Incomplete symbolic execution with Monster: Solver

Alexander Lackner
Advisor: Professor Christoph Kirsch

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

Abstract. Monster is a symbolic execution engine that can use different SMT solvers to detect bugs in C* programs. Solvers are the bottleneck of symbolic execution engines, therfore a faster solver directly leads to faster symbolic execution.
Since symbolic execution is incomplete by design, we took this a step further, implementing and designing an incomplete solver based on Aina Niemetz's work.
Our solver being incomplete does mean that it can not decide if an SMT formula is unsatisfiable. We use this to our advantage by not searching the entire set of assignments but instead a smaller subset. This should lead to a faster and less accurate solver.
We reached the following results benchmarking against Boolector and Z3, while Monster was able to be competitive when shorter and simpler programs were being fed into the engine, it struggled with long and more complex programs with no or hard-to-find solutions meaning its performance dropped significantly.
It did however always return the same results as its counterpart, showing that its accuracy is just as high which is a feat itself given the short timeframe and being constructed by three bachelor students.

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

Table of Contents

1 Introduction

Monster is a fully functional symbolic execution engine that can detect bugs in compiled C* programs by translating it into SMT-Lib and then solving these queries with an SMT-solver.
We implemented and designed a solver based on the works of Aina Niemetz, who published a paper in 2017 introducing a new approach to how the solver of a symbolic execution engine might be constructed.
While Aina Niemetz did show that the implementation was not as fast as other approaches, it was a comparably simple way to program a competitive solver which is why we decided to implement it ourselves to see if we could replicate the results and combine the entire project with Christoph Kirsch's Selfie.
There are different ways to construct a solver, a complete solver can guarantee

a higher probability that none of the bugs it was designed to find occurs. Our solver being an incomplete solver sacrifices some of the certainty that a program is bug-free for speed. It should, therefore, still guaranteed a very high probability of the program being bug-free and theoretically be a lot faster in finding these bugs, especially in combination with the backwards working propagation-based algorithm to improve the chosen inputs iteratively. This is done by adapting the inputs to the desired output via Consistent and Invertable Value calculations.
Monster also supports two complete SMT-solvers, Boolector and Z3, for benchmarking purposes and to check its correctness.
The current iteration of Monster is competitive and can even outperform its competitors when the programs are shorter and more straightforward. Its performance drops hard when longer and more complex programs are being used, though, and it is sadly borderline unusable in these cases.
While the current performance does not make it a viable alternative to the other solvers, it is a suitable base for future work done in rarity simulation, and especially given the short timeframe and being constructed by bachelor students, it is performing surprisingly good. Even though the accuracy should theoretically be lower Monster has not shown this and always reached the same results as Boolector and Z3, showing that the drop in accuracy is not as significant.

1.1 The Problem

With programs getting more and more complex, it is tough to argue on a bit level that a program is bug-free. Even straightforward programs, which appear to be correct at first glance, can have glaring errors or cases of undefined behavior.
Take this elementary program for example:
It increments every single number in an array by one. Simple enough, and it will

```
increment_array(int array a) {
  int idx = 0;
  while idx < a.length {
    a[idx] = a[idx] + 1;
    idx = idx + 1
  }
  return a;
}
```

Fig. 1. Example program

do its job most of the time.
Suppose one of the values was to overflow though this might lead to problems down the line.
This is, of course, a straightforward example. Still, it is supposed to show that since the complexity of programs has grown so much over the years and keeps

growing, it is becoming increasingly challenging for programmers to find all the possibilities where programs could error.
Humans are not fit to find every possible error, and most of the time, only edge cases end up being tested. Even though new programmers are always told to write tests and test-driven development is the new norm, it is still humans that write these tests and think about the test cases, and this practice is prone to the same human errors that appear in the code.
A way to theoretically make sure that some bugs won't occur in a program is by testing it for every single possible input. It is not feasible to do this by executing the code, however. A program with a single 64-bit integer as input would have to be run 2^{64} times. The time complexity of this practice grows exponentially with the input size, and it is therefore almost impossible to do this for more extended programs or programs with massive inputs.
This is further complicated by not knowing how long to let the test run. One could execute the code for a single input forever since we can not predict if the code might crash in later iterations of the same loop.
It is, therefore, just not possible to test a program in this way. Instead, we execute code symbolically. We simulate the machine state for the current input and can drastically reduce the time it takes to run these tests.

1.2 Symbolic execution

A symbolic execution engine does the symbolical execution of a program. Instead of executing the program for every possible input, we execute it symbolically and decide which inputs lead to the execution of which branches in a program. We look for an assignment that leads to the execution of a path which in turn leads to a machine state that we defined as an error. If this is possible, we just proved that the chosen assignment leads to an error, and the program is therefore not bug-free.
executing these programs symbolically is much faster than testing them with every possible input and therefore much more manageable. It is still a very time-consuming process, but modern symbolic execution engines are becoming faster and faster, and competitions are being held worldwide. The time taken is further reduced since we only take branches into account that can cause errors and limit the set of possible inputs for a given branch by adding constraints that will be explained later.
It is still not feasible for extended programs to do this for all the possible inputs since loops can run indefinitely. Still, there are multiple approaches to this to reduce the probability of missing a bug. Once finished, the engine either returns the exact inputs that lead to an error or confirm that for every single possible input (or at least for most of them), there is not a single one that leads to a bug, therefore confirming that our program is bug-free in the regards to the bugs it was designed to find or at least with a very high probability.

1.3 Complete and incomplete symbolic execution

One approach to symbolically execute is creating a Satisfiability Modulo Theory (SMT) Formula out of the given program and trying to find an assignment for it such that the Formula is true. Such an assignment is called satisfiable. Most competitive solvers use bit blasting to search for these, an approach that is very efficient when used with smaller inputs but scales very poorly for growing input sizes. To tackle this problem, bit blasting relies heavily on pre-processing to reduce the input significantly. If this can not be done, though, the run time suffers greatly.
The developers behind Boolector, a state of the art SMT-solver which builds on the same principle, had a different idea, though.
In 2017 Aina Niemetz et al. introduced a new way of implementing a symbolic execution engine [1]. It was a fascinating approach since instead of using a complete symbolic execution engine which pretty much all of the state of the art symbolic execution engines are she suggested using an incomplete symbolic execution engine.
Complete symbolic execution engines use complete solvers, which do not only confirm if the found assignment leads to an error but also if there is no such assignment and the Formula can not be satisfied. Vouching that the program is bug-free in regards to the bugs it was programmed to find. Providing proof that there is no such assignment is, of course, much more challenging than proofing that there is such an assignment, and this is what incomplete symbolic execution engines use to their advantage.
An incomplete symbolic execution engine can guarantee even less that a program is actually bug-free. Using an incomplete solver only searches through a small subset of possible inputs, and it is possible to miss the satisfiable assignment, which leads to an error. But since we can reduce the set of solutions we have to symbolically execute by quite a bit, we are theoretically faster, and by only searching where we expect an assignment that leads to an unwanted machine state, we reduce the chance of falsely assuming that the program is bug-free significantly.
If no satisfiable assignment can be found, complete solvers terminate after looking through the entire set of possible inputs. Since incomplete solvers do not do that, they terminate after reaching a previously defined timeout.
This, in combination with a few other improvements made this new approach look like a comparable easy way to construct a symbolic execution engine which can compete with some of the state of the art engines and solvers.

1.4 Lazy and Eager

Symbolic execution engines not only differ in their usage of complete and incomplete solvers but also in how they evaluate the given code. Eager evaluation techniques like bit blasting translate the entire SMT formula into propositional logic and then decide with an SAT solver if the Formula has a solution.

On the other hand, a lazy approach only evaluates parts that can be executed after each other. This is done by an SAT solver checking consistency across the Formula and then assess only parts which are actually reachable.
This lazy approach removes the time it takes to translate potentially useless parts of the Formula. Since translating this Formula is exponential to the Formula size, this can potentially save a lot of overhead and is why most SMT solvers use the lazy approach.

1.5 Monster

Our incomplete symbolic execution engine is called Monster and is based on the paper mentioned above by Aina Niemetz.
Monster was designed to work in union with Selfie, a self-compiling compiler, self-executing emulator, and self-hosting hypervisor. Selfie is a project of the Computational Systems Group at the Department of Computer Sciences of the University of Salzburg in Austria and part of every CS students curriculum here in Salzburg.
Selfie compiles C* files, a subset of C, to RISC-U, a subset of the RISC-V instruction set. RISC-U has 14 different instructions: LUI, ADDI, LD, SD, ADD, SUB, MUL, DIV, REMU, SLTU, BEQ, JAL, JALR, and ECALL.
Monster itself was entirely written in Rust, a language that guarantees memory safety while being extremely fast and stable. The example files included are, of course, written in C* and are then compiled by Selfie to RISC-U code before being fed into Monster.
Feeding Monster compiled machine code was a conscious design decision that has the significant upside of a bug-free program not being compromised by a compiler introducing new bugs. While unlikely, it must not happen in this case. Another upside is that it can consume code generated by other compilers and is not dependent on a specific programming language. This has not been implemented yet but might be done in the future.

1.6 Concrete and symbolic Values

To keep track of the current machine state and if parts of the program are reachable, we must keep track of values assigned to variables. Assigning a concrete value to a variable means knowing exactly what value the variable currently has independently of any inputs. An example is $x = 7$. Assigning the value 7 to x and it, therefore, being concrete.
A variable with a symbolic value is a variable to which a value was assigned, but it was not a concrete value. An example is assigning x the value of a read call and then assigning $y = x + 2$. Both x and y are symbolical values since both are dependent on user input.
These symbolic values might change though. For example if we enter the following if condition: if (x == 2) we can assume that inside the condition $x = 2$ and $y = 4$ therefore both being concrete values now.

1.7 Bugs

Monster can currently detect two different types of bugs. First, a division by zero. In some programming languages, this is a valid operation, but in our case, a division by zero is illegal and will therefore lead to a crash. Second an exit code greater than zero. In Selfie, any exit code greater than zero indicates that a problem of some sort occurred and is therefore relevant to us. For example, exit code 19 means that Selfie is out of virtual memory.
On top of that, we also detect so-called undefined behavior. An example would be a write call on uninitialized memory. We have no idea what is written there and can therefore not predict what happens next, resulting in undefined behavior.

It is worth mentioning here that symbolic execution engines can not detect every single possible bug by design! An example is arrays. Accessing an array like this, for example, a[i], is only possible if i is a concrete value. Dynamic memory access is not supported since we would have to know precisely what is being read, which is impossible. Another example where this problem occurs is dereferencing pointers.
Another problem in symbolic execution is very long or endlessly running loops. As Alan Turing showed in the halting problem, there is no general algorithm that can decide whether an arbitrary computer program and input will finish running or run forever.
Not knowing if a program ever stops leads to us not knowing if there will be a bug somewhere in the future. At some point, the symbolic execution engine has to abort the current loop's execution, and deciding when this point is reached is not easy. Loops can, therefore, by design be undecidable for us.

1.8 Collaboration

This project was not done by myself but in collaboration with Alexander Linz and Christian Mösl. We cooperated in every aspect, and each of us spent some time programming parts that are now part of the other's Theses or helping each other out when we were stuck. In the end, each of us wrote about the part of the project which he spent the most time on. Alexander Linz spent most of his time on the engine and benchmarking. Christian Mösl was responsible for the engine and the solver. And I spent most of my time programming the solver.
I recommend reading Alexander Linz's and Christian Mösl's Theses prior to this one to guarantee a deeper understanding of the subject. While it is a standalone Thesis, I am only going to briefly touch on some subjects which are explained in far greater detail in their papers.
I also want to give special thanks to Fabian Nedoluha, helping us at the beginning of the project, and our Advisor Christoph Kirsch who we could always ask for advice.

2 SMT

Satisfiable Modulo Theory (SMT) is a first-order logic formula used "to decide whether a formula over Boolean variables, formed using logical connectives, can be made true by choosing true/false values for its variables."[2] This is a constraint satisfaction problem known as SAT and is NP-complete.
Therefore, we can use SMT to describe if a particular machine state is reachable in regards to the chosen input. Hence Monster transforms every program it is being fed into an SMT-Formula before handing it over to the solver.
To understand how the solver works first, we need to look at SMT-Language and its Formulas and how they look and are built.

2.1 Solutions to Formulas

There are two possible solutions to a Formula, either satisfiable (Sat) or unsatisfiable (Unsat). A satisfiable assignment is an assignment for the variables such that the Formula is true. This assignment is also called a witness.
On the other hand, it is Unsatisfiable if there is not a single assignment for the variables to make the Formula true. This is, of course, much harder to prove since a single witness will not suffice. The entire set of possible assignments needs to be checked if it may be a satisfiable assignment.

2.2 The Basics

Lets take a look at a simple example of how such a Formula might look like.

```
(set-option :print-success false)
(set-logic QF_UF)
(declare-const p Bool)
(assert (and p (not p)))
(check-sat)
(exit)
```

Fig. 2. Example SMT Formula

We are going to ignore the first two lines for now. In the third line, the constant boolean p is declared, and in the fourth line, we assert $p \wedge \neg p$.
In the fifth line, we then check if what we have defined is satisfiable. Obviously, there is no satisfiable assignment for p such that $p \wedge \neg p \approx \top$ and this snippet is unsatisfiable.
Since we only have a single variable, the set of assignments is minimal, only

$p \approx \top$ and $p \approx \bot$. The complexity grows exponentially with each added variable. This is especially relevant when looking at 64-bit Integers since these have 2^{64} different assignments and represent the variables in our code.

2.3 Constraints

An essential part of SMT Formulas is that they include the constraints for the inputs that are defined in the code.

```
if (*x > 1) {
  if (*x < 6) {
    *x = *x - 2;
    *x = 10 / *x;
  }
}
```

Fig. 3. Constraint example

Taking a look at the code snippet, it is evident that to execute the code in the 2nd if condition $x > 1 \land x < 6$ therefore severely limiting the set of inputs that reach the code in the 2nd if condition.
These constraints lead to the solver not trying values for the inputs that would not reach the line, leading to an error since the branch in question would not be executed for the chosen inputs. This dramatically reduces the runtime, especially the more constraints there are.

2.4 Actual Formulas

To show how exactly a SMT-Lib Formula is created out of code, we will take a look at a simple code example first. Anyone who has dealt with C or C++ code before should have no problem to understand this example, but for anyone not used to the C syntax, it will be explained here.

A C* code example
In the first three lines, we declare a variable called x, which is of type uint64_t*, a pointer to an unsigned 64-bit integer. We then allocate 8 bytes of memory using malloc and write the user input to the allocated space. Whenever we use $x*$, we are dereferencing the pointer, referring to the memory stored where the pointer is pointing, not the pointer itself.
The rest of the code should be self-explanatory. The only parts worth mentioning are the division by x in line five and the return call in the last line.

```
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. 4. Trivial C* code example

These two are our exit conditions when looking at errors and, therefore, interesting for Monster. In line five, a possible division by zero leads to an error, and in the last line returning any $x > 0$ is an error.
Monster's job is to check if there are inputs that lead to these errors and return one such assignment.

The example code as SMT-Lib

```
(declare-fun x0 () (_ BitVec 64))
(assert (= x0 (_ bv1 64)))
(declare-fun x1 () (_ BitVec 64)); "read(0, 69640, 8)[0 - 8]"
(declare-fun x2 () (_ BitVec 64))
(assert (= x2 (_ bv1 64)))
(declare-fun x3 () (_ BitVec 64))
(assert (= x3 (ite (bvult x1 x2) (_ bv1 64) (_ bv0 64))))
(declare-fun x4 () (_ BitVec 64))
(assert (= x4 (_ bv0 64)))
(declare-fun x5 () (_ BitVec 64))
(assert (= x5 (ite (= x3 x4) (_ bv1 64) (_ bv0 64))))
(declare-fun x6 () (_ BitVec 64))
(assert (= x6 (bvand x0 x5)))
(check-sat)
```

Fig. 5. Example SMT-Lib snippet refering to code shown in 6

This is only a snippet of the entire SMT-Lib Formula that is being created out of the code shown in 4. Since multiple different errors can occur in different branches with different restraints, multiple Formulas can be created for a single

```
uint64_t main() {
    uint64_t *x;
    x = malloc(8)
    read(0, x, 8);

    if (*x < 1)
}
```

Fig. 6. Actual code the SMT-Lib snippet is referring to.

program. In this case, we will only take a look at the SMT snippet, which refers to the code shown in 6.
First, the function $x0$ is being declared, a 64-Bit Vector, as is every function in this snippet, and it is asserted that $x0$ equals BitVector 1. Afterward two more functions are declared $x1$ and $x2$, with $x1$ being the value that is put in by the user and $x2$ once again being asserted that it equals BitVector 1.
In the next two lines, $x3$ is being declared and afterward asserted that it either has the value 1 or 0 depending on if $x1$ is smaller than $x2$. This represents the if condition in line 5 as can be seen in 4.
$x4$ and $x5$ get declared, and it is asserted that $x4$ has the value 0 and $x5$ 0 or 1 depending on if $x3$ and $x4$ are equal.
The last function being declared is $x6$, and it is asserted that its value is equal to $x0 \wedge x5$. Then the satisfiability is being checked. This part of the SMT-Lib Formula only checks for reachability and not if there is a satisfiable assignment to cause the division by zero. If there is no satisfiable assignment to actually reach the if condition, there is no use in trying to find a satisfiable assignment such that a division by zero occurs.

3 Monster

We will now take a look at how exactly all of these concepts come into fruition in Monster.
Before the solver can try to find a satisfiable assignment for a Formula Monsters engine needs to prepare this Formula for the solver and choose a so called Candidate path.
Most programs can take different paths depending on the input. For example, in our code snippet 4, different values of x can make a difference in what branches in the code get executed and what errors can occur.
The limiting factor in incomplete symbolic execution engines is time. We only get as many chances at trying different inputs as we can execute the chosen path in the given timeout time. Therefore, we want to select a Candidate path that is fast to execute, leading to faster symbolical execution and more inputs tested before getting timed out. Secondly, a path that can actually be executed.

Us getting to test more inputs leads to a greater chance of actually finding a satisfiable assignment. Therefore, a higher probability of finding bugs quicker and not executing paths that are not reachable minimizes time loss.
More information on how exactly Candidate paths are chosen, can be found in Christian Mösls paper on the engine of Monster.
Once a path has been chosen, it is translated to SMT-Lib and handed over to the solver, which will try to find a satisfiable assignment for it. If one can be found, the solver returns the assignment, and Monster is done. Otherwise, if none can be found in the given timeframe, the engine will select a new path. We repeat this process until a satisfiable assignment is found, there are no more Candidate paths, or a certain execution depth has been reached. The third terminating condition is in the case of loops which could be executed forever.
If there are no more paths or we reach our max execution depth, the program is bug-free to us. Of course, there is always a slight possibility of there being a bug.
This is a rough overview of how the entire process looks like. Now we take a look at how exactly the solver works.

3.1 Formulas

A Formula is the representation of a program in Monster. To create such a Formula, the program is parsed line by line, and we construct a directed acyclic graph out of the RISC-U code where each instruction is either an edge, a node, or both.
There are three different node types: Constant, Input, and Operator. A Constant represents a constant value in the program, such as $x = x + 2$, where 2 is a constant value. An input is a user input into the program, and these are the most interesting nodes to us since these are the values that the solver will modify. An Operator is an operation on one or two Constant or Inputs like in the above example $x = x+2$ where we operate on the Constant 2 and the Input or Constant x.
The possible exit points of the program are the leaf nodes of our graph and the nodes where an error could occur. In our case, this is a return or the DIVU operation.
A return is the end of the program since it leads to no more code below it is executed. A division by zero, which we look for, has the same effect and is, therefore, another possible exit.

3.2 BitVectors

Every variable in our code is represented by a BitVector, a Vector consisting only of 1s and 0s. Since we use Unsigned 64 Bit Integers in our case, a BitVector has a length of 64.
BitVectors are the representation of input and constant nodes in Formulas and are, in our case, implemented as an unsigned 64-bit integer. Therefore we can represent numbers in a range from 0 to $2^{64} - 1$ and denote a BitVector that has

the maximum possible value of $2^{64} - 1$ by *ones*.
We implement various operations for BitVectors, including the basic arithmetic operations of addition, subtraction, multiplication, and division. All of these operations are wrap-around, and we defined that the division through zero has the result *ones* since this is as SMT-Lib defines it.
On top of that, we support Bitwise And, Or, Xor, left shifting, right shifting, Negation and Not.
The only other operations which may need explanation are ctz, modinverse, addo, and mulo. Ctz counts the trailing zeros of the BitVector. Addo and mulo check if the addition or multiplication of two BitVectors overflows. And modinverse returns the modular inverse of the given BitVector.
All of these operations are necessary to aid us in the calculation of Consistent Values and Invertible Values that are important in our algorithm.

3.3 Consistent Values

One of the key features this algorithm relies on is backpropagation and, therefore, the ability to adapt the inputs of an equation to the desired output. Specifying, for example, that the solution of $x < y$ has to be true, we can guarantee that the nodes following it can rely on the fact that x is smaller than y. To be able to do this, we have to be able to calculate Consistent and Invertible Values.
Consistent Values are calculated when the desired result of an equation is known, but neither of its two inputs is a constant value.
Suppose we take a look at the example earlier that $x < y \approx \top$, we now have to choose an arbitrary x such that there is at least one y such that $x < y$. Since we can always choose $y = x + 1$, one might think that any x might suffice, but the highest number one can represent with 64 bits is $2^{64} - 1$. So, in this case, any $x < 2^{64} - 1$ is a valid Consistent Value, and since there is more than one possible solution, we choose it randomly.
It is to note that it is always possible to compute at least one Consistent Value for any x, y, z and any operation $\_$ in the form of $x\_y = z$.

3.4 Invertable Values

Invertible Values are calculated if the result and one of two variables in the equation are known. Looking at our earlier example of $x < y \approx \top$ where we calculated a Consistent Value for x, which is a random number that's smaller than $2^{64} - 1$.
Since x is not just symbolic anymore, but we assigned it a concrete value, we can now calculate an Inverse Value for y. This is simply done by choosing from a range of numbers such that $x < y \leq 2^{64} - 1$. This time around it might not be possible, though, to calculate such a value. It is always possible to calculate an Invertible Value if the other value is a Consistent Value since this is the definition of Consistent Values. But if the other value is a constant, for example, it might not be possible to compute an Invertible Value at all!
In some cases, calculating an Invertible Value is also not quite as obvious and

easy as in our given example. Take the following division for example: $x/3 = 4$. The obvious and for some only solution to this equation is 12, but since we are using integers and not doubles or floats, 13 and 14 are also correct results since if we divide any of these numbers by 3 and do not consider decimals, the result is 4.
This is the main challenge when calculating Invertible Values, making sure to consider every possible solution and return each possible solution with the same probability. While it is easy to prove that the value we chose is an Invertible Value, it is not easy to prove that any value we could have returned is an Invertible Value and that we considered every single Invertible Value.
We support Invertible Values for eight of the 14 machine instructions in RISC-U. These are ADD, SUB, MUL, DIVU, SLTU, Bitwise AND, NOT, and Equals. We never need to calculate an Inverse or Consistent Value for any of the other instructions and therefore only support these eight.

Operation	Invertability Condition
$x + s \approx t$	$\top$
$x - s \approx t$	$\top$
$x * s \approx t$	$(-s \vee s) \wedge t \approx t$
$x \div s \approx t$	$(s * t) \div s \approx t$
$s \div x \approx t$	$s \div (s \div t) \approx t$
$x < s \approx t$	$s \neq 0$
$s < x \approx t$	$s \neq ones$
$x \& s \approx t$	$t \wedge s \approx t$
$x = s \approx t$	$\top$

Table 1. The Invertability Condition for each of the eight supported instructions [1].

As you can see, Add, Sub, Not, and Equals are always Invertible. The others, on the other hand, not so much. An easy-to-grasp example is SLTU. It is of course not possible to compute an $x < s$ if $s = 0$. In every other case, it is possible to compute such an x.

If it is possible to calculate an Invertible Value, table 2 shows how it is calculated. Some of them are just solving a formula for an unknown x.

3.5 The Algorithm

The algorithm Monster is based on is a propagation-based local search algorithm. Local search because we try to improve the inputs with each iteration until a satisfiable solution is found. Furthermore, propagation-based because we propagate the solutions through the graph.
The critical part here is that we do not only propagate from the inputs to the

Operation	Invertibility Condition
$x + s \approx t$	$t - s$
$x - s \approx t$	$t + s$
$s - x \approx t$	$s - t$
$x * s \approx t$	$((t >> s.ctz) * (s >> s.ctz)^{-1}) \vee (r \wedge (ones << (64 - s.ctz)))$
$x \div s \approx t$	some $s * t \leq x < s * t + s$
$s \div x \approx t$	some $s \div t - t < x \leq s \div t$
$x < s \approx \top$	some $x < s$
$x < s \approx \bot$	some $s \geq x$
$s < x \approx \top$	some $s < x$
$s < x \approx \bot$	some $x \geq s$
$x \& s \approx t$	some $x \vee t$
$x = s \approx \top$	s
$x = s \approx \bot$	some $x \neq s$

Table 2. The Invertable Value calculation for every single one of the eight supported instructions [1]. The Bitvector r is in this case a randomly generated Bitvector.

outputs but also backward from the outputs to the inputs. This means that we choose our desired outputs and then adapt the inputs to the outputs.
An assignment is satisfying if the root node is true. Therefore we always set our root node to true and continue from there.
We repeat this process until we have either found such an assignment or run out of time.

3.6 Path Selection

Since there remains some randomness in selecting input values and we only modify a single input on each iteration, the decision on which input to change is crucial.
Whenever we have calculated the result of the Formula with the newly chosen values, and the root node does not equal 1, we then start the process of back-propagation.
A simple example would be a node with a logical and where one of the input nodes is 1 and the other 0. It is evident that changing the input on the side that is not true yet has a larger chance of success since modifying an input that contributes to a satisfiable assignment makes no sense.
Another important part is following paths that lead to an input. An operation with only constants will always have the same results independent of the chosen inputs.
Especially important are so-called essential inputs. An essential input n is an input that has to change such that the operator node o can assume the target value t [1]. This will change an input that leads to the assignment not being satisfiable and possibly improving our possible solution.
If neither of the branches is constant or essential, we select one of them randomly

since, in this case, we can not decide which branch is more promising.

Using this technique, we guarantee to select the branch which gives us the highest possible probability of changing an input that leads to a satisfiable assignment and therefore improving the assignment iterative step by step.

3.7 Usage of Consistent and Invertible Values

Once we decided changing which input gives us the highest chance of success, we calculate a new assignment for the input node. If possible, we compute an Invertible Value and assign it to the node. Otherwise, if an Invertible Value can not be computed, we calculate a Consistent Value. There always is a 10% chance we use a Consistent Value even if the computation of an Inverse Value is possible. This is done to prevent Monster from possibly getting stuck. Assigning an Invertible Value to a node will only change the value of this single input, and sometimes, as can be seen in the example I am about to give, this does not suffice. Assigning it a Consistent Value, on the other hand, will lead to new values being assigned to both nodes with a high probability.
An example is an add instruction in combination with an equals of the form $(x + 2) + x \approx t$ with $t \approx 4$ and currently $x \approx 0$. Propagating back to the right-hand side of the input, in this case, x and setting it to an Invertible Value, will lead to x being assigned the value 2. The problem is that x will be assigned 0 again on the next iteration, leading to an endless loop.
Propagating to the assignment's left-hand side and choosing an Invertible Value leads to the same problem. Assigning 2 to x and it being reverted to 0 next iteration.
Only choosing a Consistent Value for x here can lead to the correct assignment of $x = 1$ by either directly assigning it 1 or by assigning it 3 and then assigning it 1 on the next iteration.
Afterward, we calculate the new results and check if this leads to our root node being true. If it is, we are done; otherwise, the backpropagation starts again, and we continue unless the timeout was reached.

3.8 Experiments

In the end, it all boils down to the process of iteratively improving our randomly created solution by modifying a single input every iteration and checking if this did the trick. Although this is a straightforward process, we got surprisingly good results in our benchmarks.
We benchmarked Monster against two different SMT solvers. Boolector, designed by a team which Aina Niemetz was a part of, and Microsoft backed Z3. All three of them operated on the same SMT Formula created out of the same C* file and were obviously done on the same hardware under the same conditions.
Currently, Monster provides excellent results when used on our example programs. Since these programs are comparably small, we quickly find a path with an assignment that leads to an error. This is the main thing that slows Monster

down: finding assignments for Formulas with no satisfying assignment. Since we run into a Timeout and the timeout has not yet been appropriately tuned, this costs us a lot of time.
One case where this hit us especially hard is programs with no satisfiable assignment the first few times the loop runs. Since each loop costs us 3 seconds, this leads to abysmal performance.
When only looking at our example files through Monster performs very well. It can compete with Boolector and outperforms Z3 by a considerable margin!
In the more simple file count_up_down-1.c the solver has to find a value such that a division by zero occurs. Symbolically executing it leads to the results shown in 7.

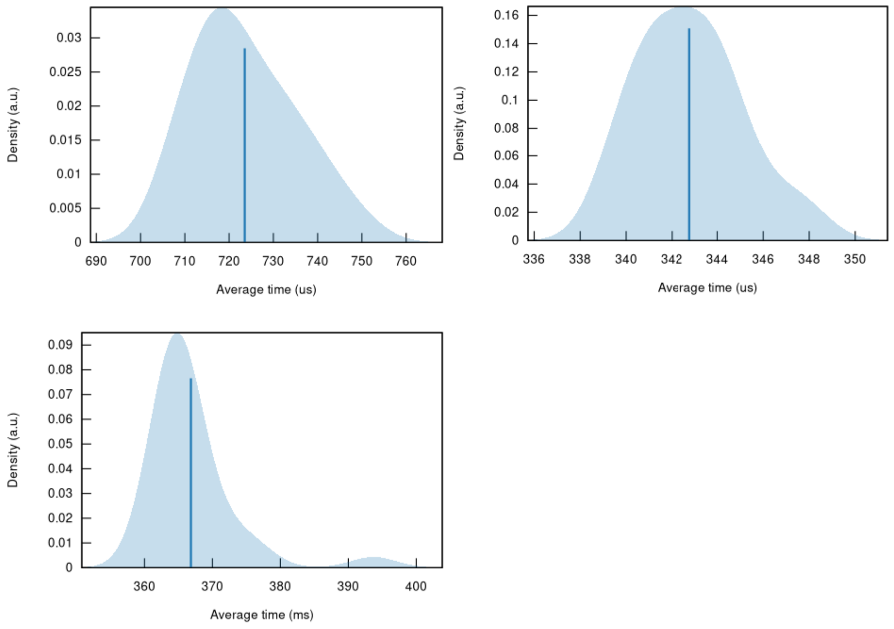

Fig. 7. Benchmarks of a simple program. Boolector being top left, Monster top right and Z3 bottom left.

As we can see, Monster performs almost twice as fast as Boolector in this example. Z3 is even further behind, needing about a thousand times longer. Of course, it's still ms, so almost not notable, but it is taking significantly longer than Monster and Boolector. This is primarily the case because of Z3's huge overhead, leading to an abysmal performance on smaller files.

Next, we are taking a look at a more complicated program sum01-1.c. As above, the goal is to find an assignment that a division by zero occurs. in this example, it is far more complex, though, and the resulting runtime as shown in 8 displays this.

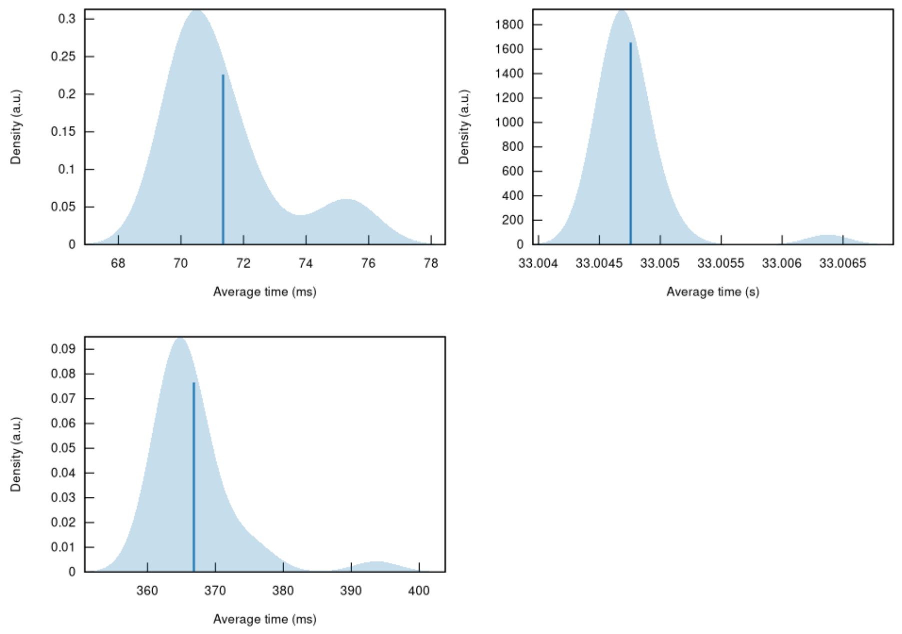

Fig. 8. Benchmarks of a more complex program. Boolector being top left, Monster top right and Z3 bottom left.

As we can see, Boolector is still performing the best by a large margin, but the gap to Z3 has shrunk significantly since Z3s overhead is not as punishing in this example. Monster, on the other hand, is performing substantially worse than before, needing over 30 seconds. This is a perfect example of our timeout slowing us down.
It is worth noting that the developers of Boolector, the inventors of the algorithm Monster is based on, did not implement their idea into Boolector since complete symbolic execution engines still outperform incomplete engines. Whenever there is no satisfiable assignment, incomplete engines lose a lot of time and can therefore be faster in some edge cases but generally get outperformed. They did implement it into Bitwuzla, though, which did not perform as well as Boolector.
Most programs used to benchmark symbolic execution engines in competitions are bug-free, and this favors complete symbolic execution engines since this slows down incomplete engines.

For more benchmarks and a deeper analysis of the results check out Alexander Linz's paper on the System of Monster.

4 Conclusion

In Conclusion Monster does exactly what we envisioned it to do. It detects the bugs we programmed it to find and while it does need longer in some cases it can compete and even outperform other SMT-solvers in some other cases. While it is not a commercially useable symbolic execution engine it does perform incredibly well when used on our example files and this is what it was set out to do.
Especially given it was programmed in six months by three bachelor students it is an impressive piece of work but there are of course improvements to be made.

4.1 Possible Improvements

Timeout A very obvious and straightforward improvement is reducing the time it takes until we return a timeout. Currently, we declare an SMT Formula unknown if we do not find a satisfiable assignment in three seconds. This is way too long and one of the main reasons we are so much slower in some cases.
We thought about various ways how to find the sweet spot where we can be sure not to miss any satisfiable assignments and get the best possible performance. One possibility was somehow tieing the duration of the timeout to how complex it is to find a satisfiable assignment. However, it is tough to predict this complexity. The number of branches, loops, and the length of the program can give a general idea, but we did not test yet if these factors correlate.
These metrics could lead us to optimal timeouts for each program, limiting the time we look for a satisfiable solution when the probability of finding one is already very low. Since this is the most significant time loss currently, it is also an easy way to improve the runtime of our solver.

Ternary Vectors Something that was mentioned in the paper and implemented in Bitwuzla is Ternary Vectors. Ternary Vectors are Vectors that have three different states per Symbol instead of two. Being 1, 0, and *, *, in this case, is an unknown bit. For example, the Ternary Vector *0* could be 000, 001, 100, and 101.
We implemented Ternary Vectors as a pair of BitVectors, one called high and the other low. The previous Ternary *0* being represented by low: 000 and high: 101.
Taking these Constant Bits into consideration when calculating Invertible or Consistent Values greatly reduces the set of possible results we could calculate and not lead to a satisfiable assignment. An Example given is the following: Consider a formula $(1110\&x) \not\approx 0000$. The least significant bit (LSB) of the left-hand side operator forces the LSB of the bitwise and always to 0. Therefore one does not need to try 0000 and 0001 since the result is the same [1]. With Ternary Vectors, we can integrate this information into the calculation of Inverse and

Consistent Values and save time this way.
Our implementation worked flawlessly, but we ran into some unforeseen problems calculating Invertible and Consistent Values. Because when considering these Constant Bits, the calculation of Constant and especially Invertible Values becomes incredibly complex.
For example looking at the Invertibility condition of $s \div x \approx t$ which is $s \div (s \div t) \approx t$ becomes:
$s \div (s \div t) \approx t \wedge (t \not\approx ones \Rightarrow x^{hi} >_u 0) \wedge ((s \not\approx 0 \vee t \not\approx 0) \Rightarrow (s \div x^{hi} \leq_u t) \wedge \exists y.(mcb(x, y) \wedge (t \approx ones \Rightarrow y \geq_u 0 \wedge y \leq_u s \div t) \wedge (t \not\approx ones \Rightarrow y >_u t + 1 \wedge y \leq_u s \div t))))$.
We could not find a feasible way to prove the existence of such a y, and the paper sadly did not mention it either. It is not sure how much time Ternarys save, but we are still looking at ways to implement them.

Monster is still an ongoing project, and as of right now, there are even more people working on it than before. If you have any idea how to fix one of the mentioned issues, are in general just interested in the subject or want to contribute to the rarity simulation check out Monster on Github and collaborate with us.

References

[1] Aina Niemetz. "Bit-Precise Reasoning Beyond Bit-Blasting/eingereicht von Dipl.-Ing. Aina Niemetz, Bsc". PhD thesis. Universität Linz, 2017.

[2] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Communications of the ACM* 54.9 (2011), pp. 69–77.