Symbooglix: A Symbolic Execution Engine for Boogie Programs

Daniel Liew Cristian Cadar Alastair F. Donaldson
Imperial College London
United Kingdom
{daniel.liew, c.cadar, alastair.donaldson}@imperial.ac.uk

***Abstract*—We present the design and implementation of Symbooglix, a symbolic execution engine for the Boogie intermediate verification language. Symbooglix aims to find bugs in Boogie programs efficiently, providing bug-finding capabilities for any program analysis framework that uses Boogie as a target language. We discuss the technical challenges associated with handling Boogie, and describe how we optimised Symbooglix using a small training set of benchmarks. This empirically-driven optimisation approach avoids over-fitting Symbooglix to our benchmarks, enabling a fair comparison with other tools. We present an evaluation across 3749 Boogie programs generated from the SV-COMP suite of C programs using the SMACK front-end, and 579 Boogie programs originating from several OpenCL and CUDA GPU benchmark suites, translated by the GPUVerify front-end. Our results show that Symbooglix significantly outperforms Boogaloo, an existing symbolic execution tool for Boogie, and is competitive with GPUVerify on benchmarks for which GPUVerify is highly optimised. While generally less effective than the Corral and Duality tools on the SV-COMP suite, Symbooglix is complementary to them in terms of bug-finding ability.**

I. Introduction

An intermediate verification language (IVL) simplifies the task of building a program analysis tool, by decoupling the handling of the semantics of a real world programming language from the method that is used to assess the correctness of programs. Much like a compiler, a program analysis tool can have a front-end and a back-end, linked by a common intermediate representation of the input program. The front-end translates the input program from a programming language (e.g. C) into the IVL; the back-end then analyses the IVL program. A single IVL back-end can thus act as an analyser for multiple high-level languages if the front-ends are available. Example IVLs include Boogie [48], WhyML [37] and the IVL proposed by Le et al. [46].

Boogie is a small and simple IVL with a clearly-defined semantics, in contrast to the semantics of real-world programming languages that are usually prone to a degree of ambiguity. A front-end targeting Boogie commits to a specific encoding of the source language, after which program analysis is performed with respect to the precise semantics of Boogie. Resolution of source language ambiguity is controlled explicitly by the front-end, and does not taint the underlying program analysis techniques. Boogie front-ends have been developed for (fragments of) several languages, including C [33], [53], Java [20], C# [2], Dafny [49] and OpenCL/CUDA [27]. Boogie has traditionally been used for program verification, via the Boogie verifier back-end [23], but recently the IVL has also been used for bug-finding (e.g. Boogaloo [52] and Corral [44]). Corral recently replaced the SLAM tool [21] as the engine that powers Microsoft's Static Driver Verifier [43].

Recently there has been growing interest in the use of symbolic execution for testing code written in production languages such as C [28], [39] and Java [19]. In contrast, few attempts to apply symbolic execution in the context of IVLs have been reported [46], [52]. Having a flexible symbolic execution engine for a popular IVL such as Boogie would provide symbolic testing capabilities for any input language for which there is an IVL front-end.

Symbolic execution has already been implemented at the level of intermediate languages—notably KLEE [28], which operates at the level of the LLVM IR [45]. An industry-strength compiler IR such as LLVM's presents advantages and disadvantages for symbolic execution, stemming from the high precision with which language features and the target platform are taken into account (e.g. undefined behaviour, calling conventions, etc.). This precision enables a symbolic execution engine to find low-level bugs, but also adds complexity, in terms of run time overhead, feature addition and code maintenance. In contrast, the syntactic and semantic simplicity of Boogie makes it an ideal platform on which to initially study the combination of symbolic execution with other analyses, and the design of new optimisations for more effective symbolic execution.

We present the design and implementation of Symbooglix (**Sym**bo**l**ic **Boog**ie Ex**e**cutor), an open source [14] symbolic execution engine for Boogie, written in `C#`, that aims to provide:

1) **High coverage** of the Boogie language, supporting all features exercised in practice by existing front-ends;
2) **Faster and more precise** bug-finding capabilities for Boogie programs translated from real-world code, compared with existing IVL back-ends.

We took an *empirically-driven* approach to optimising Symbooglix: we selected a small training set from two large benchmark suites and used only this set to drive optimisation of the tool. This prevented us from over-fitting Symbooglix to our benchmarks, which might unfairly bias comparison with other tools and misrepresent how well Symbooglix would perform on further benchmarks.

We evaluate Symbooglix on two large benchmark suites. The first suite consists of 3749 C programs taken from

the benchmark suite of the International Competition on Software Verification (SV-COMP) [5], translated to Boogie using the SMACK front-end [53]. The translated programs use mathematical integers to represent bitvectors in the original C programs and have a large number of branch points. The second suite consists of 579 GPU kernels written in OpenCL and CUDA, translated to Boogie using the GPUVerify front-end [27]. These programs use bitvector types and exhibit loops with large bounds. We use these suites to conduct a comparison between Symbooglix and five state-of-the-art Boogie analysers: the Boogie verifier [3]; Boogaloo [52], an existing symbolic execution tool; Corral [44] and Duality [50], based on *stratified inlining* and *interpolation*, respectively; and GPUVerify [27], a GPU kernel verifier. Boogaloo does not support bitvectors, so cannot handle the GPU benchmarks, and Symbooglix significantly outperforms Boogaloo on the SV-COMP suite. Our results show that Symbooglix finds more bugs than GPUVerify in the GPU benchmarks, despite GPUVerify being highly optimised for this domain (albeit for verification not bug-finding). On the SV-COMP benchmarks, Symbooglix is generally less effective than Corral and Duality, but is complementary to them in terms of bug-finding ability.

We discuss problems that make an apples-to-apples comparison between Symbooglix and KLEE [28] hard, and present a best-effort comparison of the tools using the SV-COMP benchmarks. Because it is highly tuned towards C, KLEE is more efficient than Symbooglix and finds bugs in more programs, but Symbooglix is able to verify more programs than KLEE can, and finds a significant number of distinct bugs.

In summary, our main contributions are:

1) The design and implementation of Symbooglix, providing symbolic bug-finding capabilities for all high-level languages for which there is a Boogie front-end;
2) An empirically-driven approach to optimising the tool, based on the separation of benchmarks into a small *training* set and a large *evaluation* set, providing an unbiased method for assessing our optimisations;
3) An experimental evaluation on 4328 Boogie programs, assessing the effectiveness of Symbooglix and comparing it with five state-of-the-art Boogie analysers.

II. Background

A. Symbolic execution

Symbolic execution [42] works by enumerating the feasible paths of a program. Program input is treated as *symbolic*, i.e. as a set of unknown values, rather than concrete (i.e. as a set of constants). As the program is executed, constraints are gathered on the input. On encountering a branch, the constraint set and each branch condition are solved to determine which paths leaving the branch are feasible, those paths are then explored.

Symbolic execution can be used to detect bugs, by checking user-supplied assertions, or through implicit safety checks that can be automatically instrumented (e.g. asserting that division by zero is not possible). On reaching an assertion during symbolic execution, the negation of the asserted condition conjoined with the current constraint set is solved. A satisfying assignment to the combined set of constraints yields inputs that will cause the assertion to fail (modulo non-determinism in the application under test). Another use case for symbolic execution is high-coverage test case generation. When exploration of a path terminates (due to program termination or identification of a bug) the gathered constraints can be solved to generate a concrete input that will cause the same path (modulo non-determinism) to be followed.

Interest in symbolic execution has increased dramatically due to the availability of efficient Satisfiability Modulo Theories (SMT) solvers that can handle large sets of constraints [36]. Notable symbolic execution tools include CREST [34], JPF-SE [19], KLEE [28], PEX [54] and SAGE [39].

B. The Boogie IVL

We illustrate some of the core features of Boogie using the simple example of Figure 1. Procedure `checked_div` takes two integers `a` and `b` and returns the result of dividing `a` by `b` unless `b` is zero, in which case `a` is returned and a global `err` flag is set. Procedure `test_div` checks that dividing `a*b` by `a*b` (by calling `checked_div` on line 18) yields the value 1 without setting the error flag, as long as both a and b are non-zero. Instead of employing integer multiplication directly, multiplication is modelled using an *uninterpreted function* $\texttt{MUL} : (\texttt{int} \times \texttt{int}) \rightarrow \texttt{int}$ (line 2). An *axiom* constrains `MUL` to satisfy the *integral domain* property of integer multiplication: if a and b are non-zero then $a \cdot b$ is non-zero (lines 3-4). This abstraction captures exactly the property of multiplication needed to verify this example, avoiding potentially expensive reasoning about non-linear multiplication.

Procedure `checked_div` stores its result in an explicit return variable, `r` (line 5), and the post-condition (`ensures`) and pre-condition (`requires`) for `checked_div` and `test_div` are formalised (lines 6 and 17, respectively). The body of `checked_div` uses a non-deterministic `goto` statement (line 8) and two `assume` statements (lines 10 and 13) to model an `if` statement. Control may transfer non-deterministically to either one of the `l_then` and `l_end` labels targeted by the `goto`. An `assume` e statement blocks further program execution (in a non-erroneous manner) if the guard e does not hold. Thus for any concrete value of `b`, exactly one of the assume statements at lines 10 and 13 will cause execution to block.

The Boogie `int` data type represents the infinite set of mathematical integers. The language also supports fixed-width bitvector types. This allows a front-end for a C-like language to either model machine integers precisely (the GPUVerify front-end does this [27]), or to model machine integers as mathematical integers (the SMACK front-end does this [53]).

We discuss some further Boogie features not illustrated by the above example that we shall refer to later in the paper.

The havoc command accepts a sequence of variable names and sets each variable to a non-deterministic value. This allows abstract modelling of side effects, e.g. reading a value from the network with no knowledge of what that value might be.

```
1  var err: bool;
2  function MUL(int, int): int;
3  axiom (forall a, b: int ::
4    (a != 0 && b != 0 ==> MUL(a, b) != 0));
5  procedure checked_div(a: int, b: int) returns (r:int)
6    ensures (err && b == 0) || (!err && r == a div b); {
7    err := false;
8    goto l_then, l_end;
9  l_then:
10   assume b == 0;
11   err := true; r := a; return;
12 l_end:
13   assume b != 0;
14   r := a div b;
15 }
16 procedure test_div(a: int, b: int) returns (r:int)
17   requires a != 0 && b != 0; {
18   call r := checked_div(MUL(a, b), MUL(a, b));
19   assert r == 1; assert !err;
20 }
```

Fig. 1. An example Boogie program performing checked division.

Maps allow modelling of array and heap data. For types S and T, the type $[S]T$ represents a total map from S to T. If type S has an infinite number of distinct values (e.g. if $S = \mathbb{Z}$) then a map of type $[S]T$ has an infinite number of keys.

Global constants can be declared, and axioms used to restrict their values. The `unique` qualifier specifies that a global constant of type T should have a distinct value from any other `unique`-qualified global constant of type T.

C. Related Boogie tools

In our evaluation (§V) we compare against five existing Boogie analysers which we now briefly survey. All the tools use Z3 [35] for constraint solving.

The Boogie verifier [23] applies weakest precondition methods [24] to transform each procedure in a Boogie program into a verification condition to be checked by an SMT solver. Procedure contracts and loop invariants must be manually provided to enable reasoning about calls and loops; annotations are not inferred automatically. The Boogie verifier should thus not be expected to achieve useful results on the unannotated benchmark suites we consider.

Boogaloo [52] is a symbolic execution tool for Boogie programs that aims to provide a way of debugging failed verification attempts. Boogaloo incorporates two unsound features that may cause bugs to be missed: certain program variables are *concretised* during execution and the domain of maps is *finitised* to eliminate quantified constraints.

Corral [44] is a whole-program analyser built on top of the Boogie verifier. Corral first transforms every loop into a tail-recursive procedure call, then uses Houdini [38] to infer (possibly trivial) pre/post specifications for all procedures. Bug-search from an entry procedure, *main* say, proceeds roughly as follows. Procedures called from *main* are *summarised* by their specifications, so that *main* becomes a loop- and call-free procedure, *main'* say, that soundly over-approximates *main*. If verification condition (VC)-based analysis [24] concludes that *main'* is correct, Corral reports that the program is verified. If analysis yields an error trace through *main'*, Corral checks whether the trace traverses any summarised procedures. If not, the trace exposes a real bug, which Corral reports; otherwise, the summarised procedures are *inlined* into *main'*, and any procedures that they in turn call are summarised. Corral reports *unknown* if a given maximum inlining depth is reached.

Duality [50] employs Craig interpolation to compute inductive invariants at program points, generalising the *Impact* algorithm [51]. Though these invariants are used to prove software correctness, bugs may be identified during invariant search.

GPUVerify [27] attempts to prove that a CUDA or OpenCL kernel is free from certain defects that are specific to GPU programming. A parallel kernel is translated into a sequential Boogie program, instrumented with assertions that check whether it is possible for two distinct threads to race, or diverge on a barrier. The Houdini algorithm [38] is used to infer invariants for over-approximation of loops; this over-approximation may lead to GPUVerify reporting false alarms.

III. Design and Implementation

Symbolic execution of the Boogie IVL. Symbooglix implements in-memory symbolic execution, where explored paths are stored explicitly as *execution states*. That is, for each explored path, Symbooglix keeps track of the execution state (program counter, stack and global variables) so that execution can be resumed at a later stage. As in previous execution systems [28], efficient sharing between states is achieved through a copy-on-write strategy (see *Efficient execution state cloning* in §IV).

The non-deterministic commands `goto` and `havoc` make Boogie a good match for symbolic execution. A `goto` command takes multiple target basic blocks and non-deterministically picks a target to which control transfers. This corresponds to the concept of forking new paths at a branch point in symbolic execution. However, there are two differences in the way forking new paths is implemented. First, in conventional languages, forking usually occurs at `if` statements, so that execution is forked into two paths, following the *then* and *else* sides of the branch. In contrast, Symbooglix implements n-way forking to match the semantics of `goto` in Boogie. Second, branching at a `goto` is *unconditional* in Boogie; Boogie programs can exploit unconditional branching to model program behaviours abstractly. Traditional branching can be simulated via `assume` statements with mutually exclusive conditions (see lines 10 and 13 of Figure 1), and Symbooglix is optimised for this case (see *Goto-assume look-ahead* in §IV).

Recall that `havoc` is used to assign non-deterministic values to a set of variables. In the context of symbolic execution, this involves giving each variable a fresh symbolic value. Two other core Boogie commands that are central to symbolic execution are `assume` and `assert`. When Symbooglix interprets a command of the form `assume` e, it first asks its solver to check whether expression e is satisfiable in the current state. If so, e is conjoined to the current path condition and execution continues. Otherwise, the path is terminated. To interpret an `assert` e command, Symbooglix checks both whether e and $\neg e$ are satisfiable in the current state (note it is possible for both to be satisfiable). If $\neg e$ is satisfiable, Symbooglix records that the assertion can fail and thus that the program under analysis is erroneous. Regardless of this, if e is satisfiable then

execution continues with e conjoined to the path condition, so that analysis continues with respect to inputs that do not cause the assertion to fail.

The `requires` clause on program entry has the purpose of constraining the initial program state, thus it is treated like an `assume`. All other `requires` clauses are treated as assertions on procedure entry. Similarly, `ensures` clauses are treated as assertions on procedure exit. A specification-only procedure (which has no body) is executed by asserting its `requires` clause, havocking its `modifies` set (specifying which global variables might be updated), and assuming its `ensures` clause.

Path exploration. An important aspect of symbolic execution is the order in which feasible paths are explored. This is typically controlled by search heuristics. In Symbooglix we use a variant of the depth-first search (DFS) strategy that aims to prevent the search getting stuck in loops by always preferring to follow the path leaving a loop if it is feasible. This variant behaves like a normal depth first search in all other aspects.

Constraint solving. Symbooglix's symbolic expressions are constructed using Boogie's expression building API. To support our work, we have contributed several related changes to the upstream Boogie project (including many bug fixes). The most important change was the ability to make expressions immutable. This allows safe sharing of expressions across execution states, and enables efficient structural equality testing between expressions, which is used by many optimisations.

To answer satisfiability queries during exploration and obtain concrete solutions to the collected constraint sets, constraints are printed in the standard SMT-LIBv2 format [25], and then passed to an SMT solver. Our current implementation uses the Z3 constraint solver [35], due to its support for all the different features required by Symbooglix, such as integers, bitvectors, quantifiers, maps and uninterpreted functions.

Inconsistent assumptions. Boogie differs from conventional languages in that the entire program execution is subject to a set of initial constraints, specified via axioms, the `unique` qualifier on global constants, and `requires` clauses associated with the procedure from which execution commences (see §II). If the initial constraints are *inconsistent* (i.e. they are equivalent to `false`), the program is *vacuously* correct. In our experience, inconsistency of initial constraints is often unintentional, and indicative of a problem with the Boogie program under consideration. To guard against this, Symbooglix supports an optional mode that checks the consistency of initial constraints before execution starts. If this mode is disabled then Symbooglix requires that the assumptions are consistent, otherwise its behaviour is undefined.

IV. Optimisations

Our initial design of Symbooglix included only a few basic optimisations which we implemented without benchmarking the tool. We then optimised Symbooglix in an empirically-driven manner, guided by performance on a set of benchmarks. Because we wanted to compare Symbooglix with other Boogie analysis tools and wished to understand the extent to which Symbooglix's optimisations would be generally applicable, we were cautious not to overfit Symbooglix's optimisations to the benchmarks used in our experimental evaluation. To avoid this, we randomly selected a *training set* consisting of 10% of our benchmarks, and benchmarked Symbooglix exclusively with respect to this training set during optimisation (see §V-D). Below, we summarise the main optimisations we implemented.

Unique global constants constraint representation. This optimisation was motivated by performance problems we observed when running on the SV-COMP portion of the training set, which declares global variables with the `unique` qualifier (see §II-B). Our initial approach to handling `unique`, by emitting a quadratic number of constraints to assert pairwise disjointness, did not scale well. To improve performance, we took advantage of the SMT-LIBv2 `distinct` function, which returns true iff all its arguments are pairwise distinct and is efficiently handled by Z3.

Global dead declaration elimination. We observed that benchmarks in the SV-COMP training set often declare many global variables, functions and axioms that are not used by the program. These *dead declarations* are emitted by SMACK for every Boogie program it generates, e.g. for SMACK's general-purpose floating point representation and memory model. We implemented an analysis that initially marks a global declaration as *necessary* if it is used syntactically by a procedure in the program, and then iteratively marks further declarations as *necessary* if they are referred to by a declaration already marked as *necessary*. Once a fixed point is reached, all declarations not marked as *necessary* are removed.

Goto-assume look-ahead. We developed this optimisation based on intuition related to symbolic execution of Boogie programs. Recall from the example of §II-B that conditional control flow is realised in a Boogie program through a combination of `goto` and `assume` commands. The initial implementation of Symbooglix would always fork at a `goto` command. However, the current path constraints often mean that one of the `assume` statements targeted by the `goto` will have a failing guard. (This is a known issue in symbolic execution: prior work has reported that often fewer than 20% of symbolic branches encountered during execution have both sides feasible [30].) As a result, Symbooglix would often spawn a new execution state, only to kill it at the next `assume` instruction. The *goto-assume look ahead* optimisation changes how the `goto` command is handled by looking ahead at the next instruction of each target basic block. If the next instruction is an `assume`, we check whether it is satisfiable, and if not, we do not fork a new state for that path.

Expression simplification. These optimisations simplify expressions as they are constructed, by folding constants (e.g., `5+4 = 9`) and rewriting certain expression patterns (e.g., `1+x+2 = 3+x`). The patterns we simplify are based on potential simplifications we observed when running Symbooglix on the training set, and on some of the patterns used by KLEE. To help ensure preservation of the exact semantics of Boogie's operators during simplification, we used the Z3 SMT solver to verify correctness of many of the rewriting patterns. One example of subtle semantics is that the `div` and `mod` operators

use euclidean division. Because our implementation language, C#, uses truncated division we had to implement euclidean division in terms of truncated division (following [47]) to provide constant folding for `div` and `mod`.

Constraint independence. This constraint solving optimisation, inspired by EXE [29], eliminates those constraints which are not necessary to determine the satisfiability of a given query. The optimisation transitively computes dependent constraints by considering the set of used variables and uninterpreted functions until a fixed point is reached.

Map updates at concrete indices. This optimisation was inspired by the way in which KLEE handles array accesses at concrete indices. In Symbooglix the value of a map variable was originally represented as an expression tree. Initially the expression is just the symbolic variable representing the map (e.g., `m`). As the map is populated, map updates are added to this expression. For example writing 5 followed by 15 to index 0 would yield `m[0 := 5][0 := 15]`. The expression tree can thus become large after many updates to the map, which can impede performance due to large expressions being passed to the constraint solver. To address this, we try to avoid increasing the size of the expression tree representing a map when possible. We optimised for the case where only concrete indices are used to index a map. In this case, rather than updating the expression tree representing the map with new writes, we store a set of pairs mapping writes at concrete indices to their corresponding value. When reading from a location at a concrete index that was previously written, the corresponding value is returned directly. In the case of a read from a concrete index not contained in the set, a map selection expression that reads from the current version of the expression tree (without the current set of concrete writes) at that concrete index is returned. In the case of a read/write at a symbolic index, Symbooglix switches to using an expression tree to represent the map by flushing all the stored writes to the expression tree.

Map updates at symbolic non-aliasing indices. This optimisation was motivated by several benchmarks where symbolic indices were used to index maps, but such that the associated indices could not alias. In the examples we investigated, the indices were always of the form `C + s`, with `C` a constant integer, distinct for each index, and `s` a symbolic integer variable, common among the indices. Clearly for constants $C_0 \neq C_1$, we are guaranteed to have $(C_0 + s) \neq (C_1 + s)$. This lead us to generalise the *map updates at concrete indices* optimisation to internally store a mapping of non-aliasing (concrete or symbolic) indices to expressions (rather than constant indices to expressions). Expression aliasing is determined by simple syntactic patterns; we currently recognise the case of distinct constant literals (capturing the initial optimisation), and the set of expressions matching the pattern `C + s`, where C is distinct in each expression and where the types of `C` and `s` are integers. This optimisation is currently implemented only for integers, and not bitvectors.

Efficient execution state cloning. This optimisation was inspired by how KLEE handles cloning of execution states, and was motivated by high memory usage in Symbooglix on the training set. The number of execution states can grow quickly during symbolic execution, so efficient state cloning, both in terms of memory used and time taken, is important. Symbooglix originally cloned states in a simple, non-optimal manner. Because expressions are immutable in Symbooglix they never need to be cloned, but the data structures that a state uses to refer to them do. The initial implementation simply made new copies of these data structures. However, profiling several memory-intensive training set benchmarks revealed that a lot of memory was being used by these data structures. To overcome this, we implemented more efficient execution state cloning using C# immutable data structures. For our internal representation of maps, which are not immutable, we added a simple copy-on-write mechanism similar to that which KLEE uses to represent memory objects.

V. Evaluation

We now present in detail our method for evaluating Symbooglix and a selection of other Boogie-based tools. For reproducibility, the tools and all non-commercially sensitive benchmark programs are made available online at http://symbooglix.github.io.

A. Benchmark suites

We consider two benchmark suites containing Boogie programs from two distinct problem domains, and originating from two different languages.

The *SV-COMP benchmark suite* (abbreviated to SV-COMP) consists of programs generated from the C benchmarks used in the 2015 "International Competition on Software Verification" (SV-COMP 2015), translated into Boogie using SMACK [53]. We used the SMACK-translated programs made available online by the SMACK authors [13]. The repository contains 3760 benchmarks, of which we use 3749: four benchmarks exhibit inconsistent assumptions (see §III) and seven are empty [12].

The *GPU benchmark suite* (abbreviated to GPU) consists of Boogie programs generated from a set of 579 GPU kernels written in OpenCL and CUDA, which have been collected to evaluate the GPUVerify tool [22], [27]. The original kernels are drawn from a number of open source GPU benchmark suites and one commercial suite. Among these kernels, at least 32 exhibit data race errors (which manifest as failing assertions in the Boogie programs generated by GPUVerify): 5 are genuine bugs previously found by GPUVerify, and 27 are artificial bugs injected in a previous evaluation of GPUVerify [27].

The SV-COMP and GPU suites provide a comprehensive and challenging set of evaluation benchmarks, covering correct and buggy examples. The SV-COMP suite utilises mathematical integers while the GPU suite utilises bitvector operations.

B. Benchmark preparation

Checking for inconsistent assumptions. As discussed in §III, Symbooglix expects the initial set of assumptions of a Boogie program to be consistent if its optional checking mode is disabled. In our evaluation we decided not use the checking mode because no other tool performs this check and thus

Symbooglix would be unfairly penalised in terms of tool run time. We removed or fixed all benchmarks exhibiting inconsistent assumptions (4 benchmarks in each suite [11]).

Labelling benchmarks. To compare Symbooglix with competing tools in terms of bug-finding ability and capability to perform exhaustive verification, we tagged each benchmark with one of the following labels:

1) *Correct:* the benchmark is free from bugs;
2) *Incorrect:* the benchmark contains at least one bug;
3) *Unknown:* the benchmark may or may not contain bugs.

To infer as many *correct* and *incorrect* labels as possible, we devised the following experiment. For each benchmark, we ran each compatible Boogie tool introduced in §II (in multiple configurations, as detailed in §V-C), with a timeout of 900 seconds and a memory limit of 10 GiB. We did not run Symbooglix at this point, because initially we wanted to use the labels to select a training set for Symbooglix and evaluate its progress over time (see §V-D). If one tool classified a program as *correct* and another tool classified the program as *incorrect*, we investigated the reasons for this, knowing that one tool must be wrong in its analysis. Otherwise, if at least one tool classified a program as *correct* we labelled the program *correct*, while if at least one tool classified a program as *incorrect* we labelled the program *incorrect*. Because Boogie and GPUVerify can produce false positive bug reports, we ignored cases where these tools classified a program as *incorrect* during the labelling process. We labelled a benchmark *unknown* if no tool could reliably classify the program as *correct* or *incorrect*, except in cases where an existing label indicating the program's status was provided by the benchmark suite.

During the labelling process, we checked for generic failures and crashes in the tools. This revealed one bug shared by Boogie and Corral [4], four bugs in Corral (including a case where Corral would report a false positive), and a crash bug in Duality [6]. We reported these bugs and in some cases provided our own fixes. The Corral bugs were promptly fixed.

The SV-COMP suite already provides labels for its constituent benchmarks. However, we found 76 discrepancies between these existing labels and the ones we inferred. Because the labels are for the original C programs in SV-COMP, the mismatch could be caused by an incorrect translation from C to Boogie by SMACK, or it could be a genuine mislabelling of the original benchmark. We did not examine all 76 cases, but for the ones we did examine, the discrepancy was caused by the translation process. For example, two of the benchmarks check whether a self-equality comparison on an arbitrary floating point variable can fail. The assertion in the original C program could fail because `NaN == NaN` is false (NaN stands for "Not a Number". Floating-point comparisons always return false if one of the operands is NaN) however the corresponding assertion in the Boogie program could not fail because `NaN` was not represented in SMACK's model of floating point numbers. We reported this issue to the SMACK developers [8].

For the GPU suite, we labelled the benchmarks that contained deliberately injected bugs as *incorrect*. We surprisingly found mismatches between two kernels that were supposed to have injected bugs and the results reported by the tools applied to those kernels. In one of the kernels it turned out that the injected code did not actually induce a bug so this kernel was removed from our benchmark suite. The other kernel had a mistake (introduced by an error in the injection process) causing the injected bug to be unreachable. We fixed this kernel in our benchmark suite so that the injected bug was reachable.

TABLE I
INITIAL AND FINAL BENCHMARK LABELLINGS.

	SV-COMP			GPU		
	Initial	**Final**	**Training**	**Initial**	**Final**	**Training**
Correct	2705	2704	270	479	491	45
Incorrect	1044	1045	104	37	38	3
Unknown	0	0	0	63	50	9
Overall	3749	3749	374	579	579	57

The *Initial* columns in Table I summarise our labelling without Symbooglix. At the end of our study, having optimised Symbooglix, we were able to re-label the benchmarks based on additional accuracy from Symbooglix's results. The *Final* columns in Table I reflect this labelling. Note that the number of *Correct* labels decreases in the *Final* labelling because the labels provided with some SV-COMP programs were incorrect with respect to the Boogie programs (as discussed earlier).

C. Tools evaluated

We compare Symbooglix with all actively maintained open-source tools that analyse Boogie programs: the Boogie verifier, Boogaloo, Corral, Duality, and GPUVerify (see §II for background on these tools). We did not run Boogaloo and Duality on the GPU suite because they do not support the bitvector types and operations generated by the front end of GPUVerify. We could not apply GPUVerify to SV-COMP, because GPUVerify only supports analysis of Boogie programs generated by its own front-end. The configuration used for each tool was as follows:

Boogaloo was run twice on SV-COMP. Once with a loop bound of 8 and once without a bound. Concretisation was disabled to avoid false negatives and counter-example minimisation was disabled to increase performance.

Boogie was run with the `-errorLimit:1` option, so that at most one error is generated.

Corral was run twice on each suite: once with a recursion bound of 8 and once with a very large bound ($\sim 2^{30}$) on SV-COMP; and once with a bound of 64 and once with a very large bound ($\sim 2^{30}$) on GPU. We picked a bound of 8 for SV-COMP because this was used by the authors of SMACK for their SMACK+Corral SV-COMP 2015 submission. The larger bound of 64 for GPU was chosen because our experience with these benchmarks indicated that loops with large iteration counts are common (a consequence of the throughput-oriented nature of GPU applications). The very large bound ($\sim 2^{30}$) is approximately half the largest integer that Corral supports for specifying the bound; the Corral authors advised against using the largest integer due to potential overflow bugs in Corral.

Duality was run using the same large bound as used for Corral ($\sim 2^{30}$); we used the same bound since Duality is built on top of Corral. We did not consider a smaller bound because the

interpolation-based analysis used by Duality depends upon the ability to unwind a program to a significant depth.

GPUVerify was run on the GPU suite with automatic invariant inference enabled. We disabled extra invariants with which the benchmarks had been manually annotated, so that GPUVerify ran in an unassisted manner.

Symbooglix was run using its default settings, except that the checking of inconsistent assumptions was disabled (see §V-B) and the timeout per solver query was set to 30 seconds. Having a solver timeout prevents Symbooglix getting stuck checking the feasibility of a particular path but may prevent full exploration of the benchmark.

Each tool was allowed a maximum execution time of 900 seconds per benchmark (which is the time used at the last edition of SV-COMP, except that we use wall clock time instead of CPU time). A run of a tool on a single benchmark consists of two pieces of information, the result type and the execution time. The former is the answer the tool gives—*bug found*, *verified* (i.e. no bug found, no tool crash, and no bound, memory limit, solver or global timeout reached), or *unknown* (i.e. no bug found and not *verified*). In the case of Symbooglix and Boogaloo *verified* is equivalent to exploring all feasible paths and finding no bugs.

Each tool was executed three times on the same benchmark, and these runs were combined using the following approach. For results types, if at least one run reports *verified* or *bug found*, we take that result (we initially observed conflicting result types due to tool bugs, but these disappeared once the bugs were fixed). Otherwise, if all runs result in *unknown*, the overall result is *unknown*. To combine the execution times, we treat any of the three results that were of type *unknown* as having taken the maximum allowable time (i.e. 900 seconds). We then compute the arithmetic mean of these times. The rationale here is to penalise tools that terminate abnormally (e.g. crash) after a short amount of time.

All tools except Boogaloo are written in C#. To run them on our Linux machine, we used Mono `3.12.1`, with a minor patch [16] to fix crashes we were experiencing.

D. Evaluation of Empirically-Driven Optimisations

As discussed in §IV, we took an empirically-driven approach to optimising Symbooglix, incrementally optimising the tool guided by a *training* set. The training set was obtained by taking the prepared benchmarks (see §V-B) and randomly selecting 10% of each label for both benchmark suites. The number of benchmarks used for our training set broken down by label can be seen in the *Training* columns of Table I, totalling 374 benchmarks from the SV-COMP suite and 57 benchmarks from the GPU suite. The size of the GPU training set is not exactly 10% of the the initial benchmark labelling because the training set was selected based on results from an early run of the tools in which the tools were not run optimally. This led to fewer benchmarks being labelled *Correct* and more benchmarks being labelled *Unknown*.

At various intervals during Symbooglix's optimisation we stopped development and ran that version of Symbooglix on the training set. We refer to these versions of Symbooglix as *snapshots*. We monitored our progress by comparing the performance of the latest snapshot to previous snapshots. The following table details the eleven snapshots, giving each a short name and indicating the order in which the optimisations of §IV were added. Due to the nature of our development, the optimisations are applied cumulatively.

1) **Baseline:** the starting point for our optimisation work; incorporates *unique global constants constraint representation*.
2) **GlobalDDE:** adds *global dead declaration elimination*.
3) **GotoAssumeLA:** adds *goto-assume look-ahead*.
4) **ExprSimpl:** adds *expression simplification*.
5) **ConstrIndep:** adds *constraint independence*.
6) **RemSomeRecur:** improves an algorithm that searches expressions for symbolic variables and uninterpreted functions by making it iterative and caching results; adds further expression simplification rules; adapts stack size to avoid overflow errors.
7) **RemSomeDbg:** removes a data structure used for debugging that was accidentally left behind. We discovered this after profiling the memory usage of Symbooglix.
8) **MapConstIdx:** adds *map updates at concrete indices*.
9) **MapSymIdx:** adds *map updates at symbolic non-aliasing indices*.
10) **EffcntClone:** adds *Efficient execution state cloning*.
11) **SmplSolv:** optimises the solver interface to assess whether the expression to be checked for satisfiability is constant or already in the constraint set.

Experimental setup. To assess the progress of our optimisation effort, we ran each snapshot on the training set on a single machine with an eight core Intel Xeon CPU (3.3GHz) with 48GiB of RAM running Linux. We used the process described in §V-C to run Symbooglix, enforcing a 5GiB memory limit per benchmark.

To visualise the progress of Symbooglix over time we use *quantile function* plots as used in SV-COMP [5]. In Figure 2 the top and bottom plots show results for the eleven snapshots for the SV-COMP and GPU training sets, respectively. Each curve represents a run of Symbooglix on a particular snapshot. We compute a score for each snapshot by adding one point for each benchmark that the snapshot accurately classifies as *correct* or *incorrect* and subtracting one point for inaccurate classifications. For classification we used the *initial* labelling of §V-B, but updated this labelling as Symbooglix managed to classify additional benchmarks. Each point denotes a benchmark that was correctly classified as *correct* or *incorrect*. The y-coordinate is the time taken to analyse the benchmark, and for each curve, benchmarks are sorted based on time. The x coordinate represents the accumulated score for the snapshot. Thus a point (x, y) shows that analysis takes y seconds or fewer for the previous x benchmarks plotted. The y-axis uses a linear scale between 0 and 1, and a log scale thereafter. This prevents times close to 0 from making the range of the y-axis excessively large. The ordering of the rightmost data point on the axis ranks the snapshots in terms of classification ability, the width of the curve along the x-axis is the number of correct classifications, and the area under a plot represents the total execution time for the correctly-classified benchmarks in that snapshot.

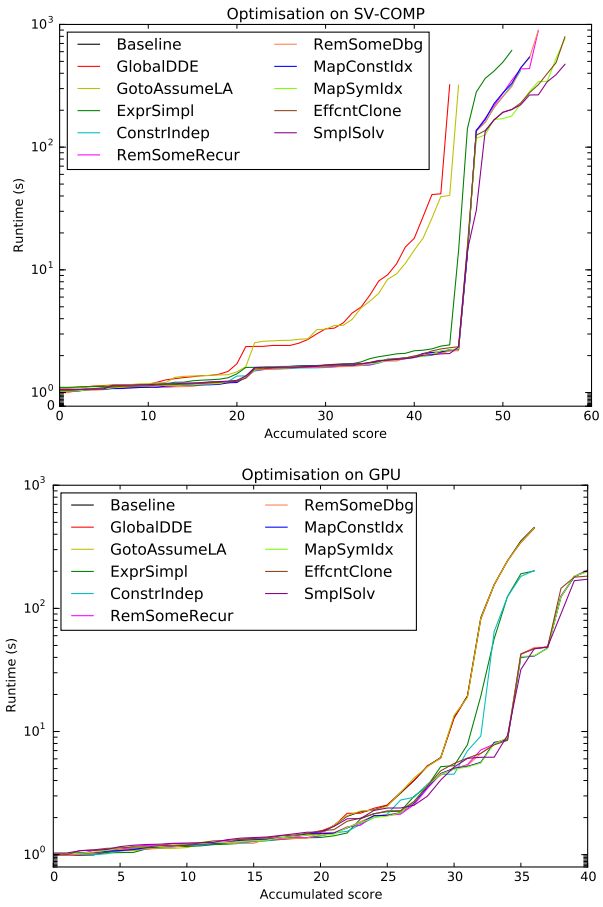

Fig. 2. Quantile function plot for Symbooglix running on the training sets for SV-COMP (top) and GPU (bottom) at different snapshots. Due to the large number of snapshots, the plot is designed to be viewed in colour. The maximum possible accumulative scores are 374 (SV-COMP) and 57 (GPU).

Snapshot results for the SV-COMP training set. The top plot in Figure 2 shows that quantile plot for SV-COMP. Most snapshots bring improvements to either the number of correctly classified benchmarks or the overall run time (or both). The most substantial impact is made by the *GlobalDDE* and *ExprSimpl* optimisations. Note that the *Baseline* is not visible on the plot because the SV-COMP benchmarks contain many unused universally-quantified axioms over uninterpreted functions. In *Baseline* these are all passed to the constraint solver, causing all benchmarks to time out. The *GlobalDDE* snapshot eliminates these unused axioms, allowing Symbooglix to make progress. The other high-impact optimisation is *ExprSimpl*, which allows six additional benchmarks to be correctly classified, and also brings a significant improvement in running time. Finally the *MapSymIdx* optimisation allows four additional benchmarks to be classified.

Snapshot results for the GPU training set. The bottom plot in Figure 2 shows the quantile plot for the GPU suite. *Baseline* is present in this plot because the GPU suite does not use quantified axioms, allowing Symbooglix to make progress from the beginning. As in SV-COMP, *ExprSimpl* improves performance (though here does not classify additional benchmarks). *RemSomeRecur* leads to a significant performance gain and four additional correctly-classified benchmarks. Several optimisations do not make any difference on the GPU suite: *GotoAssumeLA* (due to the very limited amount of forking that occurs in the GPU benchmarks), *MapConstIdx* (GPUVerify's front-end employs a symbolic representation of thread ids, meaning that maps are rarely indexed concretely), and *MapSymIdx* (because the optimisation does not currently support bitvectors).

Snapshot results for the entire SV-COMP and GPU suites are included on the project website [15].

E. Comparison of Symbooglix with Other Boogie Analysers

Experimental setup. We ran the comparison of the tools on a large general purpose computing cluster [7] with 20-core Intel Ivybridge CPU nodes, each with 128 GiB RAM running Linux. We used the approach discussed in §V-C to run each tool, and enforced a memory limit of 10 GiB per benchmark. Our timing results are prone to fluctuations due to hardware differences between nodes; we in part account for this by reporting averages over three independent runs.

Results table. Table II shows the extent to which the tool configurations we compare were able to verify or find bugs in the SV-COMP and GPU suites. For Boogaloo and Corral, the 8, 64 and NB suffixes indicate whether the tools were invoked with a bound of 8, 64, or with no bound (for Corral, NB actually means the huge bound of $\sim 2^{30}$). The *Verified* and *Bug found* columns indicate, for each tool, the number of benchmarks labelled *correct* and *incorrect*, respectively, that the tool could accurately classify as such. *False alarms* identifies cases where a tool reports a *correct* benchmark as *incorrect*. *Unknown* captures timeouts, memory exhaustion and crashes. As expected, only the Boogie verifier and GPUVerify report false alarms, and no tool reported a false negative (classifying an *incorrect* benchmark as *correct*).

Comparison with Boogaloo. Table II shows that for the SV-COMP benchmarks, Boogaloo-NB is more effective than Boogaloo-8. Symbooglix verifies more benchmarks than Boogaloo-NB: 236 vs. 64. The tools verify 58 common benchmarks, with Symbooglix verifying 178 benchmarks for which Boogaloo-NB reports *unknown*, and Boogaloo-NB verifying 6 benchmarks for which Symbooglix reports *unknown*. Symbooglix was also able to find more bugs than Boogaloo-NB: 395 vs. 122. The tools find bugs in 107 common benchmarks, with Symbooglix finding bugs in 288 benchmarks for which Boogaloo-NB reported *unknown*, and Boogaloo-NB finding bugs in 15 benchmarks for which Symbooglix reports *unknown*. Recall that Boogaloo cannot be applied to the GPU suite because it does not support bitvectors.

Comparison over the SV-COMP suite. Comparing all the tools applied to the SV-COMP suite, Table II shows that Corral and Duality are the clear winners, with Corral-NB performing best in terms of bug-finding ability, and Duality proving most capable at verifying benchmarks. It is not surprising that Symbooglix is less effective at verification than these tools, since symbolic execution is primarily geared towards finding bugs, and suffers from path explosion on bug-free programs. In

TABLE II
RESULTS FOR BOOGIE ANALYSIS TOOLS APPLIED TO THE SV-COMP AND GPU SUITES, USING FINAL CLASSIFICATION LABELS.

SV-COMP suite				
Tool	**Verified**	**Bug found**	**False alarm**	**Unknown**
Boogie	0	1021	2668	60
Boogaloo-8	43	122	0	3597
Boogaloo-NB	64	122	0	3563
Corral-8	1348	541	0	1860
Corral-NB	1365	553	0	1831
Duality	1856	426	0	1467
Symbooglix	236	395	0	3118
GPU suite				
Tool	**Verified**	**Bug found**	**False alarm**	**Unknown**
Boogie	260	35	165	119
Corral-64	298	28	0	253
Corral-NB	297	28	0	254
GPUVerify	403	34	76	66
Symbooglix	303	35	0	241

terms of bug-finding ability, Symbooglix is some way behind Corral-NB, finding bugs in 395 vs. 553 benchmarks.

To assess whether Symbooglix and Corral-NB have *complementary* bug-finding capabilities, we compared times taken for these tools to find bugs for all SV-COMP benchmarks labelled *incorrect*. The comparison is visualised by the scatter plot of Figure 3. A point at (x, y) indicates that for a given *incorrect* benchmark, Corral-NB and Symbooglix took x and y seconds, respectively, to find the bug. Cases where the tools reported *unknown* are treated as reaching the 900s timeout limit. Points above the diagonal indicate that Corral-NB outperformed Symbooglix (295 cases), points below the diagonal indicate that Symbooglix outperformed Corral (309 cases). Both tools reported *unknown* for 441 benchmarks (these points lie at the top right corner of the plot). The shape of the plot clearly shows that the tools have complementary abilities when it comes to bug-finding: the tools find bugs in 344 common benchmarks, but in 51 cases Symbooglix finds a bug where Corral-NB does not (the points lying on the far right vertical) and in 209 cases Corral-NB finds a bug where Symbooglix does not (the points lying on the top horizontal). The large number of points lying close to the x-axis indicate cases where Symbooglix finds a bug within a matter of seconds, but where the time taken by Corral varies dramatically. For 70 benchmarks where Corral-NB takes more than 100s to find a bug, Symbooglix finds a bug within 10s, and there are no benchmarks for which the reverse is true. An analogous plot comparing Symbooglix with Duality, presented on our project website [15], shows a very similar picture.

Comparison over the GPU suite. Table II shows that Symbooglix finds the most bugs in GPU among the tools that do not report false alarms: 35 bugs compared with 28 bugs found by both Corral-NB and Corral-64 (the same bugs are identified by each Corral configuration). Furthermore, Symbooglix finds a superset of the 28 bugs found by Corral. The Boogie verifier also find 35 bugs, but with a high false alarm rate (165 alarms); GPUVerify finds 34 bugs, with a lower false alarm rate (76 alarms). Comparing GPUVerify and Symbooglix further, both tools find bugs in 31 GPU benchmarks, with Symbooglix finding 4 bugs not found by GPUVerify, and GPUVerify finding 3 not found by Symbooglix.

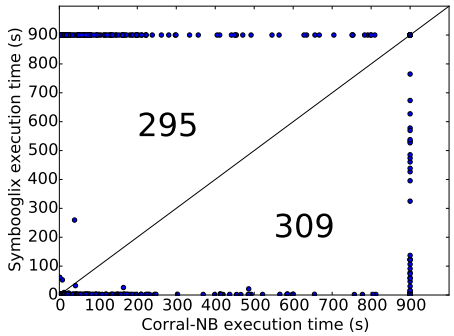

Fig. 3. Comparison of bug-finding times on the SV-COMP suite for Symbooglix and Corral-NB.

Symbooglix is able to verify slightly more benchmarks than Corral-64: 303 vs. 298, and the tools are highly complementary at verification: each tool managed to verify 225 common benchmarks, with Symbooglix verifying 78 benchmarks where Corral-64 reports *unknown*, and Corral-64 verifying 73 benchmarks where Symbooglix reports *unknown*. As expected, since it was designed for this purpose, GPUVerify is able to verify the largest number of GPU benchmarks: 403. Symbooglix and GPUVerify can verify 258 common benchmarks, with Symbooglix able to verify 45 benchmarks where GPUVerify reports *unknown*, and GPUVerify able to verify 145 benchmarks where Symbooglix reports *unknown*.

Our results show a stronger performance from Symbooglix on the GPU suite compared with the SV-COMP suite. We attribute this to the fact that the benchmark suite makes no use of quantifiers (in translating OpenCL and CUDA kernels to Boogie, the GPUVerify front end uses domain-specific strategies for avoiding quantifiers [22]), and to a process of *predication* applied by the GPUVerify front-end, whereby conditional code is largely flattened, reducing the number of paths in the resulting Boogie program [27].

F. Comparison with KLEE

Because we also apply Symbooglix to benchmarks that arise from C programs, it seems natural to compare the tool with KLEE, a state-of-the-art symbolic execution tool targeted towards C code. Various issues make this comparison less straightforward than it might appear: due to various engineering issues, KLEE cannot be applied out-of-the-box to the SV-COMP examples, and an apples-to-apples comparison is not possible because the Boogie benchmarks that Symbooglix analyses have already been translated by the SMACK front-end, which may have changed the semantics (and shape) of the benchmarks. However, we believe that a brief comparison is still useful in highlighting some differences between the tools.

We took the 374 SV-COMP benchmarks used during Symbooglix's training phase and removed 7 floating point benchmarks which KLEE cannot handle (Symbooglix can handle them because SMACK provides an abstraction for floating point operations). We then removed the benchmarks where the original SV-COMP labels (i.e. those for the C programs) did not match the labels inferred for the corresponding

TABLE III
RESULTS FOR KLEE AND SYMBOOGLIX ON THE REDUCED TRAINING SET.

Tool	Verified	Bug found	False alarm	Unknown
Symbooglix	17	31	0	313
KLEE	10	54	1 (see text)	296

Boogie programs. After this filtering, we were left with 361 benchmarks [10], which we call the *reduced training set*.

We modified [9] KLEE to support the built-in verifier functions used by the SV-COMP benchmarks and ran it on the reduced training set. This revealed several engineering issues. The SV-COMP benchmarks are a mix of 32-bit and 64-bit C benchmarks, and KLEE only works correctly when it is compiled for a target that matches the compilation target for the benchmarks. This required us to build a 32-bit and 64-bit build of KLEE to run on the 32-bit and 64-bit benchmarks respectively. We also found that KLEE cannot run the majority of the 64-bit benchmarks, which are based on code from the Linux kernel and use `extern` globals that are not initialized. Symbooglix does not have these issue because the Boogie IVL is architecture independent, and SMACK's translation does handle `extern` globals. These issues illustrate a trade-off between the levels at which the two tools operate: KLEE runs LLVM bitcode, which precisely models system implementation details, while Symbooglix runs Boogie programs, where such details are left abstract. The former approach is better at finding subtle implementation-level bugs, but is more time-consuming to apply (as illustrated by the issues above). While the latter can miss such bugs, avoiding precise system implementation details can simplify looking for bugs that are independent from these details.

With these issues in mind: Table III shows how Symbooglix and KLEE compare in terms of benchmark classification. Symbooglix was able to verify more benchmarks than KLEE. The tools verified 9 common benchmarks (KLEE was faster 2/3 of the time), with Symbooglix verifying 8 benchmarks that KLEE did not, and KLEE verifying 1 benchmark that Symbooglix did not. KLEE was able to find more bugs than Symbooglix. The tools found bugs in 18 common benchmarks (in all cases KLEE found the bug faster), with Symbooglix finding bugs in 13 benchmarks that KLEE did not, and KLEE finding bugs in 37 benchmarks that Symbooglix did not. The single false alarm reported by KLEE is, in fact, not really a false alarm: the associated benchmark is labelled as correct, but KLEE reports an out-of-bounds memory access. The benchmark is from an SV-COMP category in which memory-safety checking is not required. SMACK omits array bounds checks when translating benchmarks in this category to Boogie, but KLEE always checks array bounds and thus raises this (genuine) error. If similar issues apply in the application of KLEE to other SV-COMP benchmarks, the number of bugs found by KLEE in Table III might be higher than it would be if we could disable KLEE's automatic checks when appropriate; however, KLEE does not support disabling of these checks.

Finally, note that a useful feature of KLEE is that, on detecting a bug, KLEE can generate a concrete input to trigger it. With engineering effort, we could extend Symbooglix to query the SMT solver in order to generate conditions that would cause a buggy Boogie program to fail. If the Boogie program was generated by a front-end (e.g. SMACK or GPUVerify), extra effort, tailored to the nature of the translation into Boogie, would be required to map the failure conditions for the Boogie program to a bug-triggering input in the original program.

VI. RELATED WORK

Symbolic execution is widely established as an effective method for finding bugs in software and generating high-coverage test suites, is at the heart of several practical tools [19], [28], [34], [39], [54] and has found application in numerous domains (see [31], [32] for a discussion of its recent impact).

To our knowledge, the only other existing symbolic execution tool for Boogie is Boogaloo [52], which supports a smaller subset of the Boogie language than Symbooglix. Boogaloo was principally designed to help understand errors reported during functional verification attempts, thus the tool incorporates heuristics to limit the extent to which quantifiers (prevalent in functional verification) appear in solver queries. Symbooglix, in contrast, provides only basic quantifier handling, passing quantified expressions directly to the Z3 SMT solver.

In §II-C we provided an overview of other Boogie analysis tools: Corral [44], Duality [50], GPUVerify [27] and the Boogie verifier [23]. The combination of symbolic execution with over-approximating abstractions is a potentially rich avenue for further investigation [26], [40], [41].

VII. CONCLUSION

We have presented Symbooglix, a new symbolic execution engine for the Boogie intermediate verification language, and described an empirically-driven approach to optimising the tool. Through a large experimental evaluation on two diverse benchmark suites, we find that Symbooglix significantly outperforms Boogaloo, an existing symbolic execution tool, in terms of applicability and analysis coverage. Symbooglix is competitive with GPUVerify and out-performs other state-of-the-art Boogie analysers on a suite of benchmarks for which GPUVerify is highly optimised. On a suite of Boogie programs derived from the SV-COMP 2015 benchmark suite, the overall analysis capabilities of Symbooglix are lower than those of the Corral and Duality tools, but Symbooglix is highly complementary to these tools in terms of bug-finding ability.

ACKNOWLEDGEMENTS

This work was supported by an EPSRC Industrial CASE Studentship, sponsored by ARM Ltd, and an EPSRC Early-Career Fellowship. We thank N. Polikarpova, A. Lal, P. Hosek, S. Burbidge, M. Harvey, Z. Rakamaric, S. Qadeer and K. McMillan for assistance during Symbooglix's development and evaluation. We also thank J. Ketema, L. Pina, A. Mattavelli, A. Lascu, P. Deligiannis, and the ICST reviewers for their feedback.

References

[1] *Communications of the Association for Computing Machinery (CACM)*.

[2] "BCT: Byte Code Translator for .NET code to Boogie," https://github.com/boogie-org/bytecodetranslator.

[3] "Boogie from Microsoft Research," https://github.com/boogie-org/boogie.

[4] "Boogie needs to sanitize variable names used in SMTLIB output issue," http://boogie.codeplex.com/workitem/10246.

[5] "Competition on Software Verification (SV-COMP)," http://sv-comp.sosy-lab.org/.

[6] "[Duality] "Prover error: unknown predicate from prover: q@fp" issue," http://corral.codeplex.com/workitem/1.

[7] "Imperial College London HPC service," http://www.imperial.ac.uk/admin-services/ict/self-service/research-support/hpc/hpc-service-support/service/.

[8] "Include NaN in floating point modelling issue," https://github.com/smackers/smack/issues/66.

[9] "Modified KLEE with SV-COMP support," https://github.com/symbooglix/klee/tree/svcomp.

[10] "Modified SV-COMP 2015 benchmarks," https://github.com/symbooglix/sv-benchmarks/tree/klee_svcomp15_smack.

[11] "SMACK can generate inconsistent axioms on variables using the "unique" keyword issue," https://github.com/smackers/smack/issues/71.

[12] "SMACK generated Boogie programs – Empty bpl files issue," https://github.com/smackers/sbb/issues/1.

[13] "SV-COMP 2015 benchmarks converted to Boogie by SMACK," https://github.com/smackers/sbb.

[14] "Symbooglix GitHub repository," https://github.com/symbooglix/symbooglix.

[15] "Symbooglix Project website," https://symbooglix.github.io.

[16] "Try to fix an assertion failure triggered by calls to System.Environment issue," https://github.com/mono/mono/pull/1649.

[17] Jul. 2012.

[18] Jul. 2014.

[19] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF-SE: A Symbolic Execution Extension to Java PathFinder," in *Proc. of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, Mar.-Apr. 2007.

[20] S. Arlt and M. Schäf, "Joogie: Infeasible Code Detection for Java," in *Proc. of the 24th International Conference on Computer-Aided Verification (CAV'12)*, Jul. 2012, pp. 767–773.

[21] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Communications of the Association for Computing Machinery (CACM)*, vol. 54, no. 7, pp. 68–76, 2011. [Online]. Available: http://doi.acm.org/10.1145/1965724.1965743

[22] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, and S. Qadeer, "Engineering a Static Verification Tool for GPU Kernels," in *Proc. of the 26th International Conference on Computer-Aided Verification (CAV'14)*, Jul. 2014, pp. 226–242.

[23] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs," in *Proc. of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, Nov. 2005, pp. 364–387.

[24] M. Barnett and K. R. M. Leino, "Weakest-precondition of Unstructured Programs," in *Proc. of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, Sep. 2005, pp. 82–87.

[25] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," Department of Computer Science, The University of Iowa, Tech. Rep., 2010, available at `www.SMT-LIB.org`.

[26] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons, "Proofs from tests," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'08)*, Jul. 2008.

[27] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: A Verifier for GPU Kernels," in *Proc. of the 27th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'12)*, Oct. 2012.

[28] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[29] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically Generating Inputs of Death," in *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, Oct.-Nov. 2006.

[30] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008, a shorter version of this article appeared in CCS'06.

[31] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic Execution for Software Testing in Practice—Preliminary Assessment," in *Proc. of the 33rd International Conference on Software Engineering, Impact Track (ICSE Impact'11)*, May 2011.

[32] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Communications of the Association for Computing Machinery (CACM)*, vol. 56, no. 2, pp. 82–90, 2013.

[33] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A Practical System for Verifying Concurrent C," in *Proc. of the 22nd Theorem Proving in Higher Order Logics (THPHOLs'09)*, Aug. 2009. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=117859

[34] "CREST: Automatic Test Generation Tool for C," http://code.google.com/p/crest/.

[35] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, Mar.-Apr. 2008.

[36] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the Association for Computing Machinery (CACM)*, vol. 54, no. 9, pp. 69–77, Sep. 2011.

[37] J.-C. Filliâtre and A. Paskevich, "Why3 – Where Programs Meet Provers," in *Proc. of the 22nd European Symposium on Programming (ESOP'13)*, Mar. 2013. [Online]. Available: https://hal.inria.fr/hal-00789533

[38] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *Symposium of Formal Methods Europe*, Mar. 2001.

[39] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[40] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional may-must program analysis: Unleashing the power of alternation," in *Proc. of the 37th ACM Symposium on the Principles of Programming Languages (POPL'10)*, Jan. 2010.

[41] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "Synergy: A new algorithm for property checking," in *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'06)*, Mar. 2006.

[42] J. C. King, "Symbolic execution and program testing," *Communications of the Association for Computing Machinery (CACM)*, vol. 19, no. 7, pp. 385–394, 1976.

[43] A. Lal and S. Qadeer, "Powering the Static Driver Verifier Using Corral," in *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'14)*, Nov. 2014, pp. 202–212.

[44] A. Lal, S. Qadeer, and S. Lahiri, "Corral: A solver for reachability modulo theories," in *Proc. of the 24th International Conference on Computer-Aided Verification (CAV'12)*, Jul. 2012. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=161927

[45] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*, Mar. 2004.

[46] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC Using an Intermediate Verification Language and Symbolic Simulation," in *Proceedings of the 50th Annual Design Automation Conference*, Jun. 2013, pp. 1–6.

[47] D. Leijen, "Division and Modulus for Computer Scientists," http://research.microsoft.com/pubs/151917/divmodnote.pdf, 2001.

[48] K. R. M. Leino, "This is Boogie 2," Tech. Rep., June 2008, http://research.microsoft.com/apps/pubs/default.aspx?id=147643.

[49] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, ser. LPAR'10, Apr. 2009, pp. 348–370.

[50] K. McMillan and A. Rybalchenko, "Computing relational fixed points using interpolation," Tech. Rep. MSR-TR-2013-6, January 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=180055

[51] K. L. McMillan, "Lazy Abstraction with Interpolants," in *Proc. of the 18th International Conference on Computer-Aided Verification (CAV'06)*, Aug. 2006, pp. 123–136.

[52] N. Polikarpova, C. A. Furia, and S. West, "To Run What No One Has Run Before," in *Proc. of the 2013 Runtime Verification (RV'13)*, Sep. 2013.

[53] Z. Rakamarić and M. Emmi, "SMACK: Decoupling Source Language Details from Verifier Implementations," in *Proc. of the 26th International Conference on Computer-Aided Verification (CAV'14)*, Jul. 2014, pp. 106–113.

[54] N. Tillmann and J. De Halleux, "Pex: white box test generation for .NET," in *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, Apr. 2008.