

Robustness Testing of Intermediate Verifiers

YuTing Chen and Carlo A. Furia

Chalmers University of Technology, Sweden
yutingc@chalmers.se bugcounting.net

Abstract. Program verifiers are not exempt from the bugs that affect nearly every piece of software. In addition, they often exhibit *brittle* behavior: their performance changes considerably with details of how the input program is expressed—details that should be irrelevant, such as the order of independent declarations. Such a lack of robustness frustrates users who have to spend considerable time figuring out a tool’s idiosyncrasies before they can use it effectively.



This paper introduces a technique to detect lack of robustness of program verifiers; the technique is lightweight and fully automated, as it is based on *testing* methods (such as mutation testing and metamorphic testing). The key idea is to generate many simple variants of a program that initially passes verification. All variants are, by construction, equivalent to the original program; thus, any variant that fails verification indicates lack of robustness in the verifier.

We implemented our technique in a tool called *μgie*, which operates on programs written in the popular Boogie language for verification—used as intermediate representation in numerous program verifiers. Experiments targeting 135 Boogie programs indicate that brittle behavior occurs fairly frequently (16 programs) and is not hard to trigger. Based on these results, the paper discusses the main sources of brittle behavior and suggests means of improving robustness.

1 Introduction

Automated program verifiers have become complex pieces of software; inevitably, they contain bugs that make them misbehave in certain conditions. *Verification tools need verification too.*

In order to apply verification techniques to program verifiers, we have to settle on the kind of (correctness) properties to be verified. If we simply want to look for basic *programming errors*—such as memory allocation errors, or parsing failures—the usual verification¹ techniques designed for generic software—from random testing to static analysis—will work as well on program verifiers. Alternatively, we may treat a program verifier as a *translator* that encodes the semantics of a program and specification language into purely logic constraints—which can be fed to a generic theorem prover. In this case, we may pursue a correct-by-construction approach that checks that the translation preserves the intended semantics—as it has been done in few milestone research achievements [20].

There is a third kind of analysis, however, which is peculiar to automated program verifiers that aim at being sound. Such tools input a program complete with specification and other auxiliary annotations, and output either “ SUCCESS” or “ FAILURE”.

¹ In this paper, the term “verification” also designates *validation* techniques such as testing.

Success means that the verifier proved that the input program is correct; but failure may mean that the program is incorrect or, more commonly, that the verifier needs more information to verify the program—such as more detailed annotations. This asymmetry between “verified” and “don’t know” is a form of incompleteness, which is inevitable for sound verifiers that target expressive, undecidable program logics. Indeed, using such tools often requires users to become acquainted with the tools’ idiosyncrasies, developing an intuition for what kind of information, and in what form, is required for verification to succeed. To put it in another way, program verifiers may exhibit *brittle, or unstable, behavior*: tiny changes of the input program that ought to be inconsequential have a major impact on the effectiveness achieved by the program verifier. For instance, Sec. 2 details the example of a small program that passes or fails verification just according to the relative order of two unrelated declarations. Brittle behavior of this kind compromises the usability of verification tools.

In this work, we target this kind of *robustness (stability) analysis* of program verifiers. We call an automated verifier *robust* if its behavior is not significantly affected by small changes in the input that should be immaterial. A verifier that is not robust is *brittle (unstable)*: it depends on idiosyncratic features of the input. Using brittle verifiers can be extremely frustrating: the feedback we get as we try to develop a verified program incrementally is inconsistent, and we end up running in circles—trying to fix nonexistent errors or adding unnecessary annotations. Besides being a novel research direction for the verification of verifiers, identifying brittle behavior has the potential of helping develop more robust tools that are ultimately more usable.

More precisely, we apply *lightweight verification techniques* based on *testing*. Testing is a widely used technique that cannot establish correctness but is quite effective at finding bugs. The goal of our work is to automatically generate tests that reveal brittleness. Using the approach described in detail in Sec. 3, we start from a *seed*: a program that is correct and can be verified by an automated verifier. We *mutate* the seed by applying random sequences of predefined mutation operators. Each mutation operator captures a simple variation of the way a program is written that *does not change its semantics*; for example, it changes the order of independent declarations. Thus, every mutant is a *metamorphic transformation* [4] of the seed—and equivalent to it. If the verifier *fails* to verify a mutant we found a bug that exposes brittle behavior: seed and mutant differ only by small syntactic details that should be immaterial, but such tiny details impact the verifier’s effectiveness in checking a correct program.

While our approach to *robustness testing* is applicable in principle to any automated program verifier, the mutation operators depend to some extent on the semantics of the verifier’s input language, as they have to be semantic preserving. To demonstrate robustness testing in practice, we focus on the Boogie language [17]. Boogie is a so-called *intermediate verification language*, combining an expressive program logic and a simple procedural programming language, which is commonly used as an intermediate layer in many verification tools. Boogie’s popularity² makes our technique (and our implementation) immediately useful to a variety of researchers and practitioners.

As we describe in Sec. 3, we implemented robustness testing for Boogie in a tool called *μgie*. In experiments described in Sec. 4, we ran *μgie* on 135 seed Boogie

² <http://boogie-docs.readthedocs.io/en/latest/#front-ends-that-emit-boogie-ivl>

programs, generating and verifying over 87 000 mutants. The mutants triggered brittle behavior in 16 of the seed programs; large, feature-rich programs turned out to be particularly brittle, to the point where several different mutations were capable of making Boogie misbehave. As we reflect in Sec. 6, our technique for robustness testing can be a useful complement to traditional testing techniques, and it can help buttress the construction of more robust, and thus ultimately more effective and usable, program verifiers.

Tool availability. The tool `μgic`, as well as all the artifacts related to its experimental evaluation, are publicly available [23]. A few additional details about the experiments are available in a longer version of this paper [6].

2 Motivating Example

Let’s see a concrete example of how verifiers can behave brittlely. Fig. 1 shows a simple Boogie program consisting of five declarations, each listed on a separate numbered line.

```

1 function h(int) returns (int);
2 axiom ( $\forall x, y: \text{int} \bullet x > y \implies h(x) > y$ );
3 const a: [int] int;
4 axiom ( $\forall i: \text{int} \bullet 0 \leq i \implies a[i] < a[i + 1]$ );
5 procedure p(i: int) returns (o: int)
   requires  $i \geq 0$ ; ensures  $o > a[i]$ ; { o := h(a[i + 1]); }
```

Fig. 1: A correct Boogie program that exposes the brittleness of verifiers: changing the order of declarations may make the program fail verification.

The program introduces an integer function `h` (ln. 1), whose semantics is partially axiomatized (ln. 2); a constant integer map `a` (ln. 3), whose elements at nonnegative indexes are sorted (ln. 4); and a procedure `p` (ln. 5, spanning two physical lines in the figure)—complete with signature, specification, and implementation—which returns the result of applying `h` to an element of `a`. Never mind about the specific nature of the program; we can see that procedure `p` is correct with respect to its specification: $a[i + 1] > a[i]$ from the axiom about `a` and `p`’s precondition, and thus $h(a[i + 1]) > a[i] = o$ from the axiom about `h`. Indeed, Boogie successfully checks that `p` is correct.

There is nothing special about the order of declarations in Fig. 1—after all, “the order of the declarations in a [Boogie] program is immaterial” [17, Sec. 1]. A different programmer may, for example, put `a`’s declarations before `h`’s. In this case, surprisingly, Boogie fails verification warning the user that `p`’s postcondition may not hold.

A few more experiments show that there’s a fair chance of running into this kind of brittle behavior. Out of the $5! = 120$ possible permutations of the 5 declarations in Fig. 1—each an equivalent version of the program—Boogie verifies exactly half, and fails verification of the other half. We could not find any simple pattern in the order of declarations (such as “line x before line y ”) that predicts whether a permutation corresponds to a program Boogie can verify.

To better understand whether other tools’ SMT encodings may be less brittle than Boogie’s, we used `b2w` [1] to translate all 120 permutations of Fig. 1 to WhyML—the

input language of the Why3 intermediate verifier [9]. Why3 successfully verified all of them—using Z3 as SMT solver, like Boogie does—which suggests that some features of Boogie’s encoding (as opposed to Z3’s capabilities) are responsible for the brittle behavior on the example.

Such kinds of brittleness—a program switching from verified to unverified based on changes that should be inconsequential—can greatly frustrate users, and in particular novices who are learning the ropes and may get stuck looking for an error in a program that is actually correct—and could be proved so if definitions were arranged in a slightly different way. Since brittleness hinders scalability to projects of realistic size, it can also be a significant problem for advanced users; for example, the developers behind the Ironclad Apps [14] and IronFleet [13] projects reported³ that “solvers’ instability was a major issue” in their verification efforts.

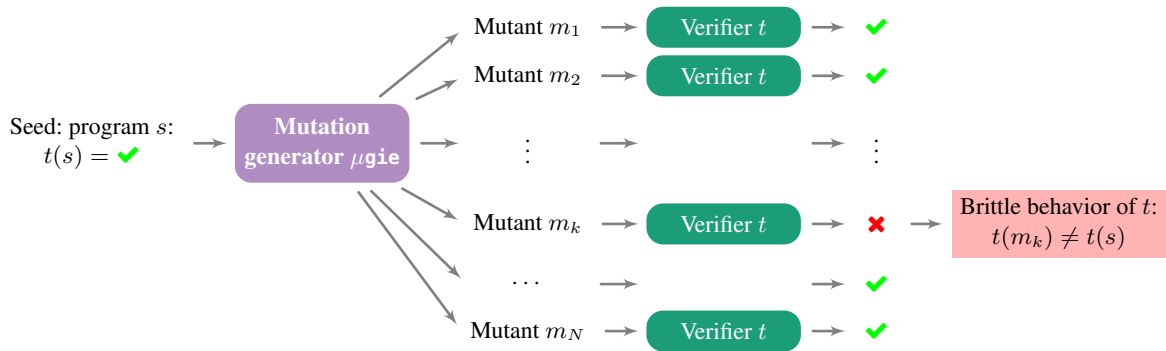


Fig. 2: How robustness testing of Boogie programs works. We start with a correct program s that some Boogie tool t can successfully verify; mutation generator μgie mutates s in several different ways, generating many different *mutants* m_k equivalent to s ; each mutant undergoes verification with tool t ; a mutant m_k that *fails verification* with t exposes *brittle behavior* of t on the two equivalent correct programs $s \equiv m_k$.

3 How Robustness Testing Works

Robustness testing is a technique that “perturbs” a correct and verified program by introducing small changes, and observes whether the changes affect the program’s verifiability. The changes should be inconsequential, because they are designed not to alter the program’s behavior or specification; if they do change the verifier’s outcome, we found lack of robustness. While robustness testing is applicable to any automated program verifier, we focus the presentation on the popular Boogie intermediate verification language. Henceforth, a “program” is a program (complete with specification and other annotations) written in the Boogie language. Fig. 2 illustrates how robustness testing works at a high level; the rest of the section provides details.

³ By an anonymous reviewer of FM 2018.

In general terms, testing requires to build a valid *input*, feed it to the system under test, and compare the system’s output with the *expected* output—given by a testing *oracle*. Testing the behavior of a verifier according to this paradigm brings challenges that go beyond those involved in generating tests for general programs. First, a verifier’s input is a whole *program*, complete with specification and other annotations (such as lemmas and auxiliary functions) for verification. Second, robustness testing aims at exposing subtle inconsistencies in a verifier’s output, and not basic programming errors—such as memory access errors, parsing errors, or input/output errors—that every piece of software might be subject to. Therefore, we need to devise suitable strategies for *input generation* and *oracle generation*.

3.1 Mutation Operators

Input generation. In order to expose brittleness of verifiers, we need to build complex input programs of significant size, complete with rich specifications and all the annotations that are necessary to perform automated verification. While we may use grammar-based generation techniques [28] to automatically build syntactically correct Boogie programs, the generated programs would either have trivial specifications or not be semantically correct—that is, they would not pass verification. Instead, robustness testing starts from a collection of *verified programs*—the *seeds*—and automatically generates simple, semantically equivalent variants of those programs.⁴ This way, we can seed robustness testing with a variety of sophisticated verification benchmarks, and assess robustness on realistic programs of considerable complexity.

Mutation operators. Given a seed s , robustness testing generates many variants $M(s)$ of s by “perturbing” s . Building on the basic concepts and terminology of mutation testing [16],⁵ we call *mutant* each variant m of a seed s obtained by applying a random sequence of *mutation operators*.

A mutation operator captures a simple syntactic transformation of a Boogie program; crucially, mutation operators should *not change a program’s semantics* but only introduce equivalent or redundant information. Under this fundamental condition, every mutant m of a seed s is equivalent to s in the sense that s and m should both pass (or both fail) verification. This is an instance of *metamorphic testing*, where we transform between equivalent inputs so that the seed serves as an oracle to check the expected verifier output on all of the seed’s mutants.

Based on our experience using Boogie and working around its brittle behavior, we designed the mutation operators in Tab. 3, which exercise different language features:

Structural mutation operators change the overall structure of top-level declarations—by changing their relative order (S_1), separating declarations and implementations (S_2), and splitting into multiple files (S_3).

⁴ [6] describes some experiments with seeds that *fail* verification. Unsurprisingly, random mutations are unlikely to turn an unverified program into a verified one—therefore, the main paper focuses on using verified programs as seeds.

⁵ See Sec. 5 for a discussion of how robustness testing differs from traditional mutation testing.

STRUCTURAL	LOCAL	GENERATIVE
S ₁ Swap any two declarations	L ₁ Swap any two local variable declarations	G ₁ Add <code>true</code> as pre-/postcondition, intermediate assertion, or loop invariant clause
S ₂ Split a procedure definition into declaration and implementation	L ₂ Split a declaration of multiple variables into multiple declarations	G ₂ Remove a trigger annotation
S ₃ Move any declaration into a separate file (and call Boogie on both files)	L ₃ Join any two preconditions into a conjunctive one	
	L ₄ Join any two postconditions into a conjunctive one	
	L ₅ Swap any two pre-/postcondition, intermediate assertion, or loop invariant clauses	
	L ₆ Complement an <code>if</code> condition and switch its <code>then</code> and <code>else</code> branches	

Table 3: Mutation operators of Boogie code in categories structural, local, and generative. Operators do not change the semantics of the code they are applied to (except possibly G₂, which is used separately).

Local mutation operators work at the level of procedure bodies—by changing the relative order of or splitting on multiple lines local variable declarations (L₁ and L₂), merging two pre- or postcondition clauses x and y into a conjunctive clause $x \wedge y$ (L₃ and L₄), changing the relative order of assertions of the same program element (L₅), and permuting the `then` and `else` branches of a conditional (L₆).

Generative mutation operators alter redundant information—by adding trivial assertions (G₁), and removing quantifier instantiation suggestions (“triggers” in G₂).

We stress that our mutation operators do not alter the semantics of a Boogie program according to the language’s specification [17]: in Boogie, the order of declarations is immaterial (S₁, L₁, L₂); a procedure’s implementation may be with its declaration or be separate from it (S₂); multiple input files are processed as if they were one (S₃); multiple specification elements are implicitly conjoined, and their relative order does not matter (L₃, L₄, L₅); a conditional’s branches are mutually exclusive (L₆); and `true` assertions are irrelevant since Boogie only checks partial correctness (G₁).

Triggers. G₂ is the only mutation operator that may alter the semantics of a Boogie program in practice: while triggers are suggestions on how to instantiate quantifiers, they are crucial to guide SMT solvers and increase stability in practice [19,5]. Therefore, **we do not consider G₂ semantics-preserving**; our experiments only apply G₂ in a separate experimental run to give an idea of its impact in isolation.

More mutation operators are possible, but the selection in Tab. 3 should strike a good balance between effectiveness in setting off brittle behavior and feasibility of studying the effect of each individual operator in isolation.

3.2 Mutation Generation

Given a seed s , the generation of mutants repeatedly draws random mutation operators and applies them to s , or to a previously generated mutant of s , until the desired number N_M of mutants is reached.

```

input : seed program  $s$ 
input : weight  $w(o)$  for each mutation operator  $o$ 
input : number of mutants  $N_M$ 
output: set of mutants  $M$  of  $s$ 

 $M \leftarrow \{s\}$  // initialize pool of mutants to seed
attempts  $\leftarrow 0$  // number of main loop iterations
while  $|M| < N_M$  do // repeat until  $N_M$  mutants are generated
  if attempts  $>$  MAX_ATTEMPTS then
    | break
  end
   $p \leftarrow$  any program in  $M$ 
   $o \leftarrow$  any mutation operator // draw with probability  $w(o)$ 
   $m \leftarrow o(p)$  // apply mutation operator  $o$  to  $p$ 
   $M \leftarrow M \cup \{m\}$  // add  $m$  to pool  $M$ 
  attempts  $\leftarrow$  attempts + 1
end
return  $M$ 

```

Algorithm 1: Mutant generation algorithm

Alg. 1 shows the algorithm to generate mutants. The algorithm maintains a pool M of mutants, which initially only includes the seed s . Each iteration of the main generation loop proceeds as follows: 1. pick a random program p in the pool M ; 2. select a random mutation operator o ; 3. apply o to p , giving mutant m ; 4. add m to pool M (if it is not already there).

Users can bias the random selection of mutation operators by assigning a weight $w(o)$ to each mutation operator o in Tab. 3: the algorithm draws an operator with probability proportional to its weight, and operators with zero weight are never drawn.

Besides the mutation operator selection, there are two other passages of the algorithm where random selection is involved: a program p is drawn uniformly at random from M ; and applying an operator o selects uniformly at random program locations where o can be applied. For example, if o is S_1 (swap two top-level declarations), applying o to p involves randomly selecting two top level declarations in p to be swapped.

Any mutation operator can generate only finitely many mutants; since the generation is random, it is possible that a newly generated mutant is identical to one that is already in the pool. In practice, this is not a problem as long as the seed s is not too small or the enabled operators too restrictive (for example, S_2 can only generate 2^D mutants, where D is the number of procedure definitions in s). The generation loop has an alternative stopping conditions that gives up after MAX_ATTEMPTS iterations that have failed to generate enough distinct mutants.

Robustness testing. After generating a set $M(s)$ of mutants of a seed s , robustness testing runs the Boogie tool on each mutant in $M(s)$. If Boogie can verify s but fails to verify any mutant $m \in M(s)$, we have found an instance of *brittle behavior*: s and m are equivalent by construction, but the different form in which m is expressed trips up Boogie and makes verification fail on an otherwise correct program.

3.3 Implementation

We implemented robustness testing as a commandline tool `μgie` (pronounced “moo-gie”). `μgie` implements in Haskell the mutation generation Alg. 1, and extends parts of Boogaloo’s front-end [25] for parsing and typechecking Boogie programs.

4 Experimental Evaluation

Robustness testing was initially motivated by our anecdotal experience using intermediate verifiers. To rigorously assess to what extent they are indeed brittle, and whether robustness testing can expose their brittleness, we conducted an experimental evaluation using `μgie`. This section describes design and results of these experiments.

4.1 Experimental Design

A run of `μgie` inputs a *seed* program s and outputs a number of metamorphic mutants of s , which are then verified with some tool t (see Fig. 2).

Seed selection. We prepared a curated collection of seeds by selecting Boogie programs from several different sources, with the goal of having a diverse representation of how Boogie may be used in practice. Each example belongs to one of six groups according to its origin and characteristics; Tab. 4a displays basic statistics about them. Group **A** contains basic Algorithms (search in an array, binary search trees, etc.) implemented directly in Boogie in our previous work [10]; these are relatively simple, but non-trivial, verification benchmarks. Group **T** is a different selection of mainly algorithmic problems (bubble sort, Dutch flag, etc.) included in Boogie’s distribution **Tests**. Group **E** consists of small Examples from our previous work [5] that target the impact of different trigger annotations in Boogie. Group **S** collects large Boogie programs that we generated automatically from fixed, repetitive structures (for example, nested conditionals); in previous work [5] we used these programs to evaluate Scalability. Groups **D** and **P** contain Boogie programs automatically generated by the Dafny [18] and AutoProof [11] verifiers (which use Boogie as intermediate representation). The Dafny and Eiffel programs they translate come from the tools’ galleries of verification benchmarks [8,2]. As we see from the substantial size of the Boogie programs they generate, Dafny and AutoProof introduce a significant overhead as they include axiomatic definitions of heap memory and complex types. In all, we collected 135 seeds of size ranging from just 6 to over 8 500 lines of Boogie code for a total of nearly 260 000 lines of programs and specifications.

Tool selection. In principle, `μgie` can be used to test the robustness of any verifier that can input Boogie programs: besides Boogie, tools such as Boogaloo [25], Symbooglix [21], and blt [5]. However, different tools target different kinds of analyses, and thus typically require different kinds of seeds to be tested properly and meaningfully compared. To our knowledge, no tools other than Boogie itself support the full Boogie language, or are as mature and as effective as Boogie for sound verification (as opposed to other analyses, such as the symbolic execution performed by Boogaloo and

GROUP	# SEEDS	LOC					TOTAL
		MIN	MEDIAN	MEAN	MAX		
A	10	17	34	44	152	439	
D	26	2000	4076	4465	8533	116101	
E	10	13	18	23	49	230	
P	30	986	1665	1911	5737	57330	
S	51	6	126	1047	7286	67006	
T	8	11	41	1662	7378	18283	
<i>all</i>	135	6	642	1718	8533	259389	

(a) Selection of Boogie programs used as seeds: for each GROUP, the number of programs in that group (# SEEDS), and their MINimum, MEDIAN, MEAN, MAXimum, and TOTAL size in non-blank non-comment lines of code. Row *all* summarizes measures over all groups.

TOOL	COMMIT	DATE	Z3
BOOGIE 4.1.1	b2d448	2012-09-18	4.1.1
BOOGIE 4.3.2	97fde1	2015-03-10	4.3.2
BOOGIE 4.4.1	75b5be	2015-11-19	4.4.1
BOOGIE 4.5.0	63b360	2017-07-06	4.5.0

(b) Selection of Boogie versions used in the experiments. For every version of the Boogie TOOL, the corresponding COMMIT hash in Boogie’s Git repository, the DATE of the commit, and the matching Z3 version.

Table 4: Boogie programs (“seeds”) and Boogie tool versions used in the experiments.

Symbooglix) on the kinds of examples we selected. We intend to perform a different evaluation of these tools using *μg*ie in the future, but for consistency and clarity we focus on the Boogie tool in this paper.

In order to understand whether Boogie’s robustness has changed over its development history, our experiments include different versions of Boogie. The Boogie repository is not very consistent in assigning new version numbers, nor does it tag specific commits to this effect. As a proxy for that, we searched through the logs of Boogie’s repository for commit messages that indicate updates to accommodate new features of the Z3 SMT solver—Boogie’s standard and main backend. For each of four major versions of Z3 (4.1.1, 4.3.2, 4.4.1, and 4.5.0), we identified the most recent commit that refers explicitly to that version (see Tab. 4b); for example, commit 63b360 says “Calibrated test output to Z3 version 4.5.0”. Then, we call “Boogie v ” the version of Boogie at the commit mentioning Z3 version v , running Z3 version v as backend.

To better assess whether brittle behavior is attributable to Boogie’s encoding or to Z3’s behavior, we included two other tools in our experiments: CVC4 refers to the SMT solver CVC4 v. 1.5 inputting Boogie’s SMT2 encoding of verification condition (the same input that is normally fed to Z3); Why3 refers to the intermediate verifier Why3 v. 0.86.3 using Z3 4.3.2 as backend, and inputting WhyML translations of Boogie programs automatically generated by b2w [1].

Experimental setup. Each experiment has two phases: first, *generate mutants* for every seed; then, *run Boogie* on the mutants and check which mutants still verify.

For every seed $s \in S$ (where S includes all 135 programs summarized in Tab. 4a), we generate different *batches* $M_O(s)$ of mutants of s by enabling specific mutation operators O in *μg*ie. Precisely, we generate 12 different batches for every seed:

$M_*(s)$ consists of 100 different mutants of s , generated by picking uniformly at random among all mutation operators in Tab. 3 **except** G_2 (that is, each mutation operator gets the same positive weight, and G_2 gets weight zero);

$M_J(s)$, for J one of the 11 operators in Tab. 3, consists of 50 different mutants of s generated by only applying mutation operator J (that is, J gets a positive weight, and all other operators get weight zero).

DEFINITION	DESCRIPTION
S	set of all seeds
$M_O(s)$	set of all mutants of seed s (generated with mutation operators O)
S_t^{\checkmark}	$\{s \in S \mid t(s)\}$
$M_O(s)_t^{\checkmark}$	$\{m \in M_O(s) \mid \neg t(m)\}$
$S_t^{\checkmark \rightsquigarrow \times}$	$\{s \in S_t^{\checkmark} \mid M_O(s)_t^{\checkmark} > 0\}$
$M_O(s)_t^{\infty}$	$\{m \in M_O(s)_t^{\checkmark} \mid t(m) \text{ times out}\}$
$\# \text{ PASS}$	$ S_t^{\checkmark} $
$\# \exists \text{ FAIL}$	$ S_t^{\checkmark \rightsquigarrow \times} $
$\% \exists \text{ FAIL}$	$100 \cdot S_t^{\checkmark \rightsquigarrow \times} / S_t^{\checkmark} $
$\% \text{ FAIL}$	$100 \cdot \text{mean}_{s \in S_t^{\checkmark}} M_O(s)_t^{\checkmark} / M_O(s) $
$\% \text{ TIMEOUT}$	$100 \cdot \text{mean}_{s \in S_t^{\checkmark}} M_O(s)_t^{\infty} / M_O(s) $
$\% \exists \text{ FAIL}$	$100 \cdot \text{mean}_{s \in S_t^{\checkmark \rightsquigarrow \times}} M_O(s)_t^{\checkmark} / M_O(s) $

Table 5: Definitions and descriptions of the experimental measures reported in Tab. 6.

Batch M_* demonstrates the effectiveness of robustness testing with general settings; then, the smaller batches M_J focus on the individual effectiveness of one mutation operator at a time. Operator G_2 is only used in isolation (and not at all in M_*) since it may change the semantics of programs indirectly by guiding quantifier instantiation.

Let t be a tool (a Boogie version in Tab. 4b, or another verifier). For every seed $s \in S$, we run t on s and on all mutants $M_O(s)$ in each batch. For a run of t on program p (seed or mutant), we write $t(p)$ if t verifies p successfully; and $\neg t(p)$ if t fails to verify p (because it times out, or returns with failure). Based on this basic data, we measure robustness by counting the number of verified seeds whose mutants fail verification: see the measures defined in Tab. 5 and the results described in detail in Sec. 4.2.

Running times. The experiments ran on a Ubuntu 16.04 LTS GNU/Linux box with Intel 8-core i7-4790 CPU at 3.6 GHz and 16 GB of RAM. Generating the mutants took about 15 minutes for the batch M_* and 10 minutes for each batch M_J . Each verification run was given a timeout of 20 seconds, after which it was forcefully terminated by the scheduler of GNU parallel [27].

4.2 Experimental Results

Overall results: batch M_* . Our experiments, whose detailed results are in Tab. 6, show that robustness testing is *effective* in exposing *brittle behavior*, which is *recurrent* in Boogie: for 12% of the seeds that pass verification,⁶ there is at least one mutant in batch M_* that fails verification.

Not all seeds are equally prone to brittleness: while on average only 3% of one seed’s mutants fail verification, it is considerably easier to trip up seeds that are *susceptible* to brittle behavior (that is such that at least one mutant fails verification): 27% of mutants per such seeds fail verification.

When the verifier times out on a mutant, it may be because: *i*) the timeout is itself unstable and due to random noise in the runtime environment; *ii*) the mutant takes

⁶ For clarity, we initially focus on Boogie 4.5.0, and later discuss differences with other versions.

GROUP	TOOL	BATCH M_*					# \exists FAIL OF M_J												
		# PASS	# \exists FAIL	% \exists FAIL	% FAIL	% TIMEOUT	% \exists FAIL	S ₁	S ₂	S ₃	L ₁	L ₂	L ₃	L ₄	L ₅	L ₆	G ₁	G ₂	
A	4.1.1	10	2	20%	9%	5%	45%	1	0	1	0	0	0	1	1	0	0	0	
	4.3.2	10	1	10%	4%	0%	42%	0	0	0	0	0	0	1	1	0	0	0	
	4.4.1	10	1	10%	4%	0%	42%	0	0	0	0	0	0	1	1	0	0	0	
	4.5.0	10	1	10%	4%	0%	42%	0	0	0	0	0	0	1	1	0	0	0	
	CVC4	6	0	0%	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	0
	WHY3	7	0	0%	0%	0%	0%	—	—	—	—	—	—	—	—	—	—	—	—
D	4.1.1	0	0	—	—	—	—	0	0	0	0	0	0	0	0	0	0	0	
	4.3.2	24	7	29%	7%	4%	23%	6	0	5	0	0	5	4	4	0	3	17	
	4.4.1	24	7	29%	7%	5%	23%	6	0	5	0	0	5	4	4	0	3	17	
	4.5.0	24	6	25%	8%	6%	33%	5	0	5	0	0	8	4	4	0	1	17	
	CVC4	0	0	—	—	—	—	0	0	0	0	0	0	0	0	0	0	0	
	WHY3	0	0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	
E	4.1.1	5	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	3	
	4.3.2	7	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	3	
	4.4.1	7	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	3	
	4.5.0	7	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	3	
	CVC4	4	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	3	
	WHY3	7	0	0%	0%	0%	—	—	—	—	—	—	—	—	—	—	—	—	
P	4.1.1	0	0	—	—	—	—	0	0	0	0	0	0	0	0	0	0	0	
	4.3.2	14	6	43%	1%	0%	2%	4	0	7	0	0	1	1	0	0	0	10	
	4.4.1	13	5	38%	1%	0%	2%	3	0	6	0	0	0	0	0	0	0	9	
	4.5.0	13	5	38%	1%	0%	2%	4	0	6	0	0	0	0	0	0	0	9	
	CVC4	0	0	—	—	—	—	0	0	0	0	0	0	0	0	0	0	0	
	WHY3	0	0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	
S	4.1.1	51	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	0	
	4.3.2	51	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	0	
	4.4.1	51	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	0	
	4.5.0	51	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	0	
	CVC4	51	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	0	
	WHY3	40	2	5%	1%	1%	25%	—	—	—	—	—	—	—	—	—	—	—	
T	4.1.1	8	1	12%	5%	5%	39%	1	0	1	0	0	1	1	1	0	1	1	
	4.3.2	8	1	12%	8%	8%	62%	1	0	1	0	0	1	1	1	0	1	1	
	4.4.1	8	1	12%	8%	8%	60%	1	0	1	0	0	1	1	1	0	1	1	
	4.5.0	8	1	12%	12%	12%	96%	1	0	1	0	0	1	1	1	0	1	1	
	CVC4	4	0	0%	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	1	
	WHY3	3	0	0%	0%	0%	—	—	—	—	—	—	—	—	—	—	—	—	
all	4.1.1	74	3	4%	2%	1%	43%	2	0	2	0	0	1	2	2	0	1	4	
	4.3.2	114	15	13%	2%	2%	18%	11	0	13	0	0	7	7	6	0	4	31	
	4.4.1	113	14	12%	2%	2%	20%	10	0	12	0	0	6	6	6	0	4	30	
	4.5.0	113	13	12%	3%	2%	27%	10	0	12	0	0	9	6	6	0	2	30	
	CVC4	65	0	0%	0%	0%	—	0	0	0	0	0	0	0	0	0	0	4	
	WHY3	57	2	4%	1%	1%	25%	—	—	—	—	—	—	—	—	—	—	—	

Table 6: Experimental results of robustness testing with μ gie. For each GROUP of seeds, for each TOOL: number of seeds passing verification (# PASS), number and percentage of passing seeds for which at least one mutant fails verification (# \exists FAIL and % \exists FAIL), average percentage of mutants per passing seed that fail verification (% FAIL), average percentage of mutants per passing seed that time out (% TIMEOUT), average percentage of mutants that fail verification per passing seed with at least one failing mutant (% \exists FAIL). The middle section of the table records experiments with batch M_* ; each of the 11 rightmost columns records experiments with batch M_J , for J one of the mutation operators in Tab. 3.

longer to verify than the seed, but may still be verified given longer time; *iii*) verification time diverges.

We ruled out *i*) by repeating experiments 10 times, and reporting a timeout only if all 10 repetitions time out. Thus, we can generally consider the timeouts in Tab. 6 indicative of a genuine degrading of performance in verification—which affected 3% of one seed’s mutants on average.

Boogie versions. There is little difference between Boogie versions, with the exception of Boogie 4.1.1. This older version does not support some language features

used extensively in many larger examples that also tend to be more brittle (groups D and P). As a result, the percentage of verified seeds with mutants that fail verification is spuriously lower (4%) but only because the experiments with Boogie 4.1.1 dodged the harder problems and performed similarly to the other Boogie versions on the simpler ones.

Intermediate verifier vs. backend. Is the brittleness we observed in our experiments imputable to Boogie or really to Z3? To shed light on this question, we tried to verify every seed and mutant using CVC4 instead of Z3 with Boogie’s encoding; and using Why3 on a translation [1] of Boogie’s input. Since the seeds are programs optimized for Boogie verification, CVC4 and Why3 can correctly process only about half of the seeds that Boogie can. This gives us too little evidence to answer the question conclusively: while both CVC4 and Why3 behaved robustly, they could verify none of the brittle seeds (that is, verified seeds with at least one failing mutant), and thus behaved as robustly as Boogie on the programs that both tools can process.⁷ In cases such as the simple example of Sec. 2 (where Why3 was indeed more robust than Boogie), it is really the interplay of Boogie and Z3 that determines brittle behavior. While SMT solvers have their own quirks, Boogie is meant to provide a stable intermediate layer; in all, it seems fair to say that Boogie is at least partly responsible for the brittleness.

Program groups. Robustness varies greatly across groups, according to features and complexity of the seeds that are mutated. Groups D and P are the most brittle: about 1/3 of passing seeds in D, and about 2/5 of passing seeds in P, have at least one mutant that fails verification. Seeds in D and P are large and complex programs generated by Dafny and AutoProof; they include extensive definitions with plenty of generic types, complex axioms, and instantiations. The brittleness of these programs reflects the hardness of verifying strong specifications and feature-rich programming languages: the Boogie encoding must be optimized in every aspect if it has to be automatically verifiable; even a modicum of clutter—introduced by μ gie—may jeopardize successful verification.

By the same token, groups A, E, and T’s programs are more robust because they have a smaller impact surface in terms of features and size. Group S’s programs are uniformly robust because they have simple, repetitive structure and weak specifications despite their significant size; Boogie scales effortlessly on such examples.

Mutation operators and batches M_J . Fig. 7 and the rightmost columns of Tab. 6 explore the relative effectiveness of each mutation operator. S_2 , L_1 , L_2 , and L_6 could not generate any failing mutant—suggesting that Boogie’s encoding of procedure declarations, of local variables, and of conditionals is fairly robust. In contrast, all other operators could generate at least one failing mutant; Fig. 7 indicates that L_3 and S_3 generated failing mutants for respectively 2 seeds and 1 seed that were robust in batch M_* (using all mutation operators with the same frequency)—indicating that mutation operators are complementary to a certain extent in the kind of brittleness they can expose.

Failures. Overall, 13 brittle seeds are revealed by 350 failing mutants in M_* with Boogie 4.5.0. Failures are of three kinds: *a*) timeouts (6 seeds, 252 mutants); *b*) type

⁷ Additionally, Why3 times out on 51 mutants of 2 seeds in group S; this seems to reflect an ineffective translation performed by b2w [1] rather than brittleness of Why3.

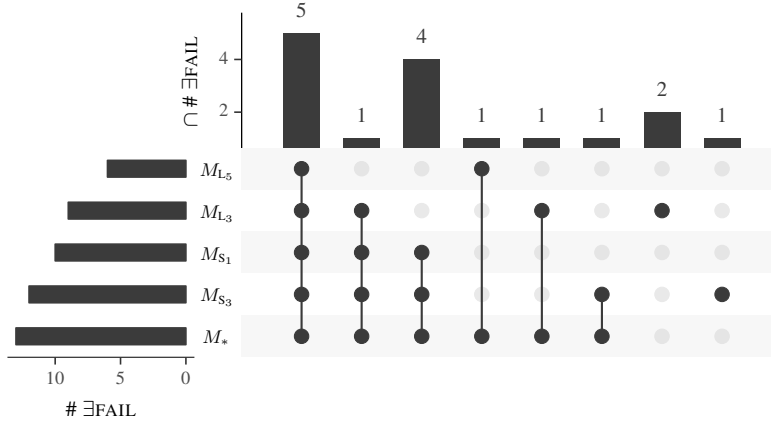


Fig. 7: For each of 16 verified seeds with at least one failing mutant with Boogie 4.5.0, which batches all exclusively include a failing mutant of those seeds. G_2 is excluded and analyzed separately; S_2 , L_1 , L_2 , L_6 could not generate any failing mutant; L_4 generated failing mutants for a strict subset of those in M_* ; G_1 generated failing mutants for a strict subset of those in M_{L_5} .

errors (5 seeds, 10 mutants); *c*) explicit verification failures (2 seeds, 88 mutants). *Timeouts* mainly occur in group D (5 seeds), where size and complexity of the code are such that any mutation that slows down verification may hit the timeout limit; verification of some mutants seems to be non-terminating, whereas others are just slowed down by some tens of seconds. *Type errors* all occur in group P and only when mutation S_3 splits the seed in a way that procedure `update_heap` (part of AutoProof’s heap axiomatization) ends up being declared after its first usage; in this case, Boogie cannot correctly instantiate the procedure’s generic type, which triggers a type error even before Z3 is involved. *Verification failures* occur in seeds of group A and D. In particular, a binary search tree implementation in group A fails verification when the relative order of two postconditions is swapped by L_5 ; while Why3 cannot prove the whole example, it can prove the brittle procedure alone regardless of the postcondition order. In all, it is clear that Boogie’s encoding is quite sensitive to the order of declarations and assertions even when it should not matter.

Triggers. Remember that mutation operator G_2 is the only one that modifies triggers, and was only applied in isolation in a separate set of experiments. As we expected from previous work [19], altering triggers is likely to make verification fail (30 seeds and 276 mutants overall; 20 seeds are only brittle if triggers are modified); most of these failures (26 seeds and 250 mutants) are timeouts, since removing triggers is likely to at least slow down verification—if not make it diverge. Operator G_2 is very effective at exposing brittleness mainly with the complex examples in groups D and P, which include numerous axioms and extensive quantification patterns. Group E’s programs are a bit special because they are brittle—they are designed to be so—but are only affected by mutation operators that remove the trigger annotations on which they strongly depend; in contrast, they are robust against all other mutation operators.

5 Related Work

Robustness. This paper’s robustness testing aims at detecting so-called *butterfly effects* [19]—macroscopic changes in a verifier’s output in response to minor modifications of its input. Program provers often incur volatile behavior because they use automated theorem provers—such as SMT solvers—which in turn rely on heuristics to handle efficiently, in many practical cases, complex proofs in undecidable logics.

Random testing. Our approach uses *testing* to expose brittle behavior of verifiers. By automatically generating test inputs, *random testing* has proved to be extremely effective at detecting subtle errors in programs completely automatically. Random testing can generate instances of complex data types by recursively building them according to their inductive structure—as it has been done for functional [7] and object-oriented [24] programming languages. Random testing has also been successfully applied to security testing—where it is normally called “fuzzing” [12]—as well as to compiler testing [28]—where well-formed programs are randomly generated according to the input language’s grammar.

Mutation testing. This paper’s robustness testing is a form of random testing, in that it applies random mutation operators to transform a program into an equivalent one. The terminology and the idea of applying mutation operators to transform between variants of a program come from *mutation testing* [16]. However, the goals of traditional mutation testing and of this paper’s robustness testing are specular. Mutation testing is normally used to assess the robustness of a test suite—by applying error-inducing mutations to correct programs, and ascertaining whether the tests fail on the mutated programs. In contrast, we use mutation testing to assess the *robustness of a verifier*—by applying semantic-preserving mutations to correct (verified) programs, and ascertaining whether the mutated programs still verify. Therefore, the mutation operators of standard mutation testing introduce bugs in a way that is representative of common programming mistakes; the mutation operators of robustness testing (see Tab. 3) do not alter correctness but merely represent alternative syntax expressing the same behavior in a way that is representative of different styles of programming.

Metamorphic testing. In testing, generating inputs is only half of the work; one also has to compare the system’s output with the *expected output* to determine whether a test is passing or failing. The definition of correct expected output is given by an *oracle* [3]. The more complex the properties we are testing for, the more complex the oracle: a crash oracle (did the program crash?) is sufficient to test for simple errors such as out-of-bound memory access; finding more complex errors requires some form of *specification* [15] of expected behavior.

Even when directly building an oracle is as complex as writing a correct program, there are still indirect ways of extrapolating whether an output is correct. In *differential testing* [22], there are variants of the program under test; under the assumption that not all variants have the same bugs, one can feed the same input to every variant, and stipulate that the output returned by the majority is the expected one—and any outlier is likely buggy. In *metamorphic testing* [26], an input is transformed into an equivalent one according to *metamorphic relations*; equivalent inputs that determine different outputs are indicative of error. Our robustness testing applies mutation operators that

determine identity metamorphic relations between Boogie programs, since they only change syntactic details and not the semantics of programs.

6 Discussion and Future Work

Our experiments with μ gie confirm the intuition—bred by frequently using it in our work—that Boogie is prone to brittle behavior. How can we shield users from this brittle behavior, thus improving the usability of verification technology?

Program verifiers that use Boogie as an intermediate representation achieve this goal to some extent: the researchers who built the verifiers have developed an intuitive understanding of Boogie’s idiosyncrasies, and have encoded this informal knowledge into their tools. End users do not have to worry about Boogie’s brittleness but can count on the tools to provide an encoding of their input programs that has a good chance of being effective.

In contrast, developers of program verifiers still have to know how to interact with Boogie and be aware of its peculiarities.

Robustness testing may play a role not only in exposing brittle behavior—the focus of this paper—but in precisely tracking down the sources of brittleness, thus helping to debug them. To this end, we plan to address *minimization* and *equivalency detection* of mutants in future work. The idea is that the number of failing mutants that we get by running μ gie are not directly effective as debugging aids, because it takes a good deal of manual analysis to pinpoint the precise sources of failure in large programs with several mutations. Instead, we will apply techniques such as delta debugging [29] to reduce the size of a failing mutant as much as possible while still triggering failing behavior in Boogie. Failing mutants of minimal size will be easier to inspect by hand, and thus will point to concrete aspects of the Boogie translation that could be made more robust.

To further investigate to what extent it is Z3 that is brittle, and to what extent it is Boogie’s encoding of verification condition—an aspect only partially addressed by this paper’s experiments—we will apply robustness testing directly to SMT problems, also to understand how Boogie’s encoding can be made more robust.

Robustness testing could become a useful help to developers of program and intermediate verifiers, to help them track down sources of brittleness during development, ultimately making verification technology easier to use and more broadly applicable.

References

1. Ameri, M., Furia, C.A.: Why just Boogie? Translating between intermediate verification languages. In: iFM. Volume 9681 of LNCS., Springer (2016) 1–17
2. AutoProof verified code repository <http://tiny.cc/autoproof-repo>.
3. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering* **41**(5) (2015) 507–525
4. Chen, T.Y., Cheung, S.C., Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology (1998)

5. Chen, Y., Furia, C.A.: Triggerless happy – intermediate verification with a first-order prover. In: *iFM*. Volume 10510 of LNCS., Springer (2017) 295–311
6. Chen, Y., Furia, C.A.: Robustness testing of intermediate verifiers. <http://arxiv.org/abs/1805.03296> (May 2018)
7. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: *ICFP*, ACM (2000) 268–279
8. Dafny examples and tests <https://github.com/Microsoft/dafny/tree/master/Test>.
9. Filliâtre, J., Paskevich, A.: Why3 – where programs meet provers. In: *ESOP*. Volume 7792 of LNCS., Springer (2013) 125–128
10. Furia, C.A., Meyer, B., Velder, S.: Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys* **46**(3) (2014)
11. Furia, C.A., Nordio, M., Polikarpova, N., Tschannen, J.: AutoProof: Auto-active functional verification of object-oriented programs. *STTT* **19**(6) (2016) 697–716
12. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. *Communications of the ACM* **55**(3) (2012) 40–44
13. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: *SOSP*, ACM (2015) 1–17
14. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad Apps: End-to-end security via automated full-system verification. In: *USENIX OSDI*, USENIX Association (2014) 165–181
15. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P.J., Lüttgen, G., Simons, A.J.H., Vilkomir, S.A., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. *ACM Computing Surveys* **41**(2) (2009) 9:1–9:76
16. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* **37**(5) (2011) 649–678
17. Leino, K.R.M.: This is Boogie 2 (2008) <http://goo.gl/QsH6g>.
18. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: *LPAR*. Volume 6355 of LNCS., Springer (2010) 348–370
19. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: *CAV*, Springer (2016) 361–381
20. Leroy, X.: Formal verification of a realistic compiler. *Communications of the ACM* **52**(7) (2009) 107–115
21. Liew, D., Cadar, C., Donaldson, A.F.: Symbooglix: A symbolic execution engine for boogie programs. In: *ICST*, IEEE Computer Society (2016) 45–56
22. McKeeman, W.M.: Differential testing for software. *Digital Technical Journal* **10**(1) (1998) 100–107
23. *μgie* repository <https://emptylambda.github.io/mu-gie/>.
24. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: *ICSE*, IEEE Computer Society (2007) 75–84
25. Polikarpova, N., Furia, C.A., West, S.: To run what no one has run before. In: *RV*. Volume 8174 of LNCS., Springer (2013) 251–268
26. Segura, S., Fraser, G., Sanchez, A.B., Ruiz-Cortés, A.: A survey on metamorphic testing. *IEEE Transactions on Software Engineering* **42**(9) (2016) 805–824
27. Tange, O.: GNU parallel—the command-line power tool. *login: The USENIX Magazine* **36** (2011) 42–47
28. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: *ACM SIGPLAN Notices*. Volume 46., ACM (2011) 283–294
29. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* **28**(2) (2002) 183–200