Incomplete Symbolic Execution with Monster: Systems

Alexander Linz
Advisor: Professor Christoph Kirsch

Paris Lodron University Salzburg, 5020 Salzburg, Austria
Department of Computer Sciences

Abstract. This thesis presents an overview of the Monster Project, an incomplete symbolic execution engine and solver. Symbolic execution is the procedure of executing a program not on concrete inputs but rather on symbolic values. An engine in this context is providing formulas to be solved, which indicate a bug in the program if they are satisfiable. This theses presents the components of Monster, mainly: The loader, the engine and the solver; all of which are designed to work independently and be interchangeable. Additionally, this theses shows the various third party solvers that have been integrated with the Monster engine and why Monster is limited to the Selfie system and thus only works with C* code and RISC-U instructions. Monsters solver follows a blueprint laid out by Aina Niemetz and her colleagues in a recent paper, which proposes that using an incomplete solver improves solving speeds for symbolic execution drastically.

Keywords: Selfie · Monster · Boolector · SAT · SMT · symbolic execution

Table of Contents

1 Introduction

Fixing and finding bugs in code is sadly, more often than not, the primary time sink of modern programming[2]; even with sophisticated integrated development environments and best practices, bug fixing is inevitable and essential. However, even when spending such a significant amount of time finding and fixing bugs, some still slip through the cracks. A great example of this is the 2014 Heartbleed Bug in OpenSSL, which allowed for the reading of all memory supposedly protected by encryption. There are many more examples of critical bugs that cost companies hundreds of millions of dollars.

Symbolic execution offers a potential solution to this problem by automatically finding bugs in code and guiding programmers towards a fix. Current symbolic execution engines are far from perfect, but their potential has been steadily increasing as new and improved methods are discovered. With this in mind, we implemented and tested symbolic execution using techniques proposed by Aina Niemetz[5] and her colleagues. By accepting that determining whether a program has bugs is reduceable to the halting problem and thus incomplete, they concluded that implementing an incomplete SMT solver could speed up the search for bugs. Monster is our attempt to reproduce the improvements they presented on SMT formulas and implement them in a symbolic execution engine that is centred around Selfie.

To test how well Monsters solver performs, many benchmarks were performed, some of which are presented in this theses, that show that the solver does achieve great speeds compared to more traditional approaches but quickly falls behind when the complexity of the given program increases to levels of real-world scenarios.

What we did achieve though were three distinct components that work great together but can also be just as easily used separately. Firstly, a loader that disassembles binaries compiled by Selfie into a data structure recognized by Rust, representing the RISC-U instructions. Secondly, an engine for the solver that executes the instructions provided by the loader symbolically and generates SAT queries that indicate bugs if they are found to be satisfiable. Thirdly, the solver itself which tries to solve the queries provided.

This theses will first give some insights into the problem that symbolic execution tries to solve as well as symbolic execution itself. In the second chapter the components of Monster will be described in greater detail. The third chapter presents our testing efforts and benchmarks.

Our project was realized with help from the Computational Systems Group at the Department of Computer Sciences of the University of Salzburg and thus worked in tandem with the Selfie[3] project of Christoph Kirsch.

1.1 The Problem

Monster, and other symbolic execution engines, try to solve the problem raised in the introduction by providing an automated way of finding bugs in programs. However, this turns out to be a complicated task, but with a massive payoff in debugging time and money saved, and it might just save lives with automated systems on an ever-increasing rise in popularity for more and more crucial systems.

Modern programming paradigms are acutely aware of this and try their best to herd programmers in a direction where bugs are detected early or even avoided by design. However, it turns out humans are not machines and make mistakes; symbolic execution could fill some of those gaps, but challenges still need to be overcome to make this approach viable.

1.2 Symbolic Execution

To understand the problem that Monster tries to solve, one must first understand what exactly symbolic execution is. That is why in this section, a summary of how symbolic execution works and how it is used will be given.

Symbolic execution is the process of executing a program not on concrete input values but rather on symbolic values (these behave like unknown variables within a range that is valid for the input data type). Executing a program in this way allows the engine that is doing the execution to determine which statements of the program can be reached by which combination of concrete inputs. That knowledge can then be used to determine if a statement can produce errors or if a specific state of the program is possible.

```
1 unsigned int x = read();
2
3 if (x < 1) {
4     x = 10 / x;
5 } else
6     return x;
```

Fig. 1. Pseudocode example (32 bit)

To illustrate how a simple symbolic execution engine would act, fig. 1 provides a piece of pseudocode that will divide by zero in line 4 if the input variable "x" reads a 0. A concrete execution of this example reads some specific value for "x" (e.g., 1) and then executes the program with that input; in this case, that would return 1.

Symbolic execution instead assigns "x" a symbolic value (often λ, Monster uses "x0" as the first symbolic value). In combination with some constraints (like $\lambda < 50$), this value represents the range of values this symbolic value can be. In our case, the first constraint for λ is that it must be in the range of an unsigned int ($0 \leq \lambda \leq 2^{32} - 1$), the second constraint is generated at line 3. Here the symbolic execution splits into two paths: $\lambda < 1$, and another where $\lambda \geq 1$. For the first path, it is then quite obvious that $\lambda = 0$ since "x" must follow all other constraints as well. This then leads the engine to conclude that $10/\lambda$ could (and in this case always will) result in a division by zero.

However, one of the most significant features of symbolic execution is not just that it can provide a line of code that could produce an error/bug; it actually, by design, also always finds a concrete set of inputs that produces this error. This set is often called a "witness" and enables reproduceablility of the error in question and also allows programmers to see why exactly an error has occured and what lead to it.

This example is a straightforward piece of software, but it already displays how a program could be symbolically executed and shows one of the biggest problems of symbolic execution called "Path Explosion". This will be discussed more thoroughly in section 1.3.

1.3 The Problem with Symbolic Execution

As mentioned before, the most glaring problem for symbolic execution is called "Path Explosion". Since symbolic execution has to check every possible path through a program, it scales terribly to more extensive programs with thousands of branching statements. Since every branching point not only creates another path but also creates an additional path for each branching point afterwards, effectively doubling the number of paths that need to be checked. This leads to a generally exponential ”explosion” of the number of paths. Even worse, loops can introduce unlimited branching points in a program, with which come unlimited paths.

There is no real solution to this problem, but Monster implements some well-known ideas to at least curb some of the problems by having an intelligent path selection algorithm, that can determine which paths should be prioritized. The implementation of this algorithm is described in Christian Moesls thesis on Monster (Sec. 1.6).

Since even with an intelligent algorithm, it takes an incredible amount of time to execute a program symbolically, any time savings can be massive in the practicability of symbolic execution as a whole. This is why the new approach first presented by Aina Niemetz[5] and her colleagues and now implemented by us as well as something that could pave the road for widespread adoption of this tech.

1.4 C* and Selfie

Monster has been created with close ties to a project called Selfie[3]. Selfie is still being developed and was started by the Computational Systems Group at the Department of Computer Sciences of the University of Salzburg as an educational platform to teach students about programming languages, compilers, and operating systems. It includes a compiler that can compile a subset of C called C* (C Star) to a subset of RISC-V called RISC-U, an emulator that can execute the RISC-U code, a hypervisor that can provide simple virtual machines that can host Selfie itself, and a C* library of simple and often used functions. These parts only represent the basic implementation of Selfie, the project is being extended continuously, and the most up-to-date version can always be found on the GitHub page of the project. Due to Selfie using subsets of both C and RISC-V, we were able to more simply implement all of Monsters’ features without being bogged down by unnecessary complications; this also means that Monster only works with RISC-U binaries compiled by Selfie.

1.5 Monster and its Capabilities

With all this in mind, Monster was designed as an ”Incomplete Symbolic Execution Engine” and is written in Rust. Rust was chosen since it can provide an incredibly stable and memory-safe platform with all the advantages of C or C++ but almost zero of the drawbacks. Monster takes already compiled (by Selfie) binary files in ELF format and executes them symbolically to find a specific set of bugs. In detail, Monster can also disassemble ELF binaries, generate a control-flow graph from the product, and generate a data-flow graph for the program.

Bugs

Monster is a somewhat limited test of the concept and thus only recognizes a small set of bugs as listed in table 1. This is very easy to extend, but for us, the basics are enough to see how effective Monster is compared to other implementations.

Internal Name	**Reason**
ExitCodeGreaterZero	Exitcode unequal 0
DivisionByZero	Division by 0
AccessToUnitializedMemory	Access to uninitialized memory
AccessToUnalignedAddress	Access to unaligned memory address
AccessToOutOfRangeAddress	Access to memory address out of range

Table 1. Known bugs defined by Monster

1.6 Collaboration

Monster as a project has had a varied group of contributors since its inception in 2020. A current list can always be found on the Github page of Monster [1]. For the specific timeframe this thesis is concerned with, the leading group consisted of Christian Moeßl, Alexander Lackner, and myself. We also had some contributors who only worked with us for a limited time, mainly Fabian Nedoluha. In addition to these people, we received help figuring out the concepts from the whole Computational Systems Group at our university.

As part of this collaboration, we are producing three distinct but inherently linked bachelor theses', all of which share the name: "Incomplete Symbolic Execution with Monster". Christian Moesl gives an in-depth view of the engine we developed, Alexander Lackner's thesis is concerned with the solver part of Monster, and mine will provide a more general overview and more insight into how we tested Monster.

1.7 Contribution

Since the leading group of Monster consists of three bachelor students, we worked closely together for almost every stretch of the way, and most things contributed to the project can not be clearly attributed to a single person.

I was mainly concerned with benchmarking, validating, and testing our solver. This was done in comparison to two other solvers.

[1] `https://github.com/cksystemsgroup/monster`

2 The Elements of Monster

Monster is based on a simple design that logically divides responsibilities and defines clear interfaces between the stages of loading, executing, and solving. In the loading stage of a complete run of Monster, the loader prepares an ELF binary file to be provided for the engine. This involves the disassembly of the binary and construction of a control-flow graph. The engine, in turn, takes this as an input to start the symbolic execution. While executing, the engine produces SMT queries that need to be solved. This, of course, is done by the solver.

The following chapter will more thoroughly explore the three main pieces and the implications of this type of architecture.

2.1 Loader and Input

The input for Monsters loader must be a binary file in ELF format compiled by Selfie (Section 1.4). The current loader and Monster as a whole make the assumption that the input follows this simple rule.

This is necessary since the binaries do not contain all the information Monster needs to execute a given program symbolically and detect all the bugs it can. However, if the compiler is known, Monster can assume all the extra needed information from the construction of the compiler itself. This simply means that the assumption of having Selfie compiled C* is just a limit Monster has to operate under. If Monster is loaded with a non-compliant binary file, the results are uncontrollable and almost certainly incorrect.

With this in mind, if all prerequisites are met, the loading process starts by disassembling the binary file and generating a directed graph where each node represents a single RISC-U instruction. The edges are added based on all possible ways the program counter can change from instruction to instruction. Special care must be taken when considering the BEQ, JAL, and $JALR$ instructions. A more granular view of this part can be gleaned by referring to the engine-specific thesis of this project mentioned here: Collaboration.

A simple example (Fig. 2) will already generate a very large graph, this is why, for demonstration purposes, fig. 3 only shows a part of the whole picture.

After finishing the graph, the secondary responsibility of the loader is to construct paths through the graph. The design of Monster allows the loader to be easily extended here; for our purpose, we implemented a shortest-path algorithm (which is the default) and, for testing purposes, a "coin-flip" algorithm that makes all branching decisions at random. The control-flow graph generation is only done once per binary file, while the engine can request the loader to keep generating paths. These paths are used by the engine to execute the code symbolically and generate useful SMT queries. How that is handled will be discussed in the following section.

2.2 Engine

Once the loader has constructed a control-flow graph for a given input program, the engine can start its work. In general, the engine has one job: determine if

```
uint64_t main() {
    uint64_t *x;

    x = malloc(8)

    read(0, x, 8);

    if (*x < 1) {
        *x = 10 / *x;
    } else
        *x = *x * 0;

    return *x;
}
```

Fig. 2. Example C* code for control-flow graph generation

a given input contains a known bug (Table 1). At first glance, this seems not too complicated, but, as discussed in section 1.3, it generally is an impossible problem to solve entirely. With this in mind, the approach[5] to the problem which our project heavily relies on accepts its incompleteness and even leans into it. More specifically, the engine has to generate SMT queries that indicate a bug if they are satisfiable. To construct these queries, the engine equates the reachability of a statement with the satisfiability of an SMT formula. How this works can be seen in fig. 4.

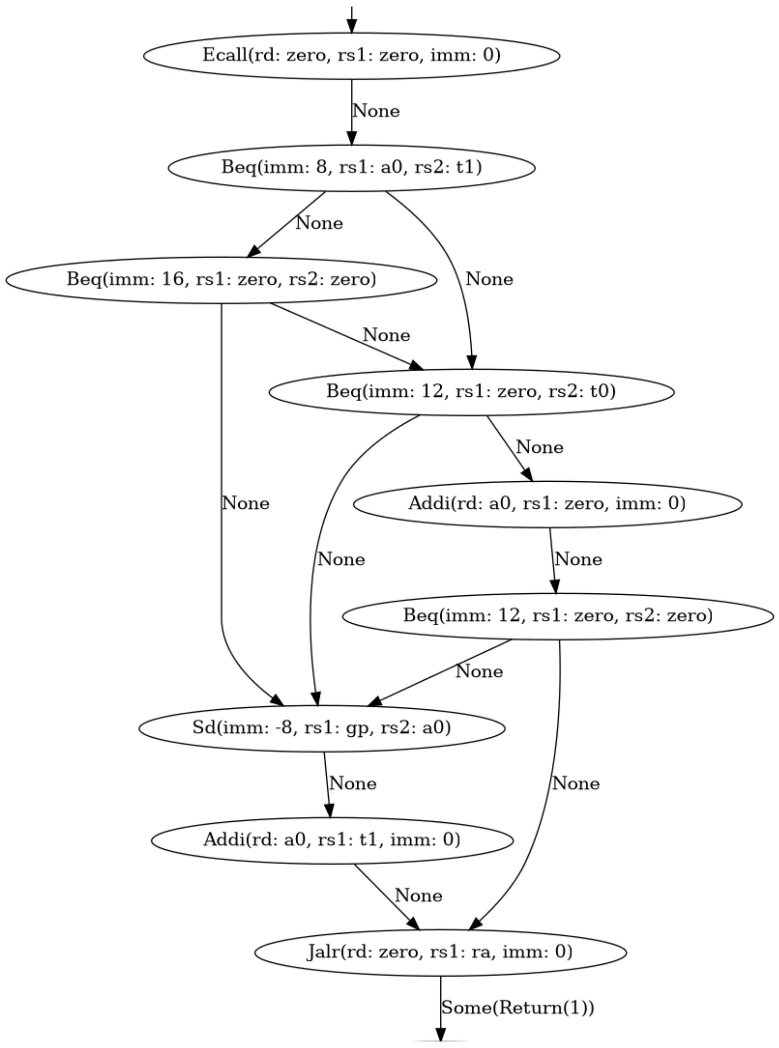
Once the engine has determined a place where a known bug can occur, a SMT formula is generated that represents the reachability of the line of code, and an appropriate addition to it is made to detect if a known bug occurs. An example of this addition would be: "$\wedge\ \sigma = 0$", for the division by zero bug (where σ is the divisor in a statement).

A complete SMT formula in Monster is represented as a data-flow-graph. The solver then uses this graph to determine if a bug can actually occur there.

2.3 Solver

Once a SMT formula in the form of a data-flow-graph has been given to the solver, it now has to determine if the formula is satisfiable. More traditional[1] (or complete) SMT solving approaches end with either a "satisfiable" (SAT), "unsatisfiable" (unSAT) or "unknown" result.

Monster mimics the approach of Aina Niemetz et al.[5] and is inherently incomplete. This means that it can never determine if a SAT formula is unSAT because the solver will keep trying to find a satisfiable assignment. The reasons for this will be discussed later in this chapter and more specifically in the solver part of this collaboration mentioned in section 1.6.



Ecall(rd: zero, rs1: zero, imm: 0)
None
Beq(imm: 8, rs1: a0, rs2: t1)
None
None
Beq(imm: 16, rs1: zero, rs2: zero)
None
Beq(imm: 12, rs1: zero, rs2: t0)
None
None
Addi(rd: a0, rs1: zero, imm: 0)
None
None
Beq(imm: 12, rs1: zero, rs2: zero)
None
Sd(imm: -8, rs1: gp, rs2: a0)
None
None
Addi(rd: a0, rs1: t1, imm: 0)
None
Jalr(rd: zero, rs1: ra, imm: 0)
Some(Return(1))

Fig. 3. Control-flow graph for a the "If" part of the example code.

```
if (condition_1) {
    if (condition_2) {
        statement
    }
}
```

$$(condition\_1) \land (condition\_2) \implies reachable$$

Fig. 4. Basic example on how to equate Reachability and Satisfiability.

The two most important consequences of this behaviour are that Monster can never determine if a program is actually bug-free, and Monster can solve more quickly than traditional solvers since it can "cut corners". Incompleteness might first look like a big problem, but since the whole field of symbolic execution is, in general, incomplete[6] itself, the incompleteness of Monsters solver does not negatively impact it much.

To solve a formula, the solver starts by trying to propagate a random set of inputs down through the data-flow graph to achieve a satisfied result; in other words, it tries the inputs to see if they result in a "True" output. If this fails, it just sets the output so that it must be "True" and tries to change the inputs, within their constraints, to match a set of inputs that result in "True". This is done by propagating back up the data-flow graph and trying to find a node with an operant that represents an error in our perfect "True" result. Although the selection of which operands are chosen is essentially random, Monsters solver takes a lot of care to prefer/prioritize operands that produce the requested result. This ensures we hit satisfying inputs (for the overall formula) quickly. The chosen operand is then manipulated by changing the input, to try and change the nodes results in a favourable way. This is just repeated until the program finds a set of inputs that satisfy the SMT formula or a timeout runs out. Due to the incomplete nature and random selection of inputs, it can never guarantee a correct result. This is why we implemented a timeout that will return an "Unknown" result.

3 Monster in Action

Monster is implemented in Rust, making it easy for anyone to compile and run as a standalone program. A setup guide can be found on the official Github[2] or crates.io[3] site of Monster.

[2] `https://github.com/cksystemsgroup/monster`
[3] `https://crates.io/crates/monster-rs`

This section will elaborate on how to read the output provided by Monster and also further describes why Monster behaves in specific ways. Section 3.3 will then show how Monster was evaluated and how the solver compares to traditional approaches.

3.1 Invoking Monster

With the help of the example in fig. 5 I will show the basic symbolic execution function of Monster, how to invoke it, and how to interpret its output. To follow this example, one also needs the current version of Selfie to produce the binaries needed as input.

Monster provides two main functions at this point: symbolic execution and control-flow graph generation. It can also be used to disassemble ELF binaries produced by Selfie.

Symbolic execution is invoked with the "execute" command and control-flow graph generation with the "cfg" command. Both can be modified with the "verbose" flag to gain a more refined understanding of what Monster is doing.

3.2 A Trivial Example

```
uint64_t main() {
    uint64_t *x;

    x = malloc(8)

    read(0, x, 8);

    if (*x < 1) {
        *x = 10 / *x;
    } else
        *x = *x * 0;

    return *x;
}
```

Fig. 5.

After compiling the code with Selfie into a binary, (from here on called: "trivial_example.o") Monster can be invoked as a symbolic execution engine using the "execute" command. The command can be modified by providing flags to edit the fundamental variables of Monster, like maximum execution depth or the solver to use; the default values are: The Monster solver, 1 Mib memory, max execution depth of 1000, and the shortest path exploration strategy.

Doing this produces the output shown in fig. 6. The output here indicates that a bug has been found and which of the known bugs (fig. 1) it is. In this case, the first bug found is a possible division by zero when the program counter is $0x000101bc$. The set of inputs (also called: a "witness") can then be used to reconstruct the context needed to produce the bug and represent the proof of the bug existing. Monster also bundles some formulas, that modify the inputs, into the witness. Each value in the witness is given an arbitrary name (x0, x1, ...) and is defined either by a concrete value or by a formula. If the value contains a formula, the result is displayed in brackets to the right.

The most important values inside our witness representation are the inputs of the program; here, the input is provided via a "read" with the "x0" value, which reads a 0. All in all, one can easily see, when looking at the code, that reading a zero in line 6 of the example always leads to a division by zero in line 9. Once this bug is found, it can be fixed in the code, and Monster can be rerun to try and find other existing bugs.

```
$ monster execute trivial_example.o

generate CFG from binary (took 238.009us)
unrolling CFG (took 613.208us)
computing distances from exit on unrolled CFG (took 915.667us)
computing shortest paths in CFG (took 1.592784ms)
bug found:
reason: division by zero
pc: 0x000101bc
witness: [
  x0 := "read(0, 69640, 8)[0 - 8]" (0),
  x1 := 1,
  x2 := x0 < x1 (1),
  x3 := 0,
  x4 := x2 = x3 (0),
  x5 := !x4 (1),
  x6 := 0,
  x7 := x0 = x6 (1),
  x8 := x5 & x7 (1),
]
```

Fig. 6. Execution of the trivial example given in fig. 5.

3.3 Evaluation & Validation

Monster comes bundled with two additional solvers: Boolector[1] and Z3[4]. Both can determine SAT, unSAT, and unknown for the purposes of solving a SAT formula. Additionally, both are proven to work correctly, and thus we can use them to validate the results of our own solver. We do need to consider how we would interpret Monster resulting in unknown (after the timeout) and a third party solver resulting in unSAT; since Monster can not deliver an unSAT result, we consider an unknown result as unSAT for this purpose.

When evaluating Monster, we also tested the speed against both; we would expect to be faster in most easy cases, but the more complicated a program is, and the deeper the bug is inside that program, the slower Monster gets comparatively. The result of this testing will be presented in the following chapter.

3.4 Evaluation Procedure

Rust comes packaged with a powerful benchmarking tool, but for this evaluation, Criterion.rs was used (found here: Criterion[4]).

This evaluation aimed to test the speed of our own solver against both Boolector and Z3. For this purpose, we were provided with a whole suite of benchmarking files based on the SV-COMP benchmarks and translated by the Computational Systems Group. The originals can be found here: sv-benchmarks[5], and the translated ones here: cstar-benchmarks[6]. The results of 3 specific files, which can all be found in the appendix (Sec. 5), will be presented.

Tests were performed on a 64-bit Linux system with an AMD processor, and special care was taken to mitigate interference with the tests by other processes on the machine. All tests had the same conditions, the only changing variables were the specific file that was benchmarked and the solver. A run was marked as complete once the engine produced a correct result. In the presented cases this always meant finding a bug.

[4] `https://github.com/bheisler/criterion.rs`
[5] `https://github.com/sosy-lab/sv-benchmarks`
[6] `https://github.com/cksystemsgroup/cstar-benchmarks/`

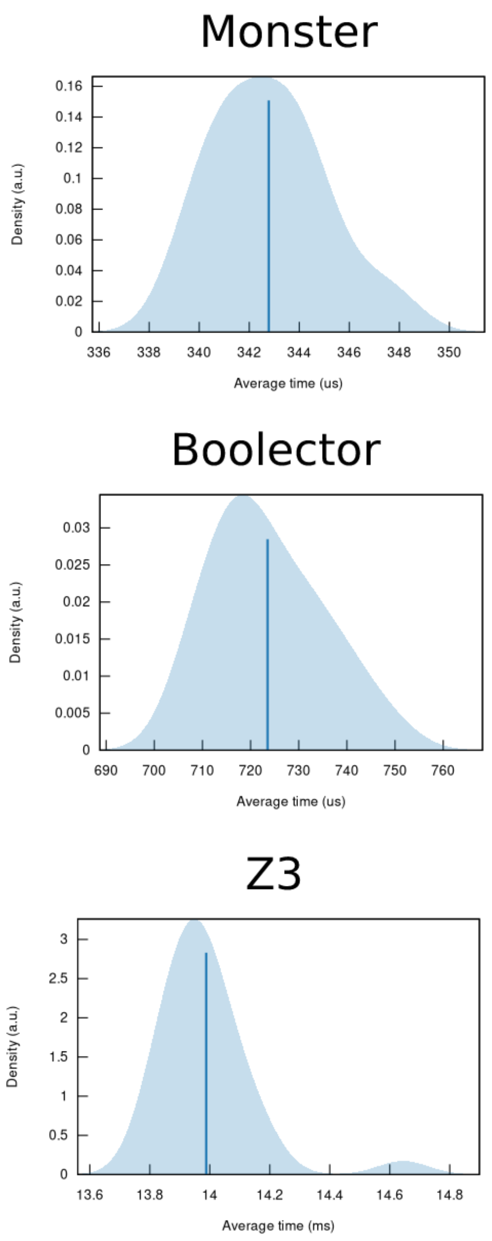


Fig. 7. simple.c Comparison

This first test was performed with a straightforward program that produces an error in all cases. The only tricky part is a while-loop that loops a number of times equal to a symbolic value. Fig. 7 shows that Monster can handle this case very well compared to Boolector and Z3. Monster clocks in at a median of $343us$ compared to $724us$ and a huge $13963us$. Z3, in particular, seems to struggle a lot more, this is the case because Z3's basic solving algorithm is not designed to be used for simple program. Z3 expects a lot of additional information about the program to choose the best tactic, which we do not provide, that is why it chooses the default tactic.

Since the path selection algorithm is the same and all tests are done in the same environment it stands to reason that our solver just simply outperforms both traditional ones.

Additionally, it can be said that Monster has a significant advantage in such simple cases as it is less optimized for complex programs. Since optimization often comes with significant overhead, which is not outweighed by the increase in efficiency when it comes to simple and small programs. Boolector seems to handle this trade-off better than Z3.

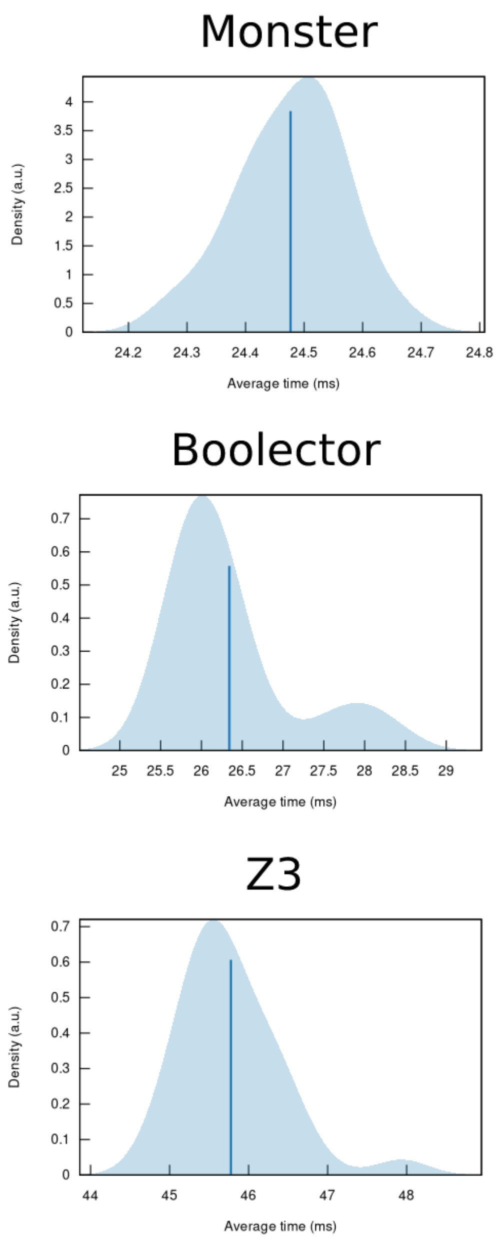


Fig. 8. count.c Comparison

The *count.c* example is just a little bit more complex in nature as the while-loop is not directly formed with a read value but rather with a symbolic value dependent on the symbolic input. However, the result is not more complex since, again, every number will work to produce a division by 0.

Fig. 8 does show that growing program complexity diminishes the lead Monster has. This time Monster takes a median time of $24.5ms$, Boolector $26.5ms$, and Z3 $45.8ms$. With the first case, Monster was 47% faster than Boolector, which shrunk to 8% here.

The example still seems to simple for Z3 to perform well, but this will change drastically in the last example.

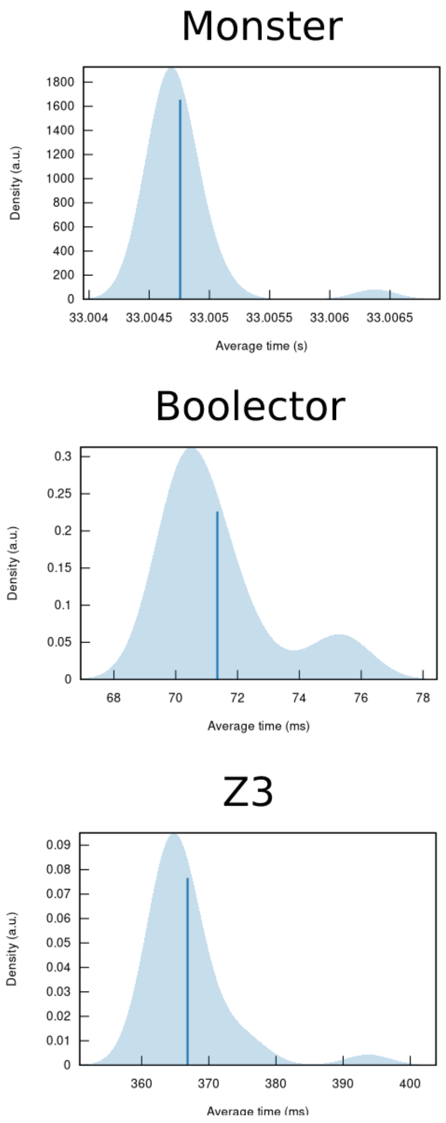


Fig. 9. sum.c Comparison

Here is where the nature of Monster begins to show. It should be pointed out that in this specific case, the graphs shown in Fig. 9 do not have the same unit of measurement for the average time axis, as Monster is just abysmally slow. With a median of $33s$ compared to Boolector's $71ms$ and Z3's $368ms$.

There is a straightforward reason why Monster performs so badly: Timeouts. In the *sum.c* example the while-loop has to run at least 33 times before a bug can be found. Since Monster can only determine the reachability of a line of code with specific inputs but not the unreachability, it does not terminate an attempt until the timeout is reached (which for this test was set at three seconds).

Boolector and Z3 can actually determine unreachability and thus continue to the next iteration of the loop sooner, cutting down the time to get to the 33rd iteration and finding the bug.

Results

The Monster solver does perform rather well when testing short and straightforward programs. It can even outperform the traditional solvers by a significant margin. However, once programs become more complex, the unrefined nature of Monster starts to show, increasing the time to result significantly. Especially the most significant advantage of Monster, incompleteness, decreases its efficiency by a lot in these cases. We think that the results still show that the incomplete approach to solvers can be viable when done right.

4 Conclusion

Monster as a project set out to try and replicate the results Aina Niemetz and her colleagues demonstated in their paper within the relatively small scope of a bachelor's thesis. Additionally, our project led to more advanced projects being started at the department. The benchmarking results show that even a simple implementation of the advanced concept of an incomplete solver can improve on some of the traditional ways of symbolic execution. It is clear that this specific version of Monster is not intended for use in large-scale symbolic execution, but it provides a proof of concept which others can replicate and improve upon.

4.1 Future Work & Improvements

Projects based on Monster have already started to pop up and do their own research and improve Monster itself.

One improvement we did think of but was not implemented is the cutting down of timeout duration. More often than not, the duration of the timeout is actually the limiting factor of Monster's efficiency. For now, it was set at three seconds, but this value is more or less arbitrary. Significant improvements could be made by implementing a variable timeout duration based on heuristics.

Another form of improvement for Monster would be the combination with an already established complete solver. In this case, we would suggest combining Monster with Boolector and use their strengths to complement each other.

5 Appendix

```
void  VERIFIER_error() {
    uint64_t x;
    x = 10 / 0;
}

void VERIFIER_assert(uint64_t cond) {
    if (cond == 0) {
        VERIFIER_error();
    }
    return;
}

uint64_t main() {
    uint64_t *n;
    uint64_t x;
    uint64_t y;

    n = malloc(8);

    read(0, n, 8);

    x = n;
    y = 0;

    while(x > 0) {
        x = x - 1;
        y = y + 1;
    }

    VERIFIER_assert(y != *n);
}
```

Fig. 10. count.c

```
void VERIFIER_error() {
    uint64_t x;
    x = 10 / 0;
}

uint64_t a = 2;

uint64_t main() {
    uint64_t  i;
    uint64_t* n;
    uint64_t  sn;

    n = malloc(8);

    read(0, n, 8);
    sn = 0;
    i = 1;
    while (i <= *n) {
        if (i < 10)
            sn = sn + a;
        i = i + 1;
    }

    if (sn == (*n)*a)
        return 0;
    else if (sn == 0)
        return 0;
    else
        VERIFIER_error();
}
```

Fig. 11. sum.c

```
void VERIFIER_error() {
    uint64_t x;
    x = 10 / 0;
}

void VERIFIER_assert(uint64_t cond) {
    if (cond == 0) {
        VERIFIER_error();
    }
    return;
}

uint64_t main() {
    uint64_t  x;
    uint64_t* N;

    N = malloc(8);

    x = 0;
    read(0, N, 8);

    while (x < *N) {
        x = x + 2;
    }

    VERIFIER_assert(x % 2);
}
```

Fig. 12. simple.c

References

[1] Robert Brummayer and Armin Biere. “Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Stefan Kowalewski and Anna Philippou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 174–177. ISBN: 978-3-642-00768-2. DOI: `10.1007/978-3-642-00768-2_16`.

[2] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison Wesley, Oct. 2002. ISBN: 0-321-11742-5.

[3] Christoph Kirsch. *The Selfie Project*. URL: `http://selfie.cs.uni-salzburg.at/`.

[4] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: `10.1007/978-3-540-78800-3_24`.

[5] Aina Niemetz, Mathias Preiner, and Armin Biere. “Propagation based local search for bit-precise reasoning”. In: *Formal Methods in System Design* 51.3 (2017), pp. 608–636. DOI: `10.1007/s10703-017-0295-6`.

[6] German Vidal. “Closed Symbolic Execution for Verifying Program Termination”. In: Sept. 2012, pp. 34–43. ISBN: 978-1-4673-2398-7. DOI: `10.1109/SCAM.2012.13`.