On Symbolic Execution of Decompiled Programs

Lukáš Korenčik
Faculty of Informatics, Masaryk University
Brno, Czech Republic
Email: xkorenc1@fi.muni.cz

Petr Ročkai
Faculty of Informatics, Masaryk University
Brno, Czech Republic
Email: xrockai@fi.muni.cz

Henrich Lauko
Faculty of Informatics, Masaryk University
Brno, Czech Republic
Email: xlauko1@fi.muni.cz

Jiří Barnat
Faculty of Informatics, Masaryk University
Brno, Czech Republic
Email: barnat@fi.muni.cz

***Abstract*—In this paper, we present a combination of existing and new tools that together make it possible to apply formal verification methods to programs in the form of x86_64 machine code. Our approach first uses a decompilation tool (remill) to extract low-level intermediate representation (LLVM) from the machine code. This step consists of instruction translation (i.e. recovery of operation semantics), control flow extraction and address identification.**

The main contribution of this paper is the second step, which builds on data flow analysis and refinement of indirect (i.e. data-dependent) control flow. This step makes the processed bitcode much more amenable to formal analysis.

To demonstrate the viability of our approach, we have compiled a set of benchmark programs into native executables and analysed them using two LLVM-based tools: DIVINE, a software model checker and KLEE, a symbolic execution engine. We have compared the outcomes to direct analysis of the same programs.

I. Introduction

Formal verification of software has made substantial progress towards real-world applicability. Nonetheless, formal methods still occupy a very small niche within mainstream engineering processes. While scalability and accessibility to the wider engineering public continue to be a major concern, growing adoption uncovers new challenges and opportunities. One of the areas that often comes up as a friction point is integration with existing, often testing-heavy development workflows.

Widely deployed development tools roughly fall into two broad categories. The first of those is based on static analysis and gives quick and rough assessment of program behaviour. Those tools are usually integrated into IDEs or into the compilation process to give early feedback to the developer. The other category is often focused on post-mortem analysis and contains tools that work with compiled programs: interactive debuggers like `gdb`, dynamic instrumentation like `valgrind` or performance analysis tools like `perf`.

The latter are usually deployed in two types of situations: either during development when diagnosing problems that were uncovered during testing, where the user is armed with full source code and often with a particular failing testcase. On the other hand, some of the same tools are also deployed when analysing crashes that happened in production or when analysing the behaviour (or misbehaviour) of third-party binaries. In those cases, availability of source code, reproducibility of the bad behaviour and even the ability to safely execute the code in question are far from guaranteed.

A. Motivation

Formal methods offer powerful tools for gaining insight into program behaviour. In particular, symbolic execution is a lightweight method that can quickly search through the state space of a program to uncover bugs or other anomalies. This ability is extremely valuable when dealing with either unknown programs or with unknown circumstances under which a known program misbehaves. Clearly, it is desirable to apply those methods (including, but not limited to traditional symbolic execution) to programs which are not readily available in source form, or where the source code is not easily processed by existing analysis tools.

One possible route to symbolic execution of machine code is through *program decompilation*: an approach where the machine code is translated back into a higher-level description (often an intermediate representation with more structure than is exposed in raw machine code). Since an array of analysis tools based on the LLVM [11] intermediate representation already exists, a suitable decompiler could make it possible to use those tools in situations where source code is not available. This paper presents our attempt at extending LLVM-based decompilation tools to provide enough structure to make symbolic execution feasible, and to adapt LLVM-based analysis tools to work with the output of such an augmented decompiler.

B. Goals

The main goal of our effort was to build a complete tool chain which takes a compiled program (i.e. machine code) as its input and performs symbolic search of its state space. The tool should be able to search for various events in the program,

This work has been partially supported by the Czech Science Foundation grant No. 18-02177S.

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00044

e.g. call to a particular function, and for memory errors (out-of-bounds memory access, double free, use-after-free and similar) and other common safety problems.

C. Contribution

There are three main areas in which we believe this paper contributes to the state of the art:

1. it explores the problem space of applying formal techniques to compiled programs in a modular, compositional manner,
2. describes specific techniques for recovery of additional program structure in the decompiled intermediate representation, and finally
3. provides a prototype tool chain[1] which demonstrates the viability and modularity of the approach, and which provides a choice of two dissimilar symbolic execution backends (KLEE and DIVINE).

II. Related Work

Since we are, in general terms, interested in the analysis of the behaviour of compiled programs, the most immediately useful tools are the standard debugging aids, which all work with executables. Out of these, the most relevant (and also the most powerful) are non-interactive runtime analyzers, such as the tool for memory error detection memcheck from the valgrind suite [15], which uses dynamic code translation and instrumentation to achieve its goals.

A. Disassembly and Decompilation

A typical examples of such a tool is IDA Pro [8], which is perhaps the most widely used binary analysis toolset in the reverse-engineering community. However, a number of alternatives to IDA exist, for instance radare2 [1]. While the tools in this category cannot themselves be used for automated program verification, they often appear as the first stage in multi-tool analysis and verification pipelines.

The tool chain presented in this paper builds on three disassembly and decompilation components. The principal tool, which drives the entire disassembly process all the way to LLVM IR generation is mcsema [17]. To parse the executable, disassemble the machine code and identify subroutines, mcsema uses the disassembler from the dyninst [14] tool set (though it can be configured to also use other tools, e.g. IDA Pro, to perform those tasks). Additionally, the conversion of individual instructions into LLVM is performed by remill [18] – this process is described in more detail in Section III-C.

B. Decompilation-based Verifiers

The design space of verification of binaries through the use of decompilation technology (as opposed to their direct analysis) has been explored in the literature. The most common approaches were based around custom intermediate representations and tailor-made disassembly or decompilation tools to go with them. Understandably, those frameworks are not mutually compatible, and analysis backends built on one cannot be, in general, easily re-targeted to a different platform.

The Binary Analysis Platform project, described in [4], is a versatile framework for analysing machine code, with focus on security analysis, program verification and reverse engineering.

Besides more traditional verification tools that build on a standard ahead-of-time decompiler, an approach based on static analysis and abstract interpretation has been proposed in [9]. In this approach, the analysis runs in lockstep with disassembly and feeds obtained information back into the disassembler to improve its precision.

In [20], the authors focus on adversarial analysis of executables, and combine a number of techniques to recover structure (control flow graphs), apply static analyses to the program (e.g. dependency analysis or program slicing) and dynamic analysis to discover vulnerabilities or other behaviours of interest, mainly using symbolic and concolic execution.

An earlier attempt to use remill with KLEE, was presented in [21] under the name KLEE-NATIVE, but the project is unfortunately no longer maintained and the provided source code fails to build with the current upstream version of remill. Unlike in our present approach, KLEE-NATIVE did not decompile the program ahead of time: instead, KLEE was modified to directly work with executables, using remill to dynamically (just-in-time) translate each instruction into a sequence of LLVM operations, which were then interpreted using the standard symbolic engine inherited from KLEE.

C. Symbolic Execution

Program verification techniques based on symbolic execution [10], symbolic program code analysis [16] and on symbolic approaches to model checking [13] have been the subject of extensive research.

A common compromise between standard testing and symbolic execution employs both representations at once, a practice known as *concolic testing* [19, 7]. The concrete value is used to guide the selection of the control flow on branches without the need to consult an SMT solver, which saves computation time and also immediately provides concrete test inputs to reproduce uncovered bugs.

A different take on symbolic execution, called *veritesting*, has been described in [2]. In this paper, the authors propose, essentially, a combination of bounded model checking (under the name static symbolic execution) and traditional (dynamic) symbolic execution.

The DIVINE model checker [3], which we use as one of the backends in the present paper, makes use of program transformation to implement symbolic execution, described in more detail in [12].

Of course, an important tool in the ecosystem is KLEE, a symbolic execution engine [5] that performs symbolic execution on top of LLVM IR [11]. Besides standalone usage as a symbolic executor, KLEE has become also a back-end tool for other types of analyses and for verification. In our present work, we use KLEE as one of the two evaluation backends.

[1] Source code available at https://divine.fi.muni.cz/2020/decompile/.

III. Decompilation

In this section, we will first describe existing decompilation strategies which target the LLVM IR and their deficiencies in the context of program verification in general and symbolic execution in particular. We will go on to discuss possible improvements and remedies to the highlighted problems, and the particular solutions that we have designed, implemented and evaluated.

A. Preliminaries

The goal of a decompiler is to recover a machine-independent description of the program, preferably in a form that can be further processed by automated tools. This process involves a number of steps and sub-components:

1. extraction of the program from an executable file, into a representation that the remaining steps can work with: we consider this process to be of marginal interest and will not further discuss it in the present paper,
2. identification of instruction boundaries (relevant for variable-length, polymorphic encodings, including `x86_64`): this step has been well studied and does not directly interact with verification efforts,
3. instruction decoding and the recovery of instruction semantics: this is the first step that we will consider in a bit more detail, in Section III-C,
4. address recognition: in machine code, numeric data and memory addresses are represented and processed uniformly; however, for further analysis, it is crucial that the constant data (appearing either as instruction operands or in the data section of the program) that represents memory addresses is tagged as such,
5. control flow extraction: the control flow in a machine program is, like other aspects of the program, entirely unstructured and is simply described by jump instructions with a numeric argument (which represents the address of the next instruction to execute); this step reconstructs a *control flow graph* which consists of *basic blocks* (nodes) and jump instructions (edges),
6. subroutine recovery, which identifies *call sites*, *call targets*, function arguments, and return values,
7. synthesis of higher-level program description, which ties the information from all the preceding steps together and emits the entire program in a suitable format.

The result of step 7 is the roughest admissible decompilation product, in the sense that if performed correctly, the result can be converted back into a working program by leveraging existing compilation tools. However, it is not the most suitable form for further analysis. We add the following steps to improve analysis outcomes:

8. recovery of intra-procedural data flow, i.e. reconstruction of a partial SSA form, detailed in Section III-D,
9. analysis of inter-procedural data flow conducted through machine registers (i.e. an enhanced and more principled version of function argument and return value identification), described in Section III-E,
10. resolution of indirect procedure calls into chains of conditional jumps, which are much easier to reason about, described in Section III-F.

The prototype tool chain that we have built and evaluated uses LLVM IR as its common language, the tool `remill` for steps 2 and 3, the tool `mcsema` takes care of step 1 and of steps 4 through 7, and finally steps 8 through 10 represent our own contribution.

B. LLVM IR

The purpose of LLVM intermediate representation is to describe programs in an architecture-neutral way that is nonetheless relatively close to how hardware executes the program. The main differences between machine code and LLVM are roughly the following:

- LLVM uses symbolic addresses, i.e. elements of the program (both data and code) are labelled, instead of living at fixed numeric addresses,
- functions are represented by their control-flow graphs made of basic blocks,
- the data flow within functions is made explicit using partial SSA (single static assignment) form on virtual registers (i.e. SSA values cannot have their addresses taken)
- arguments and return values are passed explicitly into and out of functions.

The LLVM IR is a compromise between higher-level languages (such as C or its simplified dialects like CIL) which are more expressive, but harder to implement, and machine languages which lack information that is vital for program analysis. A distinct advantage of the LLVM IR is that it can be readily described in terms of small-step operational semantics using a comparatively simple machine. This is an important benefit in our work, since the same holds for CPUs.

C. Remill

The input of `remill` is a sequence of machine instructions (with boundaries already identified) and the output is a machine-independent encoding of the semantics of this instruction sequence. The output is built around two basic concepts:

1. a model of the machine state, which describes the registers available to the machine, along with abstract memory (which is simply assumed to be an array of bytes with LLVM-compatible semantics),
2. a set of *semantic functions*, one for each machine instruction,[2] which operate on the state: they take one state as an input and produce a new state, capturing the effect of the corresponding machine instruction.

In essence, these two components describe the small-step operational semantics of the CPU in question, in an executable form: the machine state is modelled using an LLVM data type (known as `regstate`), while each of the semantic functions

[2] A combination of opcode and operands, excluding the values (but not types) of immediate operands.

is defined as a sequence of LLVM operations on this data structure.

D. SSA Recovery

As outlined in Section III-C, the decompiled program operates on a value of type `regstate`, by applying successive transforms which correspond to the original machine instructions of the compiled program. The simplest implementation, then, simply stores this `regstate` in a global (or thread-local) variable and each *semantic function* simply reads and writes data directly through this global variable. While simple, this has severe disadvantages in terms of the data flow patterns in the decompiled program, where essentially all data flow goes through a single global variable.

This type of program behaviour is completely opaque to most types of static analysis (which relies heavily on tracing the data flow in the program). For purely dynamic analyses (like those implemented in KLEE), this is not an important obstacle, but any tool which relies on static techniques, even if it is only as an optimization, will be severely affected.

However, since we know the structure of the `regstate`, it is possible to significantly improve the transparency of the data flow by unpacking the content of `regstate` into individual scalars. The first step is an intra-procedural transform that recovers register-level data flow within a single function; the second step then extends this scalar decomposition to function calls (discussed in more detail in Section III-E).

Since the code that manipulates `regstate` all comes from the definitions of semantic functions, we know that the access is always well-behaved: there are no writes that would affect two registers at once, nor are there reads that would span a boundary between two registers. Therefore, the `regstate` can be first broken down into scalars, each register getting its own variable. Since these variables are guaranteed to never escape into different threads, all use of such variables can be freely converted into SSA form using a standard algorithm.

At this point, however, the `regstate` structure needs to be reconstructed before each function call and before each return, so the current version can be passed along. This has two negative effects (which will be addressed in the next section):

1. the data flow within the function across call sites is still entirely opaque: the outgoing version of `regstate` constructed as an argument to be passed down into the callee becomes a sink, and the updated `regstate` which the function returns becomes a source, interrupting the data flow graph at each call site,
2. the data flow in and out of functions also remains completely opaque: at this stage, it is nearly impossible to tell which registers carry data into and out of functions (i.e. function arguments), which are passed through unchanged and which values are simply dead.

Besides `regstate`, which is essentially an artificial construct that arises from the decompilation process, there is another effect obscuring the flow of data: stack manipulation. Again, in a straightforward decompilation, the stack becomes a single global variable. Therefore, when values are moved between the stack and registers, the same effect happens as was the case with `regstate` earlier: the values become impossible to trace. Unfortunately, the rigid structure of `regstate` made it fairly simple to break it up into a large number of independent scalars. The situation with stack is much less optimistic.

Analysis of stack access and subsequent data flow reconstruction is much harder for several reasons:

1. the operations which access stack come from the original program, not from semantic functions, and hence almost nothing can be safely assumed about their structure,
2. each function can, in principle, freely access all stack frames (i.e. it is not limited to its own stack frame),
3. stack addresses can escape into global variables and into other threads of execution, further restricting valid transformations,
4. program bugs (e.g. buffer overflows) cannot be reliably distinguished from intentional behaviour (in-bounds access to a stack-allocated array).

Our current approach does not further analyze data flow through the stack, though this is one of the outstanding issues that could improve results with backend tools that rely on static analysis.

E. Inter-procedural Data Flow Analysis

To address the problems arising from passing a monolithic `regstate` value from function to function, we have implemented a transformation which, at the level of LLVM, unpacks `regstate` in function calls into a list of scalar arguments. Of course, alone, this change does not add anything of value, but it allows the argument lists to be pruned. With return values, there is the additional challenge that LLVM does not allow multiple return values from functions, so the return sequence needs to be handled separately from the call sequence. Return values are represented using aggregate types, similar to the original `regstate` structure, but with unused fields left out. The function prototype before and after the transform is shown in Figure 1.

```
int foo( int );                      // original
void foo( regstate * );              // remill output
struct foo_rv { long rax, rsp };  // return type
foo_rv foo( long rdi, long rsp ); // post-DFA
```

Fig. 1. Function prototypes at various stages of decompilation.

Unlike for standard intra-procedural SSA conversion, there is no established algorithm for applying the equivalent transformation across function calls. We have therefore devised a simple algorithm based on iteratively pruning the argument sets. Each iteration proceeds in four steps, and the steps are repeated until either a fixpoint is reached, or a predetermined iteration limit exhausted:

1. execute the following transformations on the entire program (using their implementations provided by LLVM):
    1. scalar replacement of aggregates and loop-invariant code motion, which slightly improve the results

obtained from the following passes, but are not essential to the algorithm,
    2. SSA construction, which identifies function-local uses of memory (loads and stores) and replaces them with SSA,
    3. control flow simplification: removes unreachable basic blocks and merges sequences of basic blocks where applicable,
    4. dead store elimination: remove `store` instructions if no future `load` can read the stored value,
    5. dead code elimination: iteratively remove instructions with unused (dead) results,
    6. dead argument elimination: an inter-procedural transformation which eliminates unused function arguments (but see also below),
    7. instruction combine pass, which simplifies some common instruction sequences into a single instruction with the same effect;
2. examine each function and remove any arguments which only have a single use – the one in the return value of the function (i.e. when the register did not change in the function and is simply passed back to the caller),
3. for each function, if a field is unused in all returns from that function, it is eliminated from the return type,
4. for each function `f`, examine call sites which refer to `f`[3] and compute the intersection of fields of the return value which are unused by the caller; the values from the resulting set are then eliminated from the return type.

In optimal conditions, the above algorithm will narrow down the data flow to only contain registers which carry actual function arguments, and live callee-save registers which are used (and hence spilled) by the callee. However, passing values stored in callee-save registers into a function is redundant in correct programs: the subroutine simply makes a copy of the register on stack, and restores the value before returning. Unfortunately, even in straightforward functions, it is hard to prove that this is the case: the portion of the stack which stores the register can be overwritten and if that is the case, short-circuiting the `store` + `load` combination will alter the behaviour of the program. In many analysis scenarios, this is undesirable.

F. Indirect Call Resolution

One of the major obstacles to understanding program behaviour (whether by humans or algorithmically) is dynamic control flow: the simplest case are conditional jumps, which are quite well understood and comparatively easy to reason about. Function-local control flow without indirect jumps is typically encoded in control flow graphs, which are a simple, static objects which give a very good picture of the behaviour of the given function.

Subroutine calls and indirect jumps both constitute a slightly different type of additional inconvenience: in case of indirect jumps, especially in machine-level programs, the set of their targets is not always clear or even possible to compute reliably and the ideal of a static control-flow graph breaks down. Subroutines, on the other hand, pose a similar problem, but not at the subroutine entry point, since in direct calls, this could be easily captured with a standard CFG. Instead, the problem arises when control flow returns from the subroutine to the call site: the return address is, in this case, dynamic, since there are possibly many call sites which call into the same subroutine (that is, after all, the reason subroutines exist). As long as the calls themselves are direct (and hence the target of the call instruction is statically known), it is at least possible to enumerate the call sites to which a function could return, making inter-procedural data flow analysis feasible, if not easy.

Unfortunately, when indirect calls are involved,[4] the situation becomes much more problematic: the forward edge (the call itself) is no longer easily resolved (and again, in machine-level programs, the forward edges cannot be reliably enumerated for the same reasons as with indirect intra-procedural jumps). This also means that enumerating *call sites* for a given subroutine becomes much harder, since every indirect call site could possibly call any of the functions in the program.[5]

Since reconstruction of vtables and similar artefacts from machine code is error-prone and specific to a combination of platform, operating system and C++ compiler (in case of C++ vtables – with hand-coded function pointer tables, this becomes even more of a hit-and-miss affair), it is desirable to resolve indirect calls in a more general and automated fashion. To this end, we have devised an approach based on dynamic methods and gradual refinement, which replaces each indirect call with a direct call to a synthetic helper function, which only uses conditional branches and direct calls to replicate the effects of the original indirect call. The resulting structure is much more transparent to further inter-procedural control-flow and data-flow analyses.

The algorithm proceeds as follows:

1. replace each indirect call with a call to a *switch box*, that is, a call-site-specific synthetic function which takes the indirect address as an argument,
2. synthesize empty switch boxes for each of the call sites replaced in step 1 – an empty switch box indicates the desired target address it has been passed, and aborts execution,

[3] In the current implementation, the results of the indirect call analysis are not yet available at this stage. Functions which have their address taken are excluded from the analysis.

[4] Indirect calls often arise due to *late binding* in C++ programs: object instances which are capable of late binding carry a pointer to a so-called *vtable*, which, for each late-bound method, lists the address of the subroutine which implements the particular method in the given object instance.

[5] A simple heuristic that can improve situation in this case is that functions whose address never appears in the program outside of direct `call` instructions cannot be invoked using an indirect jump. However, the heuristic is not completely reliable and should be avoided in rigorous verification scenarios (it is entirely possible to store, for example, just an offset of the entry point from another known address, e.g. from another function, and reconstruct the entry point address at runtime, without the literal address ever appearing in program text). Nonetheless, in practice and for non-obfuscated programs, the heuristic works very well. Unfortunately, it does not cover some common patterns, like C++ virtual functions, or analogous constructs used in C programs.

3. abstract away all input values and explore the state space of the program, noting any errors raised by the switch boxes,
4. amend each switch box which appears in a counter-example as follows: a) add a conditional branch which checks the argument for equality with the address indicated in the counter-example, b) if they match, proceed to directly call this target function, c) otherwise proceed with the previous version of the switch box;
5. repeat steps 3 and 4 until no further counter-examples in the switch boxes appear.

The choice of abstract domain in step 3 then controls the trade-off between precision and cost: coarser abstractions lead to smaller state spaces and the algorithm proceeds more quickly. However, they also produce larger switch boxes, which in turn cause downstream analyses to give coarser results. The output program after the process no longer contains any indirect calls.

G. Limitations

Decompilation is, in itself, a complex and somewhat fragile endeavour, and rigorous analysis of software comes with its own set of problems and challenges. Combining the two brings unique difficulties, but at the same time resolves some of the weak spots of decompilation (for instance, some of the more common decompilation mistakes become much easier to uncover through use of formal analysis tools) and extends the field of applicability of formal methods. In this section, we will list some of the limitations of our current approach, along with a lightweight analysis of possible improvements.

1) Formal Semantics: The first problem that appears when applying formal methods is the semantics of the languages involved: this primarily affects the definitions of semantic functions (see Section III-C). On the CPU side, the programming manuals usually do not provide formal semantics – instead, they describe the effect of each instruction in natural language. The absence of a formal description can lead to misunderstandings during implementation of the semantic function: if we are interested in proper verification (as opposed to bug finding), an imprecise semantic function can, in principle, invalidate the entire verification result.

Of course, the decompiler is another complex piece of technology and as such creates additional space for implementation errors, which likewise affect the validity of outcomes.

2) Overflows: Some classes of errors in C and C++ programs are closely tied to the boundaries of variables in memory, whether local or global. A typical example of such bugs are buffer overflows. In a compiled program, however, there is no way to differentiate between writes to the stack that are ‘correct’ from those that are ‘incorrect’ with respect to the intended behaviour of the program. In a compiled program, both global variables and the stack are contiguous chunks of memory with no boundaries between variables as they were defined in the source code.

To an extent, this type of errors can still be detected in decompiled bitcode if debug information was available during decompilation, since the debug metadata retains information about individual source-level variables, including their size and location within the data section of the program (in case of global variables), or within the stack frame of the given function (in the case of local variables).

If debug information is not available, the best achievable outcome is heuristic: the decompiler can optimistically create boundaries between variables based on stack access patterns, and violations of such boundaries can then be analyzed and corrected if they turn out to be spurious.

IV. Evaluation

To evaluate our approach, we have started with a selection of small C programs, some of which were correct and others contained a single error each.[6] The source code of each of the programs was annotated with the location of the error, if applicable. The programs were compiled with Clang version 7 to obtain executables (machine code).

In order to check that errors in the programs are detected and reported correctly, all programs were compiled with debug information (compiler switch `-g`). The decompiler can then read this debug metadata and attach information about line numbers to the decompiled LLVM IR. The line numbers then appear in counter-examples generated by the analysis tools, and we use these line numbers to check whether the error annotation in the program matches the counterexample.[7]

The backend analysis tools that we have used were the following:

- KLEE with uClibc and the Z3 SMT solver [6],
- DIVINE configured to use STP as its SMT solver.

The benchmark programs were sorted into three categories:

Explicit – test cases with no inputs, targeting mainly safety properties and calls to external (library) functions, split into 3 sub-categories:

- C_1 – small programs that interact only with the simpler parts of the standard library (48 test cases in total),
- C_2 – same, but those programs make use of more complicated library functions like `longjmp` and POSIX functions and for this reason cannot be analyzed using KLEE (60 test cases),
- *threads* – a collection of programs that use either C11 threads or POSIX threads, which again excludes analysis with KLEE (73 test cases).

Symbolic – simple programs with inputs, split into:

- *finite* – programs that contain only finite execution paths (i.e. all paths through the program eventually terminate; 251 test cases),
- *infinite* – programs which loop forever for at least some inputs (15 test cases).

Svcomp – a subset of programs with inputs, from the SV-COMP test suite, split into four sub-categories: *recursion* (81

[6] Source code available at https://divine.fi.muni.cz/2020/decompile/.

[7] Debug information is not used for any other purpose during decompilation: even if it was not present in the compiled program, the decompiled bitcode would be semantically equivalent, though counter-examples would be harder to interpret.

	C$_1$	C$_2$	threads
valgrind	40 / 8 / 0	47 / 13 / 0	45 / 20 / 8
DIVINE^1	40 / 8 / 0	52 / 8 / 0	62 / 11 / 0
DIVINE^2	40 / 8 / 0	48 / 12 / 0	32 / 41 / 0
KLEE^1	35 / 13 / 0	-	-
KLEE^2	35 / 13 / 0	-	-

TABLE I

Results of explicit benchmarks. Each cell contains three numbers: the number of tests which passed, the number of tests which failed and the number of tests which timed out. Lines marked ^{1} were executed with indirect call resolution, while on lines marked ^{2} this step was skipped. The timeout was set to 10 minutes per testcase and the stack size to 4 KiB.

	finite	infinite
DIVINE	241 / 1 / 9	15 / 0 / 0
KLEE	240 / 1 / 10	-

TABLE II

A summary of (passed / failed / timed-out) test cases in the *symbolic* test category. The time limit per test case was 25 minutes and the stack size was set to 4KiB.

	recursion	array	bitvector	loops
DIVINE	78 / 1 / 2	110 / 5 / 0	14 / 6 / 0	74 / 13 / 1
KLEE	81 / 0 / 0	115 / 0 / 0	11 / 9 / 0	74 / 14 / 0

TABLE III

A summary of (passed / failed / timed out) test cases in the SV-COMP category. The time limit in this category was set to 10 minutes per test case, while the stack size was set to 64KiB.

test cases), *arrays* (115 test cases), *bitvectors* (20 test cases), and *loops* (88 test cases).

Each of the following sections describes the results in each of the above 3 categories in more detail.

A. Explicit

The main goal of this set of tests is to compare our approach with `valgrind`: among analysis tools which can directly work with compiled programs, it is the most advanced. However, since `valgrind` cannot operate on symbolic data, we chose tests that do not use symbolic data. In this comparison, `valgrind` runs on compiled programs while both DIVINE and KLEE use decompiled bitcode.

Tests of decompiled programs were executed for each verification tool twice, both with and without indirect call resolution (see Section III-F) enabled. In the first sub-category of tests, no indirect calls are present, and therefore the absence of the resolution does not alter the results. Remaining sub-categories contain some indirect calls (`pthreads` for example), and with the indirect call resolution with linked standard library used by the verifier (since indirect call happens inside the library function) the results are significantly better. The results are summarized in Table I.

Test failures in this category were caused by 3 types of problems:

- out-of-bounds memory access: as discussed in Section III-G, decompiled bitcode cannot reliably model variable boundaries – if the original program contains out-of-bounds access to a stack or a global variable, the decompiled bitcode will perform a seemingly valid operation, since both the stack and global variables are represented by large contiguous blocks of memory,
- external functions: since KLEE does not come with a complete implementation of the standard library, it fails when the test program uses less common library functions,
- ABI compatibility[8]: both DIVINE and KLEE use custom versions of the standard library, and as a result, if the source code is compiled with the system `libc` and then decompiled, differences in the ABI of these two libraries can cause spurious errors to arise.

Overall, however, the test results are quite encouraging. In most cases, our proposed approach is on par with `valgrind`, which was designed from the start to work with compiled programs.

B. Symbolic

In this category, many of the programs were tested with different optimisation levels, which typically results in different compiled programs and hence also different decompilation results. Each such variant was counted and evaluated separately. The results are shown in Table II.

All the timeouts that occurred with the DIVINE backend were caused by a compiler optimization which replaces a ‘modulo’ operation with an equivalent multiplication. If optimisations are disabled, compilers will usually emit an `idiv` (on `x86_64`), which is expensive when executing on actual hardware. Therefore, optimizers instead emit a longer instruction sequence which performs better: unfortunately, this sequence contains a multiplication by an inverse constant, which the tested SMT-solvers were unable to perform within a reasonable time limit. In some programs, the problem disappears when the optimizations are set to a sufficiently high level, since the compiler then eliminates the ‘modulo’ operation entirely.

KLEE times out in one additional test case, when compared to DIVINE. Given more time (approximately two hours), it managed to complete each of the test cases that completed with DIVINE as the backend. However, given the relative simplicity of the test cases, we do not consider this to be a success.

The only failure which appears in this set is caused by an out-of-bounds access to a local variable, which was not detected due to limitations in the decompilation process.

Overall, the results in this category are almost as good as we could expect from the verification of the source code. Most of

[8] The main source of ABI compatibility problems are differences in the sizes or layouts of library-defined data types (for example, as a result of extra padding). Such mismatches can then lead to incorrect memory access (in the better case), or to quiet but wrong result in the worse case.

the few timeouts are not an error of the decompilation process but rather a limitation of used SMT solvers.

C. SV-COMP

None of the SV-COMP benchmarks contain indirect calls, hence all the results in this section were obtained with indirect call resolution disabled.

While in the previous categories, identifying the causes of test failures was relatively easy and straightforward, in this category it is not the case. The failures are a mix of the verifier not handling the bitcode properly and of incorrect answers, the origin of which is not easily determined.

The overall results, summarised in Table III, are similar to the other two benchmark sets and can be considered satisfactory. Both verifiers achieved a similar success rate, while the total number of failed test cases is below 15%, with the exception of the *bitvector* sub-category, where the results were quite poor.

V. Conclusions & Future Work

We have presented a tool chain for applying formal methods to programs in executable form, built largely out of existing components and technology, augmented with a few novel transformations which significantly improve analysis outcomes. While the implementation is only a prototype, the evaluation results are encouraging, and we believe that they demonstrate the viability of the approach.

There is a number of directions in which the present work can be extended. Improvements can be clearly made in the inter-procedural data flow analysis described in Section III-E, both with regards to handling of indirect calls, along with improved analysis of caller-saved registers. The other major area for improvement is detection of variable boundaries, the lack of which currently accounts for the majority of the deficiencies uncovered in our evaluation.

References

[1] Radare2. URL https://github.com/radareorg/radare2.

[2] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *ICSE*, page 1083–1094, New York, NY, USA, 2014. Association for Computing Machinery. doi: 10.1145/2568225.2568293.

[3] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of C and C++ with DIVINE 4. In *ATVA 2017*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017. URL https://divine.fi.muni.cz/2017/divine4.

[4] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, LNCS, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[7] Patrice Godefroid. Compositional dynamic test generation. In *POPL*, page 47–54, New York, NY, USA, 2007. ACM. doi: 10.1145/1190216.1190226.

[8] Hex-Rays. IDA Pro disassembler and debuger, 2020. URL https://www.hex-rays.com/products/ida/.

[9] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 423–427, Berlin, Heidelberg, 2008. Springer.

[10] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. ISSN 0001-0782.

[11] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, 2004.

[12] Henrich Lauko, Petr Ročkai, and Jiří Barnat. Symbolic computation via program transformation. In *Theoretical Aspects of Computing – ICTAC*, pages 313–332, Cham, 2018. Springer.

[13] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. ISBN 0792393805.

[14] Barton P. Miller and Andrew R. Bernat. Anywhere, any time binary instrumentation. In *Program Analysis for Software Tools and Engineering (PASTE)*, Szeged Hungary, September 2011.

[15] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[16] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin, editors. *Principles of program analysis*, 1999. Springer. ISBN 978-3-540-65410-0. doi: 10.1007/978-3-662-03811-6.

[17] Trail of Bits. Mcsema, 2020. URL https://github.com/lifting-bits/mcsema.

[18] Trail of Bits. Remill, 2020. URL https://github.com/lifting-bits/remill.

[19] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[20] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[21] Sai Vegasena and Peter Goodman. Klee-native, 2019. URL https://blog.trailofbits.com/2019/08/30/binary-symbolic-execution-with-klee-native/.