one deterministic execution.

In this paper, we propose a new encoding of code into logic formulas for infeasible code detection that allows us to identify a feasible path (actually, a path that we cannot prove infeasible) immediately from a counterexample of the theorem prover. Based on this encoding, we propose an algorithm to detect all statements for which the prover cannot find a feasible execution. We show how the new encoding can be used to develop much faster tools for infeasible code detection.

Related Work. The problem of reconstructing information about error traces from counterexamples in static verification was first discussed by Leino et al. [**?**]. They introduce so-called labels which are emitted by the theorem prover if a proof of correctness fails. These labels refer to particular assertions that could not be proved correct, and allow the reconstruction of an error trace. The motivation of this approach is essentially the same as ours, but their technique cannot be applied to infeasible code detection, as it only works for weakest precondition encodings with failing assertions. In the case of weakest liberal precondition, no label would be emitted by the prover.

There is a lot of related work on infeasible code detection. Under different names, infeasible code detection has been proposed in many papers. Probably the most significant work is the paper by Engler [**?**] which also forms the basis of the static analysis in Coverity's Prevent tool. In this work, infeasible code is called *contradicting believes* and detected using syntactic pattern matching. Findbugs [**?**] uses a similar pattern matching to identify a subset of infeasible code. In [**?**], infeasible code is called (sequential) semantic inconsistency or source-sink errors. They further detect non-sequential semantic inconsistencies which are not infeasible code. None of these approaches uses static verification based encodings of programs, so the problem and solution described in this paper do not apply there.

Different approaches have been presented that use weakest-liberal precondition encodings of programs for infeasible code detection. Janota et al. [**?**] present an approach to detect unreachable code in the presence of logic specifications. They only focus on the subset of infeasible code that is not forward reachable. To reduce the number of queries, they make use of the dominator relationship between statements in the control-flow graph to identify a minimal subset of statements that have to be checked. In [?] and [?], an approach to detect infeasible code is presented that uses auxiliary Boolean variables and the dominator and post-dominator relation on control-flow locations to minimize the number of theorem prover queries. In [?] and [?] the term infeasible code is introduced. There, we used auxiliary integer variables and an *effectual set* to define a query optimal algorithm to detect infeasible code. Another approach for infeasible code detection is presented in [?], which refers to it as *fatal code*.

All of the above approaches that detect infeasible code using static verification see the problem of finding infeasible code as a coverage problem of the control-flow graph. They try to find an optimal coverage strategy to find all feasible paths and use a theorem prover as an oracle for feasibility queries. They propose different strategies based on helper variables to send more informed queries to the prover and thus reduce the overall number of queries. In this paper, however, we believe that a theorem prover is too complex to be treated as a blackbox: we assume that helper variables that constraint the theorem prover may refrain the prover from building useful knowledge and thus make each query more expensive.

2 Examples

Infeasible code is an interesting target for static analysis. First of all, it can be detected without too much noise (i.e., false positives), and second, infeasible code is usually a good indicator for security vulnerabilities as it shows the existence of code that cannot be executed or fails inevitably. And, most importantly, infeasible code occurs in practice, and it is not rare, as shown in our experiments in Section 6. We motivate the usefulness of infeasible code detection using two instances of infeasible code in Fig. 1 that we found in the software used for the German eID[1]. Both examples are not necessarily bugs, but they show problems in the error model of the applications. In the first example, the `equals` procedure compares the `.bases` field of two objects, by first checking if the one on the left hand side is `null` and the other one is not. If so, it returns `false` otherwise it compares the size of the `bases`. Now, consider the case that `bases` and `other.bases` are `null`. In that case, the `else`-branch of the `if` statement in line 4 is executed. And, on any execution, where this `else`-branch is executed we cause a `NullPointer` exception in line 8. Hence, the `else`-branch of the conditional in line 4 is infeasible code (and a bug if we can find a test case that executes this code).

The second example is even more obvious: if `i` is bigger than `(len-1)`, the procedure throws an exception at line 3. Hence, the return statement in line 7 can never be reached and thus is infeasible code. Even though, no error occurs, it makes a significant difference if a method returns normal or with an exception, thus we assume a conceptual flaw in the error model that we have yet to confirm with the developers.

[1] `http://www.openecard.org/`

```java
// org.openecard.bouncycastle.crypto.params.
    NTRUSigningPrivateKeyParameters
public boolean equals(Object obj) {
    if (bases == null) {
        if (other.bases != null) {
            return false;
        }
    }
    if (bases.size() != other.bases.size()) {
        return false;
    }
}
```

```java
// org.openecard.bouncycastle.pqc.math.linearalgebra.
    GF2Polynomial
public void xorBit(int i) throws RuntimeException {
    if (i < 0 || i > (len - 1)) {
        throw new RuntimeException();
    }
    if (i > (len - 1)) {
        return;
    }
    value[i >>> 5] ^= bitMask[i & 0x1f];
}
```

Fig. 1: Two examples of infeasible code taken from the German eID software.

In both cases, the infeasible code is not a bug, but it shows problems in the error model, as there is some error handling code, but still there are cases which are not handled or handled multiple times. Other examples can be found using our tool, Joogie [?]. Usually, infeasible code in large methods tends to be more interesting but is not suitable to be presented in a paper.

3 Preliminaries

Throughout this paper, we only consider programs written in the simple unstructured language given shown in Figure 2. The language can be seen as a simplified version of Boogie [?] which is sufficient for demonstration purposes. The language is simple but yet expressive enough to encode high-level languages such as Java. In our experiments in Section 6, we use the Joogie tool [?] to translate Java programs into this language.

We represent executions of statements in our language by pairs of states. A state s is a function that maps program variables to values of appropriate sort. We use $s(x)$ to denote the value of a variable x at the state s. We use the weakest precondition to describe the semantics of the statements in our

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

Fig. 2: The syntax of our simple (unstructured) Language

language. Given a statement st, and two states s, s', we say that s followed by s' is an execution of st if and only if $s \models wp(st, s')$. Furthermore, we use the weakest-liberal precondition to check if a statement has no execution at all: a statement has no execution if the formula $\models wlp(st, false)$ is valid (for brevity, we use the empty set of states and the Boolean $false$ interchangeably, which is not really clean but safes a lot of writing). That is, a statement has no execution if, for any pre-state s it's execution ends in $false$ (which is not possible), or does not terminate (see definition of wlp).

A path is a sequence of statements $\pi = st_0; \ldots; st_{n-1};$ connected by sequential composition, an execution of π is a sequence of states $s_0 \ldots s_n$ such that for any $0 \leq i < n$, $s_i \models wp(st_i, s_{i+1})$, and in particular $s_0 \models wp(\pi, s_n)$. Hence, a path π has no execution, if $wlp(\pi, false)$ is valid. For brevity, we treat statements connected by sequential composition as sequences of statements and omit the semicolon if possible. We say a path is feasible if it has at least one execution and that it is infeasible otherwise.

st	$wlp(st, Q)$	$wp(st, Q)$
assume E	$E \implies Q$	$E \implies Q$
assert E	$E \implies Q$	$E \wedge Q$
$\mathit{VarId} := \mathit{Expr}$	$Q[\mathit{Expr}/\mathit{VarId}]$	$Q[\mathit{Expr}/\mathit{VarId}]$
$S; T$	$wlp(S, wlp(T, Q))$	$wp(S, wp(T, Q))$
$goto\ S_0 \ldots S_n$	$\bigwedge_{0 \leq i \leq n} wlp(S_i, Q)$	$\bigwedge_{0 \leq i \leq n} wlp(S_i, Q)$

Fig. 3: The weakest (liberal) precondition semantic of our language from Figure 2.

We extend the computation of wp and wlp from paths to programs in the obvious way. We use the standard approach to compute a formula representation of the weakest-liberal precondition shown in Figure 3. For a more detailed description of this encoding which includes language features such as procedure, we refer to [**?**,**?**].

To show that a statement st has no execution within a program P, we simply have to show that each complete path π in P that contains st has no execution. Here, a complete path is a path of P that starts in a unique initial statement and ends in a unique final statement. Throughout the rest of the paper the term path always refers to a complete path unless stated different.

Definition 1. *Given a statement st in a program P. The statement st is infeasible in P if, for any complete path π in P that contains st, the formula $wlp(\pi, false)$ is valid.*

Here, a program could be a real program, a procedure, or simply a set of related paths. For simplicity, we use the term program.

Computing a formula representation of the weakest-liberal precondition that can be understood by a theorem prover usually requires some sort of abstraction. In general, looping control-flow and the type system of high-level programming languages cannot be encoded into a first-order logic formula that can be solved by an automated theorem prover. Hence, abstraction is necessary. Such an abstraction may include elimination of looping control-flow, an approximation of finite programming language types by infinite logic types, etc. The details of such an abstraction are not in the scope of this paper and we refer to the related work for more details (e.g., [**?**,**?**]).

From here on, we assume that we have an abstraction $P^{\#} = abstract(P)$ that, for a given program P, provides us with an abstraction $P^{\#}$ of P that satisfies the following properties: a) we can compute a formula representation of $wlp(P^{\#}, false)$ that is decidable by the decision procedure of our choice, b) $P^{\#}$ has only finite paths (i.e., is loop-free), and c) $P^{\#}$ is a sound abstraction of P. That is, if we can prove that $P^{\#}$ has no execution (by showing that $wlp(P^{\#}, false)$ is valid), then P does not have an execution either. Formally, we define the soundness of the abstraction as follows:

Definition 2 (Sound abstraction). *Given a program P and an abstraction $P^{\#} = abstract(P)$. The program $P^{\#}$ is a sound abstraction of P, if there exists a mapping of paths in $P^{\#}$ to pahts in P such that each feasible path π can be mapped to a feasible path $\pi^{\#}$ in $P^{\#}$.*

We emphasize that this notion of soundness is different from soundness in verification, where the abstraction has to preserve the infeasible executions instead of the feasible ones. Implementations of such abstractions are, for example, presented in [**?**,**?**,**?**].

Given such an abstraction, we are able to check if a program P has a feasible execution by asking a theorem prover if there is a valuation s such that $s \not\models wlp(P^{\#}, false)$. Now, it would be nice if we could obtain a feasible execution of $P^{\#}$ from s. Unfortunately, and this is the main motivation of this paper, this is not possible. Our query checks for the non-existence of a feasible path in $P^{\#}$, a counterexample to this can be an arbitrary number of feasible paths. For the theorem prover, it may be sufficient to find values for a few program variables to satisfy $wlp(P^{\#}, false)$. All remaining variables are then assigned to arbitrary values. Hence, for the general case, s does not represent any particular feasible path. However, an efficient implementation that detects infeasible code needs this information to cover all feasible paths in $P^{\#}$ (because all statements that cannot be covered are infeasible in P). In the following we present a new encoding of wlp

that allows us to extract exactly one execution and its corresponding control-flow path in $P^{\#}$ from a counterexample for $wlp(P^{\#}, false)$. Based on this encoding, we further propose an algorithm to detect all infeasible statements in the original program.

4 Encoding of the weakest-liberal precondition

Computing the formula representation of the weakest (liberal) precondition of our abstract program $P^{\#}$ is straight forward and has been discussed in many previous articles (e.g., [?,?,?,?]). We avoid the exponential explosion of the formula's size that comes with branching, by introducing auxiliary variables, which we call *block variables*. Using these variables avoids copying the *wlp* of the successor blocks. For each block

$$Block_i ::= \ell_i : S_i ; \textbf{goto}\, Succ_i$$

we introduce a variable b_i that represents the formula $\neg wlp(Block_i, \mathsf{false})$, where $Block_i$ is the basic block at label ℓ_i. These variables can be defined as

$$\begin{aligned} WLP : \bigwedge_{0 \le i < n} b_i &\implies \neg wlp\Big(S_i, \bigwedge_{j \in Succ_i} \neg b_j\Big) \\ \wedge\, b_n &\implies \neg wlp(S_n, \mathsf{false}). \end{aligned}$$

Where B_n denotes a unique exit block of the program. We can now find an execution of our program $P^{\#}$ starting from its initial location ℓ_0 by asking the theorem prover of our choice to find a satisfying valuation for

$$WLP \wedge b_0$$

Note that, unlike in the previous section, we use the negated weakest-liberal precondition. That is, we say that $P^{\#}$ has an execution if there exists a state s such that $s \models wlp(P^{\#}, false)$. From a logic point of view, both ideas are the same, but for the theorem prover finding a satisfying valuation is usually easier.

Lemma 1. *There is a satisfying valuation s for the formula WLP with $s(b_i) = \mathsf{true}$ if and only if there exists an execution for the program fragment starting at the block $Block_i$.*

Proof is given in [?]. A satisfying valuation s of $WLP \wedge b_0$ corresponds to the existence of an execution of the program fragment. Moreover if $s(b_i)$ is true, the same valuation also corresponds to an execution starting at the block $Block_i$. However, it does not mean that there is an execution that starts in the initial state, visits the block $Block_i$, and then terminates. This is because the formula does not encode that $Block_i$ is reachable from the initial state.

To overcome this problem one may use the strongest postcondition to compute the states for which $Block_i$ is reachable. This roughly doubles the formula.

In our case there is a simpler check for reachability. Based on the auxiliary variables that we already introduced to encode the weakest-liberal precondition, we encode the forward reachability as follows: let Pre_i be the set of predecessors of $Block_i$, i.e., the set of all j such that the final **goto** instruction of $Block_j$ may jump to $Block_i$. Then we can encode that a block has to be also forward reachable on a satisfying assignment as follows:

$$VC : WLP \wedge b_0 \wedge \bigwedge_{1 \leq i \leq n} \left(b_i \implies \big(\bigvee_{j \in Pre_i} b_j \big) \right).$$

That is, like in the case of WLP, given a valuation s such that $s \models VC$, $s(b_0)$ is *true* if there exists a complete and feasible path. Further, by requiring that $s(b_i)$ can only be *true* if this also holds for at least one of its predecessors, $s(b_i)$ can only be *true* if it occurs on a complete and feasible path. Thus, we can reconstruct a feasible path through our program by collecting all statements in blocks whose block variables evaluate to *true*.

Theorem 1. *There is a valuation s that satisfies VC with $s(b_0) = true$ if and only if s gives rise to the execution of a complete path π. Moreover, the value of any block variable $s(b_i)$ is true if and only if there is an execution of a path π starting in s that visits block $Block_i$.*

Proof (Sketch). The proof trivially follows by induction: from Lemma 1, we already know that $s(b_0)$ is *true* if and only if there exists a complete feasible path through the program fragment. For b_1, however, the implication $b_1 \implies (\bigvee_{j \in Pre_i} b_j)$ requires that b_1 can only be *true* if there exists at least one predecessor that is also *true* (here, it can only be b_0). Further, by Lemma 1, b_1 can only be *true* if there is a feasible execution of the program starting in b_1. As our input programs are deterministic, we also know that there can only be one predecessor $block_j$ of a block $block_i$, such that $s(b_j) = true$. Hence, by induction it follows that for a valuation $s \models VC$, the valuation $s(b_i)$ is *true* if and only if b_i has a feasible prefix and suffix path and thus is on a feasible and complete path.

The encoding of VC is the main contribution of this paper: similar to WLP, it allows us to check for the non-existence of a feasible path in our program P. But, in addition to that, a counterexample of VC also provides us a feasible path in $P^{\#}$ as a witness. The major benefit of our encoding over existing approaches is that VC does not introduce additional variables (besides the ones introduced by WLP). In our experiments, we will show that this encoding allows significantly faster algorithms than existing approaches.

Now, to identify infeasible code in the original program P, we have to identify the subset of statements in $P^{\#}$ which do not occur on feasible paths. For that, in the following section, we show an algorithm to detect infeasible code using our new encoding by gradually excluding feasible paths from $P^{\#}$ until VC becomes unsatisfiable.

5 Covering algorithm

With the encoding from the previous section, each time we obtain a valuation $s \models VC$ from the theorem prover of our choice, we can identify a feasible path in $P^{\#}$ by checking the valuation of each reachability variable b_i. Now, to find all statements that occur on feasible paths, we want to make sure that the next time we ask our prover for a satisfying assignment of VC it provides us with a s' that executes a different path. In the following, we propose an algorithm to achieve this by using *enabling clauses*, which force the prover to set at least one b_i to true, that has not been true before.

Enabling Clauses. Each time we query our prover, we want to further restrict our formula to those valuations that represent feasible paths of previously uncovered blocks. Therefore, we propose algorithm *EnblClause* in Algorithm 1 that uses *enabling clauses*. An enabling clause is the disjunction of all block variables that have not been assigned to true by previous satisfying valuation of the reachability verification condition.

Algorithm 1: *EnblClause*

```
 Input: $VC$: A reachability verification condition,
        $\mathcal{B} = \{b_0, \ldots, b_n\}$: The set of block variables
 Output: $\mathcal{I}$: The set of block variables that do not have feasible executions.
 1 begin
 2     $\mathcal{I} \leftarrow \mathcal{B}$
 3     $s \leftarrow$ checksat$(VC)$
 4     while $s \neq \{\}$ do
 5         $\phi \leftarrow$ false
 6         foreach  $b_i$ in $\mathcal{I}$ do
 7             if $s(b_i) =$ true then
 8                 $\mathcal{I} \leftarrow \mathcal{I} \setminus \{b_i\}$
 9             else
10                 $\phi \leftarrow \phi \vee b_i$
11             endif
12         endfch
13         $s \leftarrow$ checksat$(VC \wedge \phi)$
14     endw
15     return $\mathcal{I}$
16 end
```

The algorithm takes as input a reachability verification condition VC, and the set of all block variables $\mathcal{B}$ used in this formula, and returns the set of block variables which cannot occur on any feasible complete path. The algorithm uses the prover `checksat` (lines 3,13) which takes a formula ϕ as input and returns a valuation $s \models \phi$ if ϕ is satisfiable or the empty set, otherwise. First, our algorithm sets the set of infeasible block variables $\mathcal{I}$ to the set of all block variables $\mathcal{B}$ (line 2). Then, it checks if there exists any satisfying valuation s for VC (line 3). If so, the algorithm removes all b_i from $\mathcal{I}$ which evaluate to `true` in s, as those occur on a feasible path (line 8). The block variables which do not evaluate to `true` in s are added to the enabling clause (line 10) to ensure that `checksat` will evaluate at least one of them to `true` in the next iteration. The algorithm terminates if all blocks have been visited once (and therefore, the enabling clause ϕ becomes $false$), or if there is no feasible execution passing the remaining blocks.

Theorem 2 (Correctness of *EnblClause*). *Given a (abstract) program P with reachability verification condition VC. Let $\mathcal{B}$ be the set of block variables used in VC. Algorithm EnblClause, started with the arguments VC and $\mathcal{B}$, terminates and returns a set of block variables $\mathcal{I}$ for which no feasible execution exists.*

Proof. In every iteration of the loop at least one variable of the set $\mathcal{I}$ will be removed. This is because the formula ϕ will only allow valuations such that for at least one $b_i \in \mathcal{I}$ the valuation $s(b_i)$ is true. Since $\mathcal{I}$ contains only finitely many variables the algorithm must terminate. If π is a feasible path visiting the block associated with the variable b_i, then there is a valuation s that satisfies VC with $s(b_i) = true$. Such a valuation must eventually be found, since $VC \wedge \phi$ is only unsatisfiable if $b_i \notin \mathcal{I}$.

Hence, *EnblClause* is a sound and complete way to detect infeasible code for loop-free programs given a complete implementation of the decision procedure `checksat`. In practice, of course, infeasible code detection is not complete as the computation of a loop-free program requires abstraction and decision procedures are usually not complete.

What we have presented so far is an encoding of loop-free programs into formulas that allows us to reconstruct feasible executions of this (abstract) program from a satisfying valuation of the formula. Based on this, we have presented an algorithm to detect all blocks in a program that do not have feasible executions (in the original program). The question now is, if this approach allows the theorem prover to discover infeasible code more efficiently than existing approaches.

Optimization. To check if there is a feasible execution for each block, it is not necessary to include all block variables in the enabling clause. We follow the idea of [**?**] and compute an *effectual set* of blocks which is sufficient to find at least one feasible execution for each basic block. For that we proceed as follows: we define a relation $\preceq$ on basic blocks, such that $b_i \preceq b_j$ if every complete path that contains b_i also contains b_j. The relation $\preceq$ can be easily constructed as a combination of dominator and post-dominator relation. As $\preceq$ is reflexive and transitive, we can define an equivalence relation $\simeq$ as $\simeq = \preceq \cap \preceq^{-1}$. We denote by $[B]$ the equivalence class of blocks B under $\simeq$. The elements of $[B]$ blocks that only appear together on a path. The partial order $\preceq$ is extended from blocks to equivalence classes of blocks as expected: $[B] \preceq [B']$ if and only if $B \preceq B'$.

Under $\preceq$, an equivalence class $[B]$ which is minimal contains blocks that only occur on paths containing (all) elements of $[B]$. Hence, finding a feasible execution for each block is equivalent to finding one execution for one element of each minimal equivalence class (see [?] for a proof). In the following we call a set *effectual* if it contains exactly one element of each minimal equivalence class.

Hence, applying *EnblClause* to an effectual set of blocks gives us the set of all infeasible blocks in the effectual set. From there, we can look up the set of all infeasible blocks from the Hasse diagram that is given by $\preceq$ (e.g., [?]).

6 Experiments

The question we are trying to answer is *does the new encoding allow us to detect infeasible code faster than existing approaches?* For that, we compare four different approaches: our approach from *EnblClause* only applied to an effectual set of block variables (**ExtWlp**); *EnblClause* applied to all block variables (**EnablingClause**), an approach similar to ours that uses *blocking clauses* instead of enabling clauses presented in [?] (**BlockingClause**); and the algorithm from [?] that injects integer variables into the program and uses assertions of linear inequalities to implement a query optimal algorithm (**OptimalCover**).

Experimental Setup. We evaluate our approach on six open-source applications under test (AUTs): Open eCard, a software to support the German eID, the CASE tool ArgoUML, the mind-mapping tool FreeMind, the time-keeping software Rachota, the word processor TerpWord, and the software that we used to analyze these programs, Joogie [?]. Table 1 gives an overview of our AUTs, including lines of code, number of analyzed procedures, and detected infeasible statements.

Program	LOC	# checked methods	# found
Open eCard	456,220	15,654	26
ArgoUML	156,294	9,981	28
FreeMind	53,737	5,613	10
Joogie	11,401	973	0
Rachota	11,037	1,279	1
TerpWord	6,842	360	3

Table 1: Results of applying Joogie to the test applications.

For comparison, we have implemented all four algorithms in Joogie[2]. Joogie provides the necessary abstraction of Java programs into loop-free programs that can be translated into logic formulas. Loops are abstracted by redirecting the back-edge of a loop to the loop exit and adding non-deterministic assignments to all variables modified inside the loop to the loop entry and exit.

[2] `http://www.joogie.org/`

Furthermore, Joogie injects run-time assertions for `null` de-reference, array-bound violations, and division by zero. Joogie applies the infeasible code detection to each procedure in isolation. I.e., it does not perform inter-procedural analysis. Calls to procedures are replaced by non-deterministic assignments to all variables that could be modified by the called function.

We use the same abstraction for each experiment, and, since all four algorithms are complete for abstract programs, the detection rate is the same in each case. Thus, we only have to compare the computation time.

All experiments are run on a workstation with 3 GHz CPU, 8 GB RAM, and 640 GB HDD. To avoid bias by the employed theorem prover, we run our experiments with Princess [**?**] (which is the standard prover in Joogie), and Z3 [**?**]. Each procedure is analyzed for at most 30 seconds. If the algorithm is not able to analyze the whole procedure within this time, we kill the prover and start over with the next procedure. Only the time spent inside the prover is stopped to eliminate noise that may be introduced by our implementation.

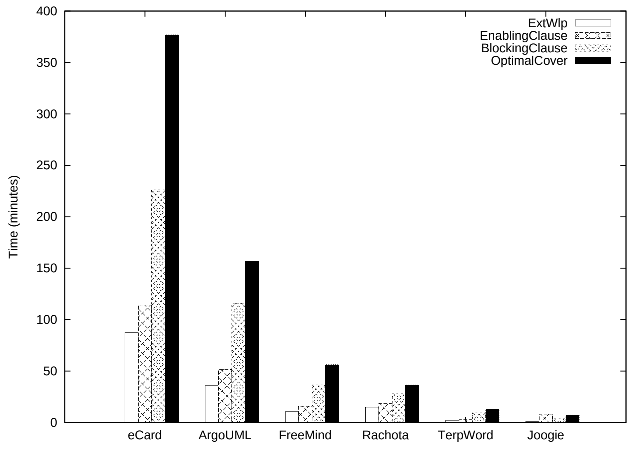


Fig. 4: Performance of the proposed encoding of the weakest liberal precondition compared to other approaches that detect infeasible code.

Results. Figure 4 shows the computation time for each algorithm on our AUTs using the theorem prover Princess. The results show that our proposed encoding together with the algorithm from the previous section yields a significant performance improvement over existing techniques. Furthermore, it shows that applying *EnblClause* only to an effectual subset of block variables results in a relatively small but visible performance improvement.

For all experiments, *EnblClause* applied on an effectual set computed a total $137,997$ queries in 152.33 minutes. *EnblClause* applied on all block variables

Program	ExtWlp		EnablingClause		BlockingClause		OptimalCover	
	Z3	Princess	Z3	Princess	Z3	Princess	Z3	Princess
Open eCard	**62**	**335**	**81**	**437**	247	866	376	1444
ArgoUML	**68**	**215**	**92**	**308**	371	697	382	941
FreeMind	**33**	**112**	**45**	**169**	216	390	244	601
Joogie	**30**	**154**	**49**	**142**	223	482	367	990
Rachota	**273**	**702**	**354**	**875**	773	1306	817	1710
TerpWord	**137**	**370**	**153**	**435**	896	1559	966	2115

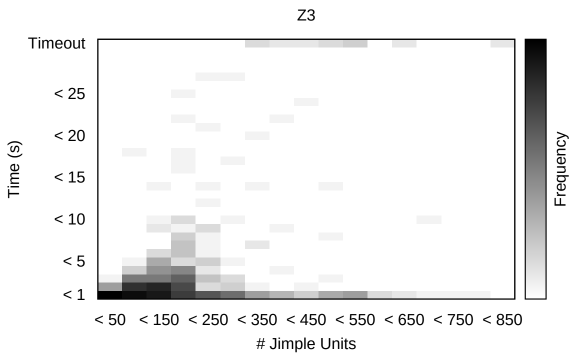
Table 2: Average time (in milliseconds) per method for each program using Z3 and Princess.

used 131,632 queries and 206.20 minutes, with blocking clauses 250,566 queries and 424.11 minutes, and the algorithm from [**?**] 132,976 queries and 646.21 minutes. Here, the time refers to the computation time inside the Princess theorem prover, not counting the overhead in Joogie. To our surprise, applying *EnblClause* only to an effectual set rather than to all block variables results in an increase of theorem prover queries, but reduces the computation time. We assume that, when working on all block variables, it is easier for the theorem prover to cover multiple blocks in one query, but this can create enabling clauses which are very hard to solve.
Table 2 compares the average computation time per AUT for each query for the theorem prover Princess and Z3. We can see that the performance improvements we achieve with the new encoding are visible regardless of the prover.

Figure 5 shows how many procedures of a particular size can be analyzed within a certain time frame using Princess or Z3 with *EnblClause*. Here, a darker color indicates that more procedures, and a lighter color means less. We can see that, for both provers, the majority of analyzed procedures up to 350 Jimple units (which is roughly the number of instructions) can be handled within 5 seconds or less. Z3 always computes an answer for procedures with less than 250 units while princess occasionally timeouts.

In conclusion, our experiments show that the new encoding results in a significant speedup of infeasible code detection compared to existing approaches. Infeasible code detection can be applied to real programs and even with a time limit of a few seconds, a large portion of the procedures in our AUTs can be analyzed.

Threats to Validity. There are several threats to validity to be considered. First, the AUTs are selected more or less randomly and may be biased towards GUI-applications. Different results may occur for other classes of applications, however, due to the size of the applications, we expect this not to happen. Another threat to validity is that we selected stable versions of the source code from the public repositories. Infeasible code detection should target code in production, rather than well tested code. However, this requires a controlled setting which is not available to us at the moment.



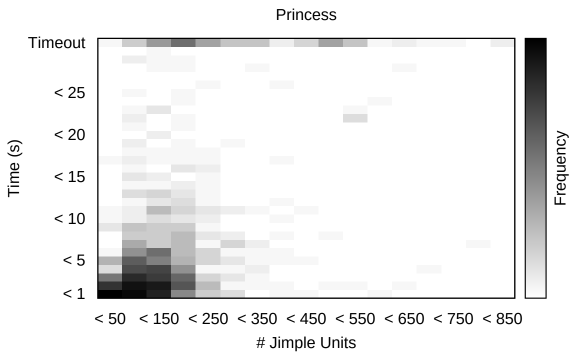


Fig. 5: Detailed comparison of the performance of configuration ExtWlp on all AUTs. The horizontal axis depicts the number Jimple Units (which is roughly LOC), the vertical axis depicts the analysis time of ExtWlp. The color of the individual boxes indicate the number of analyzed procedures.

For implementation reasons, we have to measure the computation time of Princess and Z3 at slightly different positions in the code. That is, the experiments cannot be used to compare the provers with each other. They only show that our approach yields performance improvements in both cases.

Some threat to validity arises from our restriction to Java programs. E.g., we cannot compare approaches that focus on `C` (e.g., [?,?]), which may have a different encoding of programs into logic that affects the performance of our algorithm. However, this is unlikely to happen, as our algorithm only uses logic variables that are handled by the DPLL solver, whereas the memory model usually affects the performance of the theory solver.

7 Conclusion

We have presented a new encoding of weakest-liberal preconditions for infeasible code detection. This encoding allows us to reconstruct a feasible control-flow path (in the abstract program) from a counterexample of an unsatisfiability proof. With the ability to identify feasible control-flow paths, we were able to develop a simple algorithm to detect infeasible code. The algorithm is sound because the absence of feasible executions in the abstract program automatically implies the absence of feasible executions in the original program.

We have shown that our simple algorithm outperforms existing implementations for infeasible code detection. We assume that this is because we put less restriction on the theorem prover when searching for a feasible path, and, that our helper variables are pure logic variables that do not require handling by background theories. We believe that this encoding will allow us to develop specialized background theories for our theorem prover that consider the graph structure of the abstract program to avoid exploring irrelevant or redundant control-flow paths and thus can detect infeasible code even faster.

Our experiments indicate that our tool to detect infeasible code can be applied to real-world programs, and more important, that infeasible code exists even in well tested and stable programs. With that in mind, we are looking into other application domains for infeasible code detection such as compiler optimization, or worst-case execution time analysis.

Being able to obtain feasible executions (and control-flow paths) from the prover could also help to compute under-approximated summaries (e.g., [**?**]) that can be used for bounded, and inter-procedural infeasible code detection.

To summarize, we have presented a new encoding of *wlp*, and an algorithm for infeasible code detection based on this. We were able to show on real-world examples that this encoding detects infeasible code significantly faster than existing approaches and that, based on this encoding, several improvements to infeasible code detection are possible. Finally, we found a lot of infeasible code which we will report to the corresponding developers.

Acknowledgments

Our thanks go Philipp Rümmer for his suggestions on efficient encoding of programs, and the integration with Princess. Further, we would like to thank Jürgen Christ and Jochen Hoenicke for their valuable suggestions on the construction of prover friendly formulas, and their detailed comments on large parts of this paper. This work is in part supported by the grant PEARL (041/2007/A3) and COLAB of the Macao Science and Technology Development Fund.